

Second Edition

Helen Borrie

The **Firebird Book**

A Reference for Database Developers

*An essential guide for developers and administrators
working with any version of the Firebird open source
relational database management system*

IBPhoenix Publications

A large, stylized, 3D-rendered Firebird logo in a light gray color, serving as a background for the title. The logo depicts a bird in flight, with its wings spread and its body curved as if diving or swooping. The bird is composed of thick, rounded lines that give it a sense of depth and movement.

The **Firebird Book** *Second Edition*

A Reference for Database Developers

by Helen Borrie

IBPhoenix Publications

The Firebird Book (Second Edition): A Reference for Database Developers

Copyright 2011, 2012 by Helen Borrie and IBPhoenix

All rights reserved. Except where content is indicated explicitly as copied from documents licensed under open document licensing, no part of this work may be reproduced in any form or by any means, electronic or mechanical, without the prior written permission of the copyright owners and the publisher, IBPhoenix Publications.

Reviewers were Aage Johansen, Dmitry Sibiryakov, Thomas Steinmaurer, Calin Pirtea

Design of the cover and Part headers is by Stella Damaschin.

Body text is 10pt Garamond.

AUTHOR'S FOREWORD

This, the second edition of *The Firebird Book*, has been gestating for more than seven years, since the first edition in August, 2004. The first book's milestone was Firebird 1.5. Its objective was to provide an up-to-date working reference for developers, whether coming to Firebird from another database management system or moving into it with past experience in its closed source cousin, InterBase.

In the intervening years, Firebird has evolved through three further major versions: 2.0, 2.1 and 2.5. At the time of this writing, all three are still under maintenance and the re-architected version 3.0 is in alpha development. Maintenance of version 1.5 ceased in 2009.

The energy and dedication of Firebird's development team continue to be a source of wonder and inspiration to me and, I'm sure, to anyone else who works with Firebird. The polyglot Core team—Dmitry Yemanov, Vlad Khorsun, Alex Peshkov and Adriano dos Santos Fernandes—works closely and tirelessly under Dmitry's chieftainship. Pavel Cisar and Philippe Makowski lead quality assurance testing and Claudio Valderrama is responsible for scrutinising incoming code changes. Claudio's hand is often seen in improvements to the command-line tools, as well.

Because Firebird is distributed free, in all senses of the word, it produces no revenue to pay salaries to the code workers. Except for the QA people, these volunteers depend on Firebird Foundation grants to supplement their incomes to make space for their Firebird work.

The Firebird Foundation is funded by donations, cash sponsorships and membership subscriptions from individuals and companies that perceive and acknowledge the benefits of using Firebird and keeping it under active development. You can view the current list of sponsors at <http://www.firebirdsql.org/en/sponsors/>. Bless 'em all!

Some sponsors contribute much more than cash. IBPhoenix, for example, contributes the manpower for QA, binary builds and basic documentation. Neither this book nor its predecessor would have come to fruition without IBPhoenix funding. Broadview Software hosts the issue tracker, pre-release and build servers while IBSurgeon funds the hosting and development of the Firebird web site at www.firebirdsql.org.

Amongst the volunteers who contribute their time and expertise to the Firebird Project without grants I must name Paul Vinkenoog, who has been almost single-handedly responsible for filling the free documentation gap between the legacy InterBase manuals and the release notes that are distributed with the binary kits. It is a vast and ongoing task that is often thankless. At the time of writing this, Paul is working with Dmitry Yemanov to produce comprehensive on-line SQL documentation that will not be inhibited by the copyright issues that still prohibit reuse of the legacy InterBase material.

Under the Firebird Project umbrella, volunteers also develop and maintain several of the drivers and language interfaces for Firebird—Jaybird (for Java), the Firebird .NET providers, the ODBC driver and the interfaces for PHP and Python.

If you are coming to Firebird as a newcomer, welcome! I wish you a long and happy experience with our software. To you and to those devotees who are freshening up your Firebird experience, I wish you great satisfaction from your use of this Second Edition of ***The Firebird Book***.

Helen Borrie (author) December 2011

ABOUT THE AUTHOR

Helen Borrie is a contracting software engineer who doubles as a writer and technical editor. She has been involved with database development for 30 years and with Firebird and its ancestors since 1996.

Helen is an active member of the Firebird online support community and a founding member of the Firebird Foundation, incorporated as a non-profit organisation in New South Wales, Australia, in 2002.

INTRODUCTION

What Is Firebird?

Firebird is a powerful, compact client/server SQL relational database management system (RDBMS) that can run on a variety of server and client operating system platforms, including Windows, Linux, MacOSX and several other UNIX platforms. It is an industrial-strength RDBMS that features a high level of compliance with SQL standards, while implementing some powerful language features in the vendor-specific sphere of procedure programming.

Who Needs This Book?

Developers with some database experience, who may be moving to client/server for the first time, will discover all of the basics they need to become productive with Firebird in this book. Although not a primary tutorial on SQL or database design, this guide emphasizes good client/server RDBMS design practice and documents Firebird's SQL definition, manipulation, and programming language sets with plenty of detail, insider tips, and examples.

Firebird is serious software designed for deployment on networks small and large, including some useful capabilities for stand-alone configurations. Its small footprint makes it easy for sole developers to do large enterprise development from a home office. For the database administrator or system designer, the book is a basic installation, configuration, tuning, security, and tools resource. Firebird's strengths and high compliance with standards make it an attractive medium for university-level IT study. The book will serve amply as a resource for second- and third-year computer science students using Firebird for assignment work.

For those who have been using older Firebird versions or InterBase until now, the Firebird Book introduces the security, language, and optimizer enhancements that were added with the "2-series" releases: 2.0.x, 2.1.x and 2.5.x.

Where to Find What You Need

Part One is a "101 course" for those who are new to Firebird. There, you will find the basics for understanding the client/server architecture, installing the software, getting a network client up and running, and for essential operations. The part ends with a chapter about migrating from older versions to the "2-series".

In **Part Two**, you will find a detailed reference to each of the SQL data types supported in Firebird. There is a chapter for each class of data type—numbers, date/time types, character types, and so on—with plenty of tips for using them.

Part Three examines the database and the various objects and relationships in detail. The SQL language subset for creating, modifying and removing objects is data definition

language (DDL). Its statement syntaxes and usages are covered in this section. It also includes an introduction to some useful features of the administrative command-line toolset, *gfx*.

Part Four covers the many aspects of the data manipulation language (DML) subset of Firebird SQL that you will use to create, retrieve, change and remove data. The part ends with a detailed chapter on using the command-line *isql* tool to perform batch operations, query databases interactively and get information about them.

Part Five addresses transactions: how they work, how to configure them, and tips for using them in your application programs.

In **Part Six** you will find out about server-side programming with Firebird: writing triggers, stored procedures and run-time blocks of procedural code in procedural SQL (PSQL), creating and using database events, and working error handling into your server code.

Part Seven addresses the database configuration and includes a more detailed look at the *gfx* toolset.

Part Eight covers the issues of securing and administering servers and databases: security, monitoring and backup.

Appendices and Glossary

Details of materials available in the appendices and glossary are as follows:

I: Internal and External Functions contains names, descriptions and examples for the internal functions, as well as external functions (UDFs) in the shipped `fb_udf` and `ib_udf` libraries.

II: Reserved and Non-Reserved Keywords tabulates all of the keywords, both reserved and non-reserved, that are applicable to the various versions of Firebird.

III: Context Variables lists the context variables available by version and explains their usage.

IV: Firebird Limits enumerates the various physical limits applicable to Firebird, by version.

V: System Tables and Views provides the data description specifications for the schema and monitoring tables¹ maintained by the Firebird server inside every database. It includes source code listings for some useful views you can create to interrogate the system tables.

VI: Character Sets and Collations is a full reference to the international character sets and language-specific collation sequences distributed with Firebird versions.

VII: Error Codes is a full, tabulated listing of the exception codes (SQLCODE and GDSCODE) defined in Firebird versions along with the associated symbolic constants and the message texts in English.

VIII: SQLSTATE Codes lists the SQL-standard status and error codes that were introduced in v.2.1.

IX: Database Repair How-To is a step-by-step procedure you can follow, using Firebird's own tools, when a database seems to be logically corrupted.

1. Monitoring tables do not exist in databases of ODS 11.0 or lower.

X: Default Disk Locations indicates where you could expect to find all of the components of a Firebird installation, by operating system platform.

XI: Healthcare for Databases describes good practices for keeping databases in good shape.

XII: Upgrade Scripts contains the scripts referred to in Chapter 5, *Migration Notes*.

XIII: Application Interfaces contains notes regarding some of the host language interface layers that are available to programmers writing applications with Firebird back-ends.

XIV: Resources is a compilation of resources available to Firebird users. It includes book and other documentary recommendations, along with links to some software tool and support offerings.

The **Glossary** provides summary descriptions of terminology and concepts you are likely to encounter on your journey into Firebird.

Firebird's Origins

Developed as an ongoing open source project, Firebird was the first new-generation descendant of Borland's InterBase 6.0 Open Edition code, which was released for open source development in July 2000 under the InterBase Public License (IPL).

The Firebird source code tree is maintained in a *Subversion* repository and developed on the international open source code foundry, SourceForge.net (<http://sourceforge.net>), by a stable team of professional developers, comprising both volunteers and others in specialist roles that are partly funded by grants from community and commercial contributions.



The Firebird RDBMS products and several associated modules are distributed completely free of registration or deployment fees under a variety of open source licenses.

The Firebird Project

The developers, designers, and testers who gave you Firebird and several of the drivers are members of the Firebird open source project at SourceForge, an amazing virtual community that is home to thousands of open source software teams. The Firebird project's address there is <http://sourceforge.net/projects/firebird>. At that site are the *Subversion* (SVN) source code tree and a number of technical files that can be downloaded for various purposes related to the development and testing of the codebases.

The Firebird Project developers and testers use an e-mail list forum, firebird-devel@lists.sourceforge.net, as their “virtual laboratory” for communicating with one another about their work on enhancements, bug fixing, and producing new versions of Firebird.

Anyone who is interested in watching the progress and feedback on beta development can join this forum.

Support for Application Developers and DBAs

Firebird has a powerful community of willing helpers, including a large body of active developers with many years of experience developing with and deploying Firebird and its InterBase ancestors. The esprit de corps among this large group is such that, having acquired the skills and learned the “inside tricks” from the advice of others, its members

stay around the lists as mentors to new users. The mainstream free support channel is the *Firebird-support* forum.

More specialized groups within the project's arena conduct specific forums—Java, Delphi and C++ Builder, tools, Visual Basic, ADO.NET provider, PHP, and more. The groups run as e-mail lists, and many are mirrored to a news server. The latest links to these forums can always be found at the main community website at www.firebirdsql.org/en/support/.

The *IBPhoenix* site also hosts an enormous volume of technical and user documentation, links to third-party tools, and a running news board of events happening throughout the Firebird community.

The FirebirdSQL Foundation

The FirebirdSQL Foundation (Inc.) is a non-profit foundation incorporated in New South Wales, Australia, that raises funds around the world for grants to developers working on general and special projects for extending, testing, and enhancing Firebird. Funds come in through private and corporate sponsorships, donations, and membership subscriptions. It provides a way for appreciative Firebird users to return a contribution for the free use of software and community support. Look it up at www.firebirdsql.org/en/firebird-foundation/.

Overview of Firebird

Firebird is a true client/server software, designed especially for use in local and wide-area networks. Accordingly, its core consists of two main software programs: the database server, which runs on a network host computer, and the client library, through which users on remote workstations connect to and communicate with databases managed by the server.



Administering and developing with a full-blooded SQL client/server RDBMS may be new territory for you. It can present a steep learning curve if this is your first venture into data management software that is purpose-built for multiple concurrent writers. Part Two of this guide provides an introduction to client/server concepts. If you find yourself getting lost in the following descriptions, you might like to visit Part Two first to acquire some context.

Firebird Versions

The original Firebird version 1.0.x binaries were developed by correcting and enhancing the modules, written in the C language, that the open source community inherited from InterBase 6.0. Firebird 1.5 and all subsequent versions were completely rewritten in C++, with a high degree of standardization across compilers.



At the beginning of Chapter 5, you can review the various major releases in the topic [Version Lineage](#).

Network Access

A Firebird server running on any platform accepts TCP/IP client attachments from any client platform that can implement the Firebird API.



Clients cannot attach to a Firebird server through any medium of filesystem sharing (NFS shares, Samba client connections, Windows shares or mapped drives, etc.).

A client must attach through an absolute physical path. However, from Firebird 1.5 onward, a database aliasing feature allows applications to “soft-connect” through named aliases, whose absolute paths are configured specifically on each deployed server.

A Firebird server running on a services-capable Windows host can accept attachments from Windows clients through the *Named Pipes* network protocol (sometimes wrongly referred to as “NetBEUI”).

Multi-generational Architecture

Firebird’s model for isolating and controlling the work of multiple users revolves around an architecture that is able to store more than one version of a record in the database concurrently. Multiple generations of a record can exist simultaneously—hence the term “multi-generational.” Each user task holds its own contextual view of database state (see the next section) and, when ready, writes its own versions of records on the server’s disk. At that point, the new version (or a deletion) cannot be accessed by other user tasks.

The most recently committed record version is the only one visible outside the user task that successfully posted a new version, and it continues to be the version seen by other tasks. Others will be aware that something has happened to the record because they will be blocked from updating or deleting the same record until the new version becomes “official” by being committed.

Because of its multi-generational architecture (known as MGA), Firebird has no need for the two-phase locking used by other DBMSs to control multi-user concurrency.

Transactions

All user tasks in Firebird are enclosed within transactions. A task begins with a START TRANSACTION statement and ends when work posted by the user task is committed or rolled back. A user task can make multiple requests for operations to be performed within a single transaction, including operations in more than one database.

Work is saved to the database in two stages. In the first stage, changes are stored to disk, without changing database state. In the second stage, changes are committed or rolled back by the client process. Clients can “unpost” parts of work inside an uncommitted transaction by tagging stages in a process as *savepoints* and rolling back to a savepoint without rolling back an entire transaction.

Firebird transactions are atomic, which means that all work posted in a transaction will succeed or all will fail.²

Every transaction is configurable, with three levels of isolation and a variety of strategies for fine-tuning concurrency and read-write conditions.

2. From v.2.5 onward, autonomous transactions within procedural (PSQL) modules are supported. Under certain conditions, an autonomous transaction can be configured to “stick”, even if the transaction in which the module is running is rolled back.

Stored Procedures and Triggers

Firebird has a rich language of procedural extensions, PSQL, for writing stored procedures and triggers. It is a structured language with support for FOR looping through sets, conditional branching, exception handling, and event triggering. PSQL code is compiled at creation and stored in binary form.

In the “2” series and higher, blocks of procedural code can be submitted dynamically by a client application, using the EXECUTE BLOCK syntax.

Trigger support is strong, with a Before and After phase for each DML event.

Multiple triggers may be stored for each phase/event and optionally numbered in an execution sequence. Before or After triggers that comprise behaviors for all three DML events, with conditional branching per event, are supported.³

Referential Integrity

Firebird has full support for formal, SQL-standard referential integrity—sometimes known as *declarative* referential integrity—including optional cascading updates and deletes with a number of RI trigger action options.

Database Shadowing

Firebird servers can optionally maintain database shadows. A shadow is a real-time copy of the database with some extra attributes that make it unavailable for reading until it is made available by the server. Shadows can be “kicked in” if required, either manually or automatically. The purpose of shadowing is to enable the database to become available very quickly if a disk crash occurs.

Shadowing is not replication.

Security

At host level, Firebird provides user access to the server by means of user IDs and encrypted passwords. Like any database server, it relies on adequate physical, network access, and filesystem security being in place. Firebird can store encrypted data but, except for password encryption, it provides no capability to encrypt data itself.



While an embedded server application has its uses for single-user, stand-alone applications, on Windows it bypasses the host's security gateway altogether. SQL privileges defined at database level still apply, but an embedded application can get password-free access to any database on its host machine.

SQL Privileges

Although a user must be authorized to access a Firebird server, no user except the SYSDBA and the database owner has automatic rights to anything within an individual database. Database-level security is supported on an “opt-in” basis, by means of SQL privileges. Users must be granted privileges to any object explicitly.

3. Not in Firebird 1.0.x.

SQL Roles allow sets of privileges to be aggregated and granted as a “package” to individual users. A single user may have privileges under several roles, although only one may be selected when logging into the database.

Operating Modes

Firebird server can be installed to run in one of four modes: Superserver, Classic server, Superclassic or Embedded Server. The distinction is largely a question of architecture. Any client application written to attach to the Superserver can attach in exactly the same way to the Classic or Superclassic⁴ server and perform exactly the same tasks. The reverse is also true, except that Superserver has more exacting thread-safety requirements for external function modules (user-defined functions, character set libraries, and BLOB filters).

The Embedded Server is a variant of Superserver.

Classic and Superclassic Servers

The Classic server⁵ preceded Superserver historically. It was designed late in the 1980s, when machine resources on servers were scarce and programs used them with great economy. The Classic server model continued for operating systems whose threading capabilities were non-existent or too limited to support the Superserver. Classic server remains the best option for environments where high performance is important and system resources are adequate to accommodate linear growth of usage as each new connection is added.

When a client connects to a Classic server, the server spawns a dedicated process for that attachment. Superclassic, introduced in Firebird 2.5, is like Classic except that it spawns threads for attachments.



Classic or Superclassic are the recommended options where utilisation of multiple CPUs is desirable.

Superserver

In 1996, Firebird’s ancestor, InterBase 4.1, introduced the multi-threaded Superserver for the then-new Windows 32-bit platforms. It promised to make better use of the advancing capabilities of servers and networks. Superserver’s abilities to spin off process threads and to allocate cache memory dynamically make it more capable than Classic server where the number of interactive read/write users is high and system resources are limited.

With the explosion of the GNU/Linux operating system on Intel toward the end of the 1990s, a Superserver became a realizable goal for some POSIX platforms.

A Superserver skeleton for Linux was released with the InterBase 6.0 beta open source code and was fully realized in Firebird.



Superserver does not play well with multiple CPUs, especially on Windows, where CPU affinity must be set to a single CPU.

4. Superclassic is not available prior to v.2.5.
5. Not available on Windows in Firebird 1.0.x.

Embedded Server

The embedded variant is a fully functional Firebird rserver compiled with an embedded client that connects directly and exclusively to its database. This single dynamic library (`libfbembedded.so` on POSIX, `fbembedded.dll` on Windows) uses inter-process communication space to transport client requests and server responses. The API is identical to that presented by a regular Firebird client. Nothing special is required in application code to implement an Embedded Server application.

Embedded Server caters to the extreme “low end” of scalability requirements for a Firebird server, enabling a stand-alone, fast-performing, single-machine database application to be packaged and deployed with a small footprint. Since the database can be accessed by a regular server performing a replication service when the embedded application is offline, Embedded Server is particularly suitable for “briefcase” deployment—for example, on a notebook computer, or even on a flash disk.

The Sample Database

Throughout this guide, the language examples use the sample database that can be found in the `examples` directory beneath the Firebird installation root. In all versions except v.1.0.x its name is `employee.fdb`. In Firebird 1.0.x, it is `employee.gdb`. It can be found in the `examples/empbuild` sub-directory of a standard Firebird installation.

Document Conventions

General body text is in this font.

Passages in this font are code, scripts, or command-line examples.



Passages highlighted like this are to draw your attention to something important that might affect your decision to use the feature under discussion.



Passages highlighted like this contain tips, bright ideas, or special recommendations.



Pay special attention to passages like this.



Passages like this provide a bit more information or tell you where to find it.



Passages like this signal something you need to remember.

Syntax Patterns

Some code passages present syntax patterns—that is, code models that demonstrate the optional and mandatory elements of statement and command-line syntax. Certain symbol

conventions apply to syntax patterns. To illustrate the conventions, the following extract shows a syntax pattern for the SQL SELECT statement:

```
SELECT
    [FIRST (m)] [SKIP (n)] [[ALL] | DISTINCT]
    <list of columns> [, [column-name] | expression | constant ] AS alias-name]
FROM <table-or-procedure-or-view>
    [{[[INNER] | [{LEFT | RIGHT | FULL} [OUTER]] JOIN}] <table-or-procedure-or-view>
ON <join-conditions> [{JOIN..}]
    [WHERE <search-conditions>]
    [GROUP BY <grouped-column-list>]
    [HAVING <search-condition>]
    [UNION <select-expression> [ALL]]
    [PLAN <plan-expression>]
    [ORDER BY <column-list>]
    [FOR UPDATE [OF col1 [,col2..]] [WITH LOCK]]
```

Special Symbols

Elements (keywords, parameters) that are mandatory in all cases appear without any markings. In the preceding example, the keywords SELECT and FROM are mandatory for every SELECT statement.

Certain characters that never occur in SQL statements or command-line commands are used in syntax patterns to indicate specific rules about usage. These symbols are [], { }, | , <string>, and None of them is valid in the actual command syntax. They are used in the patterns, as follows:

Square brackets [] indicate that the element(s) within the brackets are optional. When square brackets are nested, it means that the nesting, or the nested element, is optional.

Curly braces { } indicate that the elements within the braces are mandatory. The usual usage of curly braces is seen within an optional (square-bracketed) element, meaning “If the optional element is used, the curly-braced portion is mandatory.” In the preceding example, if the optional explicit JOIN clause is used

```
[[{[[INNER] | [{LEFT | RIGHT | FULL} [OUTER]] JOIN}]
```

the outer pair of curly braces indicates that the keyword JOIN is mandatory. The inner pair of curly braces indicates that, if an OUTER join is specified, it must be qualified as either LEFT, RIGHT, or FULL, with optional use of the keyword OUTER.

The *pipe symbol* | is used to separate mutually exclusive elements. In the preceding example, LEFT, RIGHT, and FULL are mutually exclusive, and inner join and any outer join are mutually exclusive. The first option of a set of choices is normally the default.

Parameters are indicated with a string representing the parameter, enclosed angle brackets <>. For example, [WHERE <search-conditions>] indicates that one or more search conditions are required as parameters to the optional WHERE clause in the SELECT syntax.

In some cases, the <string> convention may be a shorthand for more complex options, that subsequent lines in the syntax pattern would “explode,” level by level, to provide finer details. For example, you might see an expansion line like this:

```
<search-conditions> := <column-expression> := <constant> | <expression>
```

Pairs or triplets of dots ... may be used in some syntax patterns to indicate that the current element is repeatable.

Contents

Author's Foreword	iii
About the Author	v
Introduction	vi
What Is Firebird?	vi
Who Needs This Book?	vi
Where to Find What You Need	vi
Appendices and Glossary	vii
Firebird's Origins	viii
The Firebird Project	viii
Overview of Firebird	ix
Firebird Versions	ix
Network Access	ix
Multi-generational Architecture	x
Transactions	x
Stored Procedures and Triggers	xi
Referential Integrity	xi
Database Shadowing	xi
Security	xi
Operating Modes	xii
The Sample Database	xiii
Document Conventions	xiii
Syntax Patterns	xiii

Part I Firing Up with Firebird..... 1

1 Firebird Servers and Clients 3

The Role of the Server	3
Operating System Platforms	4
Databases	4
Server-side Programming.....	4
Multi-database Applications.....	6
Server Security	7
Database Security	7
The Firebird Server Models	7
Resource Usage	8
Comparing Superserver, Superclassic and Classic Architectures	9
Embedded Server.....	13
Introduction to Client/Server	13
Servers and Clients.....	13
Client/Server vs File-Served Databases	14
Characteristics of a Client/Server DBMS.....	14
Typical Deployment Topologies	17
Firebird Clients.....	20
What is a Firebird Client?.....	21
The Firebird Client Libraries	22

2 Installation..... 23

System Requirements.....	23
Server Memory (All Platforms)	23
Installation Drives	24
Minimum Machine Specifications.....	24
Operating System.....	25
How to Get an Installation Kit	27
Kit Contents.....	27
Installing a Server.....	29
The FIREBIRD Variable.....	29
Finalise your Server Choice	29
Windows.....	30
Linux and Many Other POSIX.....	33

MacOSX/Darwin.	36
Other Host Platforms	37
Testing Your Installation	38
Network Protocol.	38
Checking That the Firebird Server Is Running.	38
The Client Libraries	41
Performing a Client-Only Install.	41
Installing an Embedded Server	45
Uninstalling Firebird.	46
Linux.	46
Windows.	47
MacOSX/Darwin.	48
Other Things You Need to Know	48
Default User Name and Password	48
3 Network Setup and Initial Configuration.	51
Network Protocols.	51
TCP/IP.	51
Named Pipes	52
Local Access.	52
Mixed Platforms.	53
A Network Address for the Server	54
Hosts File	54
Server Name and Database Path.	55
Connection string syntax	56
Inconsistent connection strings for Windows connections.	57
Testing Connections.	57
If ping fails	58
Initial Configuration.	60
The Firebird Configuration File	60
Environment Variables	62
4 Operating Basics.	65
Looking for a User Interface	65
Running Firebird on POSIX	65

Superserver and Superclassic	66
Classic server	68
Running Firebird on Windows	69
Servers and the Guardian.	69
Running as a service.	70
Running as an application	71
Running Firebird on MacOSX	72
The Super* Models	72
Classic	72
Mixed Platforms	73
Database aliasing	73
The SYSDBA User and Password.	75
Linux	76
Administering Databases	76
The employee.fdb Database	76
Starting isql	76
Summary of Command-line Tools	79
5 Migration Notes	81
Version Lineage	81
<i>Firebird 1.0</i>	81
<i>Firebird 1.5</i>	82
<i>Firebird 2.0</i>	82
<i>Firebird 2.1</i>	83
<i>Firebird 2.5</i>	83
Backward Compatibility.	84
Preparing to Migrate.	85
Major Releases	85
Sub-releases	86
Dialect 1 Databases	87
Back Up!	89
Migration Tasks	90
Get the Latest Sub-release!	90
The FIREBIRD Variable.	90
Platform Issues.	91
Security Database	91
Metadata Repair	92

Application Issues	95
Migration Tools	97
Part II Firebird Data Types & Domains.....	99
6 About Firebird Data Types.....	101
The Basics.....	101
Where to Specify Data Types	101
Supported Data Types	101
SQL “Dialects”.....	102
Context Variables.....	103
Pre-defined Date Literals.....	104
Optional SQL-92 Delimited Identifiers	105
Columns	105
Domains	106
Converting Data Types	106
Changing column and domain definitions	107
Keywords Used for Specifying Data Type	108
Demystifying NULL	109
NULL in expressions.....	109
NULL in calculations	110
Gotchas with True and False.....	110
Setting a value to NULL	111
7 Number Types.....	113
Numerical Limits	113
Points About Points.....	114
Operations on Number Types.....	114
Integer Types	114
Notation	114
Types.....	115
Integer/integer division.....	116
Fixed Decimal (Scaled) Types	117
Internal Storage	118
Special restrictions in static SQL.....	119
Behaviour of fixed types in operations	119
Floating Point Types	122

Supported Float Types	123
Arithmetic mixing fixed and floating-point types	123
8 Date and Time Types	125
Choices for Date and Time Values	125
DATE	125
TIMESTAMP	126
TIME	126
Date/Time Literals	127
Recognised Date/Time Literal Formats	127
Pre-defined Date Literals	130
Type-casting of Date/Time Literals	131
Date and Time Context Variables	132
Specifying Sub-seconds Precision	132
Operations Using Date and Time Values	133
General rules for operations	134
Using CAST() with Date/Time Types	135
Quick Date/Time Casts	138
The EXTRACT() Function	138
Other Date/Time Functions	140
A sample date/time type conversion task	140
9 Character Types	143
String Essentials	143
String Delimiter	144
Concatenation	144
Escape Characters	144
String Functions	145
Limitations with Character Types	146
Fixed-length Character Data	147
Variable-length Character Data	147
Character Sets and Collation Sequences	148
Character Sets	149
Client Character Set	150
Firebird character sets	151
Special Character Sets	154
Transliteration	156

Collation Sequence	157
Adding an Alias for a Character Set	161
Custom character sets and collations	162
Metadata Text Conversion.	167
10 BLOBS and Arrays	169
BLOBs and Subtypes	169
Supported User Subtypes.	170
Custom Subtypes	170
Declaration Syntax	171
BLOB Segments.	171
When to Use BLOB Types	172
Operations on BLOBs.	173
Array Types.	175
ARRAY Types and SQL	175
When to Use an Array Type	176
Eligible Element Types	176
Defining Arrays	176
Storage of ARRAY Columns.	177
Accessing Array Data.	178
Limited DSQL Access.	178
11 Domains.	181
Benefits of Using Domains	181
Creating a Domain	182
Using a Domain	186
Domains in Column Definitions.	186
Domains in PSQL Variable Declarations	188
Using Domains with CAST().	188
Where domains won't work.	189
Defining a BOOLEAN Domain.	189
Changing a Domain Definition	189
Dropping a Domain.	191

Part III A Database & Its Objects. 193

12 Designing and Defining a Database 195

Designing a Database.	195
Description and analysis	196
The Physical Objects	197
Referential integrity	201
Indexes and query plans.	202
Views.	202
Stored procedures and triggers	202
Generators (Sequences)	203
Object Naming Conventions.	205
Optional SQL-92 delimited identifiers	205
File-naming Conventions for Databases.	206
Metadata	206
The system tables	206
Firebird's SQL Language.	207
SQL "Dialects".	208
Firebird and the ISO Standards.	208
Data Definition Language (DDL)	208
Data Manipulation Language (DML)	209
Procedural language (PSQL)	209
Interactive SQL (ISQL)	210
Schemas and Scripts	210
Using isql to run scripts	210
Resources	210

13 Data Definition Language—DDL 211

SQL Data Definition Statements.	211
CREATE	211
RECREATE.	212
ALTER	212
DECLARE.	213
DROP.	213
Storing Descriptive Text	213
Object Dependencies.	215

Using DDL to Manage User Accounts	215
Reference Material	217
14 Creating and Maintaining a Database	219
Physical Storage for a Database	219
About Security Access	220
ISC_USER and ISC_PASSWORD	221
Creating a Database	221
Dialect	221
CREATE DATABASE Statement	222
Getting information about the database	224
Single and Multi-file Databases	225
The Database Cache	227
Read-only Databases	231
Keeping a Clean Database	232
Background garbage collection	233
Sweeping	233
Garbage collection during backup	233
Objects and counters	234
Validation and repair	234
Backup and Stand-by	236
Dropping a Database	236
15 Tables	237
About Firebird Physical Tables	237
Structural descriptions	237
Creating Tables	238
Table ownership and privileges	238
CREATE TABLE statements	238
Constraints	243
Scope of Constraints	243
Integrity Constraints	243
The referential constraint	243
Named constraints	244
The NOT NULL Constraint	244

The PRIMARY KEY constraint	245
The UNIQUE constraint	250
CHECK constraints	251
Using External Files as Tables	253
Restrictions and recommendations	254
Operations	255
Converting external tables to internal tables	257
Dropped database	258
Altering Tables	258
Preparing to use ALTER TABLE	258
Altering Columns in a Table	258
Removing (Dropping) a Table	263
The RECREATE TABLE statement	263
Temporary Tables	263
Global Temporary Tables	264
Temporary Storage for Older Versions	265
Tree Structures	266

16 Indexes 267

Limits	267
Automatic vs User-defined Indexes	268
Importing legacy indexes	268
Directional indexes	268
Query plans	269
How Indexes Can Help	269
Sorting and grouping	269
Joins	270
Comparisons	270
What to Index	270
When to index	271
Using CREATE INDEX	271
Mandatory elements	271
Optional elements	272
Multi-column Indexes	274
OR predicates in queries	274

Search criteria	275
Inspecting Indexes	275
Making an Index Inactive.	275
Housekeeping.	276
‘Index is in use’ error	276
Altering the structure of an index	276
Dropping an Index.	276
17 Referential Integrity	279
Terminology	279
The FOREIGN KEY Constraint	280
Implementing the constraint	280
Action triggers to vary integrity rules	282
Interaction of Constraints	283
Custom Action Triggers.	284
Lookup Tables and Your Data Model	284
The REFERENCES Privilege.	286
Handling Other Forms of Relationship	286
One-to-one relationship.	287
Many-to-many relationship	287
Self-Referencing Relationship	290
Mandatory relationships.	291
“Object is in Use” Error	291
Part IV Working With Data.	293
18 Data Manipulation Language—DML	295
The Concept of Sets.	295
Cardinality and degree	296
A table is a set.	296
Output sets	296
Input sets	297
Output sets as input sets	298
Cursor sets	298
Nested Sets	299

SQL Privileges and DML	300
19 DML Queries	301
The SELECT Statement	301
Clauses in a SELECT statement	302
INSERT and UPDATE Statements	311
INSERT INTO	311
UPDATE	315
UPDATE OR INSERT	319
MERGE	320
The DELETE Statement	322
EXECUTE Statements	322
Using Parameters	323
Batch operations	324
Queries That Execute Server-side Code	325
Executable procedures	325
Selectable procedures	326
DML operations and state-changing events	326
Referential integrity action clauses	327
Custom triggers	327
Query Plans and the Optimizer	328
Plans and the Firebird query optimizer	328
Understanding the optimizer	330
Examples of plans	332
Specifying your own plan	337
Optimal Indexing	338
Housekeeping indexes	340
Index toolkit	342
20 Expressions and Predicates	345
Expressions	345
Predicates	346
The Truth Testers	347
Assertions	347
Deciding What is True	348
Elements Used in Expressions	348

SQL Operators	350
Precedence of Operators	350
Concatenation Operator	351
Arithmetic Operators	351
Comparison Operators	351
Logical Operators	356
The IS [NOT] NULL Predicate	357
Existential Predicates	357
Considering NULL	360
NULL in Expressions	360
NULL in Calculations	361
Gotchas with True and False	361
NULL and External Functions	362
Setting a Value to NULL	362
Using Expressions	363
Computed Columns	363
Search Conditions	369
Ordering and Grouping Conditions	371
Expression Indexes	372
CHECK expressions in DDL	372
Expressions in PSQL	373
Function Calls	373
Conversion Functions	374
String Functions	376
Function for Getting a Generator (Sequence) Value	377
Aggregating Functions	378
Functions for Setting and Getting Contextual Data	379
External Functions (UDFs)	382
21 Querying Multiple Tables	387
Kinds of Multi-table Queries	387
Joining	388
Subqueries	388
UNION queries	388
Using Relation Aliases	388
The internal cursor	389
Joining	390

The INNER join	390
OUTER joins	393
Equi-joins	396
Re-entrant joins	397
Subqueries.....	399
Specifying a column using a subquery.....	399
Searching using a subquery	401
Inserting using a subquery with joins	401
The Derived Table.....	402
UNION Queries	402
Union compatible sets	402
UNION ALL DISTINCT.....	403
Using run-time columns in unions	404
Search and ordering conditions.....	405
Re-entrant UNION queries	405
Recursive Queries.....	406
22 Ordered and Aggregated Sets	407
Considerations for Sorting.....	407
Presentation order of sorting clauses	407
Indexing	409
The ORDER BY Clause	409
Sorting items.....	409
Sort direction	412
NULLS placement	413
The GROUP BY Clause	413
The groupable field list	413
Aggregating expressions	414
The grouping item	415
The HAVING sub-clause	417
The COLLATE sub-clause	418
Using ORDER BY in a grouped query	418
Advanced grouping conditions	418
Aggregating Functions.....	421
AVG()	421
SUM()	421
MAX()	422

MIN().....	422
Using COUNT() as an aggregating function	422
The LIST() function	423
23 Views and Other Run-time Set Objects.....	425
Forms and Functions.....	425
Views.....	426
What is a view?	426
Creating a View	427
How views can be useful	431
Some simple view specifications	431
Read-only and updatable views	432
Modifying a view	435
Dropping a view	436
Privileges for Views	436
Using views in SQL	437
Using query plans for views.....	438
Derived Tables	439
Rules for Derived Tables.....	440
When to Use a Derived Table.....	441
Common Table Expressions	441
Syntax for a CTE	442
Other Virtual Set Objects	446
Global temporary tables (GTTs).....	446
Selectable stored procedures	446
External virtual tables.....	447
24 Interactive SQL Utility (isql).....	449
Interactive Mode	449
Default text editor	449
Starting isql.....	450
Connecting to a database.....	451
Using the Interface.....	452
Interactive Commands.....	456
General <i>isql</i> Commands.....	457
SHOW Commands	460
SET Commands.....	469

Exiting an interactive <i>isql</i> session	476
Command Mode <i>isql</i>	476
Operating <i>isql</i> in command mode	476
Command-line switches	477
Creating and Running Scripts	480
About Firebird scripts	480
Basic steps	483
Managing your schema scripts	486

Part V Transactions 489

25 Overview of Firebird Transactions 491

The ACID Properties	492
Atomicity	492
Consistency	492
Isolation	492
Durability	493
Context of a Transaction	493
One transaction, many requests	493
Transactions and the MGA	494
Post vs COMMIT	494
Rollback	495
Row locking	495
Table-level locks	496
Inserts	497
Transaction “Aging” and Statistics	497
Transaction ID and age	497
“Interesting transactions”	498
Background garbage collection	499
Keeping the OIT and the OAT moving	500
Transaction statistics	502

26 Configuring Transactions 505

Anatomy of a Transaction	505
The Default Transaction	505
Default settings	506
About Concurrency	506

Factors affecting concurrency	507
Isolation level	507
Standard levels of isolation	507
Locking Policy	509
Access Mode	510
Table Reservation	511
Uses for Table Reservation	511
Parameters for table reservation	512
Other Optional Parameters	513
Record Versions	513
Dependent rows	514
Locking and Lock Conflicts	514
Timing	514
Pessimistic locking	514
Lock conflicts	515
27 Programming with Transactions	519
The Language of Transactions	519
The API	520
Starting a Transaction	520
The transaction handle	520
The transaction parameter buffer (TPB)	521
Accessing the Transaction ID	522
Using the TID in applications	522
Progress of a Transaction	523
Nested Transactions	523
User savepoints	524
Exception handling extensions in PSQL	525
The Logical Context	525
Ending transactions	526
Diagnosing exceptions	528
Receiving exceptions	529
Multi-database Transactions	530
Limbo transactions	531
Restricting databases	531

Pessimistic Locking	531
Table-level locking	532
Statement-level locking	532
The “dummy update” hack	533
Explicit locking	534
Stored Procedures, Triggers and Transactions	536
Stored Procedures	536
Triggers	537
“Savepoints” in PSQL	537
Autonomous Transactions	537
Tips for Optimizing Transaction Behaviour	537

Part VI Programming on the Server 539

28 Procedural SQL—PSQL 541

Overview of Server Code Modules	541
About Stored Procedures	542
About Relation Triggers	543
About Database Triggers	544
About Executable Blocks	544
PSQL Language Extensions	544
Restrictions on PSQL	545
Exceptions	546
Events	546
Security	546
Elements of procedures and triggers	547
Statement terminator	547
The CREATE statement	548
Header elements	549
Body elements	549
Language elements	550
Programming constructs	552
BEGIN...END blocks	552
Conditional blocks	552
Variables and Parameters	555
SELECT...INTO statements	562

Flow of control statements	564
Execute Statement	567
Using Cursors	571
The Undeclared Named Cursor	572
The Explicit Named Cursor	573
Developing Modules	576
Adding comments	577
Case-sensitivity, white space and size	577
Managing your code	578
Compiling stored procedures and triggers	579
Altering and dropping modules	579
Deleting source from modules	580
Internals of the Technology	581
Effects of changes to modules	581
29 Stored Procedures and Executable Blocks	583
Styles of Stored Procedure	583
Creating a Stored Procedure	584
Header elements	584
Body elements	585
Executable Procedures	587
Complex processing	588
Support for “live” client sets	588
Operations in executable procedures	588
A multi-table procedure	589
Using (calling) executable procedures	591
Outputs and exits	592
Recursive Procedures	592
Selectable Stored Procedures	594
Uses for selectable procedures	594
The Technique	595
A simple procedure with nested SELECTs	597
Calling a selectable procedure	598
Nested procedures	599
A procedure with running totals	603
Viewing an array through a stored procedure	605
Testing procedures	606

Procedures for combined use	607
Using the Internal RDB\$DB_KEY	607
About RDB\$DB_KEY	607
Benefits	608
Inserting	609
Duration of validity	610
RDB\$DB_KEY with multi-table sets	611
Changing a Stored Procedure	611
Effect on applications	612
Syntax for changing procedures	612
Dropping a Stored Procedure	614
Restrictions	615
Run-time PSQL	615
Coding EXECUTE BLOCK	616
Using EXECUTE BLOCK in applications	618
30 Triggers	619
Classes of Trigger	619
Table-level Triggers	619
“Database Triggers”	620
DDL Triggers	620
About Table-level Triggers	620
Phase and event	620
Sequence	621
Status Active/Inactive	622
Creating Table-level Triggers	622
An alternative syntax	623
Header elements	623
The trigger body	624
Special PSQL for Table-level Triggers	624
Table-level triggers at work	625
Updating Other Tables	628
Referential Integrity Support	631
Updating rows in the same table	634
Implementing Auto-Incrementing Keys	635
Changing Triggers	637

ALTER TRIGGER	638
CREATE OR ALTER TRIGGER	639
RECREATE TRIGGER	639
Higher-level Triggers	639
Trigger Events	639
Creating and Changing the Higher-level Triggers	641
Dropping Triggers	641
31 Accessing Other Databases from PSQL	643
Extensions to EXECUTE STATEMENT	643
Autonomous Transaction	643
The Optional Extension Clauses	644
Transaction Behaviour	645
External Queries	646
Authentication	647
Exceptions	647
32 Error Handling and Events	649
Exceptions in PSQL	649
Types of Exceptions	649
What is an exception?	650
Exceptions in Action	650
Trapping and Handling Exceptions	652
The WHEN statement	652
Nested exceptions as savepoints	654
Exceptions in triggers	657
Run-time Exception Messaging	658
Error Codes Listings	659
Events	659
Uses for Events Notification	659
Elements of the Events Mechanism	660
Synchronous listening	661
Asynchronous signalling	662
Using POST_EVENT	663

Part VII Configuring Firebird..... 665

33 Configuring Firebird and Its Environment 667

Default Configuration	667
Finding the Firebird Root Directory.....	667
Environment Variables	668
Windows.....	668
POSIX	669
The Variables	670
The Firebird Configuration File	672
Parameter Syntax	673
Editing Parameters.....	674
Version Differences	674
Parameters in Detail.....	675
Configuring the TCP/IP Port Service.....	675
How the server sets the listening port.....	675
Setting up a client to find the service port	677
Configuring the services file	679
Embedded Server.....	679

34 Configuration Parameters in Detail..... 681

Settings for All Platforms and Servers.....	681
<i>AuditTraceConfigFile</i>	681
<i>Authentication</i>	682
<i>CompleteBooleanEvaluation</i>	682
<i>ConnectionTimeout</i>	683
<i>DatabaseAccess</i>	683
<i>DatabaseGrowthIncrement</i>	684
<i>DeadlockTimeout</i>	684
<i>DefaultDbCachePages</i>	684
<i>DummyPacketInterval</i>	684
<i>FileSystemCacheThreshold</i>	685
<i>FileSystemCacheSize</i>	685
<i>GCPolicy</i>	686
<i>LegacyHash</i>	686
<i>LockAcquireSpins</i>	686
<i>LockHashSlots</i>	686
<i>LockGrantOrder</i>	687
<i>LockMemSize</i>	687

<i>LockSemCount</i>	688
<i>LockSignal</i>	688
<i>MaxUserTraceLogSize</i>	688
<i>OldColumnNaming</i>	689
<i>OldSetClauseSemantics</i>	689
<i>Redirection</i>	689
<i>RelaxedAliasChecking</i>	689
<i>RemoteAuxPort</i>	690
<i>RemoteBindAddress</i>	690
<i>RemoteFileOpenAbility</i>	690
<i>RemoteServicePort</i>	691
<i>RemoteServiceName</i>	691
<i>RootDirectory</i>	691
<i>TcpRemoteBufferSize</i>	691
<i>TcpNoNagle</i>	691
<i>TempBlockSize</i>	692
<i>TempCacheLimit</i>	692
<i>TempDirectories</i>	692
Settings Applicable to Microsoft Windows.	693
<i>CpuAffinityMask</i>	693
<i>GuardianOption</i>	694
<i>IpcName</i>	694
<i>MaxUnflushedWrites</i>	694
<i>MaxUnflushedWriteTime</i>	695
<i>ProcessPriorityLevel</i>	695
<i>RemotePipeName</i>	695
<i>UsePriorityScheduler</i>	695
Settings Applicable to POSIX Platforms	696
<i>BugCheckAbort</i>	696
Configuring external locations.	696
<i>UdfAccess</i>	696
<i>ExternalFileAccess</i>	697
Deprecated Settings	698
<i>OldParameterOrdering</i>	698
<i>server_working_size_max</i>	698
<i>server_working_size_min</i>	698
<i>CreateInternalWindow</i>	698
<i>DeadThreadsCollection</i>	699

35 Configuring and Managing Databases 701

The <i>gfx</i> Tool Set.	701
Using the Tools	702

Shutting Down a Database	702
Configuration Options	705
Default Cache Size	705
Forced Writes	706
Access Mode	707
Page Fill Capacity	708
Sweep Interval	708
SQL Dialect	709
Page Size	710
Management Tools	711
Garbage collection	711
Sweeping	711
Analysing and Repairing Logical Corruption	712
Transaction recovery	713
Managing Database Shadows	715
Summary of <i>gfx</i> switches and options	716
<i>gfx</i> error messages	718

Part VIII Administering & Securing Firebird 719

36 Protecting the Server and its Environment 721

Securing the Environment	721
Physical Security	721
Use Securable Filesystems	722
Protect Backups	722
Platform-based protection	722
Wire Security	726
Web and Other n-Tier Applications	726
Use Dedicated Hosts	726
Establish Trustworthiness	726
Firewalls	727
Server Multi-hop	727
Denial-of-Service Attacks	727
Managing User Access	728
The Security Database	728
Firebird “Native” Users	729
Platform Users	729

Privileged Users	731
Entering User Credentials via SQL.....	732
The <i>gsec</i> Utility.....	734
37 Database-Level Security.....	741
Default Security and Access.....	741
Planning an Access Scheme.....	742
Metadata Tables	742
SQL Privileges	742
Objects	743
Users	743
Native Firebird users	744
POSIX users and groups	744
Windows trusted users.....	744
Users with escalated privileges.....	745
Database objects as ‘users’.....	745
Privilege Restrictions	745
Granting Privileges	746
UPDATE rights on columns.....	746
REFERENCES rights on columns	747
Privileges needed by objects	748
Granting the EXECUTE privilege	748
Privileges on Views	749
Bundling Multiple Privileges	749
Lists of privileges	749
The ALL privilege	750
Roles	750
Privileges for Multiple Users	751
Granting the Right to Grant Privileges	753
Granting Privileges on Behalf of Another	753
The GRANTED BY clause.....	754
The System Role RDB\$ADMIN.....	754
Extending RDB\$ADMIN power for Windows Administrators	754
Unintended Effects with Privileges.....	755
Revoking Privileges	756

Using REVOKE	756
Security Scripts	759
Creating a script	760
Installing perms directly from a procedure	762
A Trick to Beat Idiot Users and Bad Guys	764
38 Monitoring and Logging Features	765
Monitoring Database Activity	765
How Monitoring Works	765
Using MON\$	766
The MON\$ Tables	769
Trace and Audit Services	769
Modes of Use	769
Trace Sessions	770
Trace Plug-in Facilities	780
Collecting Database Statistics— <i>gstat</i>	780
<i>gstat</i> command-line tool	780
Monitoring Locks	791
Locking in Firebird	791
The Lock Manager Module	792
The Lock Print Utility	794
Lock configuration settings	814
39 Backing Up Databases	817
<i>gbak</i> or <i>nBackup</i> ?	817
The <i>gbak</i> Utility	818
About <i>gbak</i> Backup Files	818
<i>gbak</i> 's other talents	819
Database backup & restore rights	820
Running a Backup	821
Running a Restore	825
Restore switches	827
Using <i>gbak</i> with the Firebird Services Manager	831
<i>gbak</i> Error Messages	833
Incremental Backup Tool (<i>nBackup</i>)	837
About <i>nBackup</i>	837
Making Backups	838

Restoring from <i>nBackup</i> Files	842
Using the Freeze Utility	843
SQL Support for nBackup.	845
nBackup command-line options summary	846
Database Shadowing	847
Benefits and limitations of shadowing	847
Implementing shadowing.	848
Replication	852
“Warm Backups”	852
40 The Services Manager	853
About the Services Manager	853
Accessing the Services Manager API	853
Services Manager Clients	854
The <i>fbsvcmgr</i> Utility	854
Using <i>fbsvcmgr</i>	854
fbsvcmgr specifics	856
Services API Functions	857
Part IX Appendices	859
I Internal and External Functions	861
Internal Functions	861
Conditional Logic Functions	861
Date and Time Functions	862
String and Character Functions.	864
BLOB Functions	869
Mathematical Functions.	870
Trigonometrical Functions	872
Binary Functions	873
Miscellaneous Functions	874
External Functions.	875
Conditional Logic Functions	875
Mathematical Functions.	876
Date and Time Functions	881
String and Character Functions.	884
BLOB Functions	889
Trigonometrical Functions	890

Building regular expressions	894
Characters	894
Quantifiers	896
OR-ing terms	897
Sub-expressions	897
Escaping special characters	897
II Reserved and Non-Reserved Keywords	899
Keywords	899
III Context Variables	911
About Context Variables	911
IV Firebird Limits	913
V System Tables and Views	917
Metadata Tables	917
System Views	940
Monitoring Tables	942
VI Character Sets and Collations	951
Implemented and Activated Character Sets	951
VII Firebird Error Codes	957
VIII SQLSTATE Codes	979
IX Database Repair How-To	987
Indications of Possible Corruption	987
Preparing for Analysis and Repair	988
Steps for Recovery Using Command-line Tools	989
Step 1	989
Step 2	990
Step 3	990
Step 4	990
Step 5	991
Step 6	991
Failed Repair	991
X Default Disk Locations	993
XI Healthcare for Databases	1001
Considerations for a Maintenance Regime	1001
1 Backups	1001
2 Garbage Collection	1002
3 Index Statistics	1004

4 Page Size	1008
5 Disk Usage.....	1008
Other Considerations.....	1009
Usage Monitoring.....	1009
Data Access and Security.....	1009
XII Upgrade Scripts.....	1011
Security Database.....	1011
Metadata Repair	1014
XIII Application Interfaces	1019
Application Development	1019
Dynamic client/server applications	1019
The Firebird Core API	1020
Application interfaces using the API.....	1020
Designing Databases for Client/Server Systems	1022

Index i

XIV Resources.....	xxvii
Free Documentation	xxvii
Books	xxvii
Free help.....	xxviii
SQL and Firebird server support.....	xxviii
Client interface support	xxviii
Third-Party Tools.....	xxix
Commercial Help and Support	xxxi

GLOSSARY xxxiii

SECOND EDITION

PART I



Firing Up with Firebird

CHAPTER

1

FIREBIRD SERVERS AND CLIENTS

The Firebird server is a program that runs on a host node in a network and listens for clients on a communication port. It serves requests from multiple clients to multiple databases simultaneously.

SuperServer is a multi-threaded process that starts a new thread for each attached client. In Classic, a new process is started for each connection. Firebird 2.5 introduced Superclassic, a modification of the Classic model that launches threads to attach clients from a single parent process.

Firebird servers can run on a wide spectrum of hardware and accept client connections from applications running on incompatible systems. Small-footprint server installations can be done on outdated equipment, even old boxes running 32-bit Windows XP or a minimal Linux. At the other end of the scale, Firebird servers are running in distributed environments, managing databases in the Terabyte range.

Of course, it is not realistic to plan an enterprise information system to run on a 32-bit Windows box. However, it is a simple matter to start with a minimally configured server and scale both vertically and horizontally as need arises.

The Role of the Server

The server's jobs include

- managing database storage and disk space allocation
- regulating all transactions started by clients, ensuring that each gets and keeps a consistent view of the permanently stored data that it has requested through the client
- managing commits, data and housekeeping
- maintaining locks and statistics for each database
- handling requests to insert, modify or delete rows and maintain the currency and obsolescence of record versions

- maintaining the metadata for each database and servicing client requests to create new databases and database objects, alter structures, validate and compile stored procedures and triggers
- servicing client requests for result sets and procedure execution
- routing messages to clients
- maintaining cached data to keep frequently-used data sets and indexes in the foreground
- maintaining a separate security database for verifying user access

Operating System Platforms

Firebird server platforms include, but are not limited to

- Linux, FreeBSD and several UNIX operating systems
- Microsoft Windows service-capable platforms NT 4.0 and Windows 2000 (Server or Workstation editions), XP Professional and Server 2003. Windows 9x, ME and XP Home can support servers listening on TCP ports but not Named Pipes protocols such as NetBEUI
- MacOS X (Darwin)
- Sun Solaris Sparc and Intel
- HP-UX

Databases

Each database exists in one or more files, which grow dynamically as need arises. Database files must be stored on disk storage that is under the physical control of the machine on which the server is hosted. Only a server process can perform direct I/O on the database files.

A Firebird database file consists of blocks of storage known as pages. The size of one database page can be 1, 2, 4, 8 or 16 Kb and it is set at database creation time. It can be changed only by restoring a backed-up database and specifying a new size. Different databases on the same server can have different page sizes.

The server maintains a number of different page types in each database—data pages, several levels of index pages, BLOB pages, inventory pages for various accounting purposes, and so on. It lays out pages in a geography known only to itself and a handful of gifted wizards. Unlike file-served database management systems, Firebird does not store tables in physical rows and columns but in a continuous stream, on pages. When a page is nearing full capacity and more rows are to be written, the server requests disk space from the operating system and allocates a new page. Pages from a single table are not stored in any contiguous sequence. In fact, pages belonging to a single table could be distributed across several database files on several disks.

Server-side Programming

Among Firebird's powerful features for dynamic client/server application programming is its capability to compile source code on the server into a binary form for run-time interpretation. Such procedures and functions are executed completely on the server,

optionally returning values or data sets to the client application. Firebird provides two styles of server-side programming capability: stored procedures and triggers. In addition, external functions, also known as “user-defined functions” (UDFs) can be written in a high-level language and made available to the server for use in SQL expressions.

Stored procedures

Firebird’s procedure language (PSQL) implements extensions to its SQL language, providing conditional logic, flow control structures, exception handling (both built-in and user-defined), local variables, an event mechanism and the capability to accept input arguments of almost any type supported by Firebird. It implements a powerful flow control structure for processing cursors which can output a dataset directly to client memory without the need to create temporary tables. Such procedures are called from the client with a SELECT statement and are known to developers as “selectable stored procedures”.

Stored procedures can embed other stored procedures and can be recursive. All stored procedure execution, including selection of data sets from procedures and embedded calls to other procedures, is under the control of the single transaction that calls it. Accordingly, the work of a stored procedure call will be cancelled totally if the client rolls back the transaction.

Autonomous Transactions

In the normal workflow of a stored procedure, it is not possible to start and commit a transaction autonomously from within a stored procedure, as it breaks the consistency model whereby the client application maintains full control over what, when and how database state changes.

However, in v.2.5 the capability was introduced to execute autonomous transactions in a limited way, using the `IN AUTONOMOUS TRANSACTION DO..` construct or an enhanced implementation of `EXECUTE STATEMENT` in Firebird’s procedural language (PSQL) that provides a `WITH AUTONOMOUS TRANSACTION` attribute. The latter route can be used to enable the updating of data in another database.

Triggers

Triggers are special procedures you can create for specific events in a database, for automatic execution at some appointed phase of the event.

Row-level triggers

During the process of posting the insertion, update or deletion of a row to the server, you can write triggers for the table that “fire” before or after that event. A trigger fires for every row in its parent table that is affected by the event. Any table can have any number of triggers to be executed before or after inserts, updates and deletions. Execution order is determined by a position parameter in the trigger’s declaration. Row-level triggers have some language extensions not available to regular stored procedures or to dynamic sql, notably the context variables `OLD` and `NEW` which, when prefixed to a column identifier, provide references to the existing and requested new values of the column.

Triggers can call stored procedures but not other triggers.

Work performed by table-level triggers will be rolled back if the transaction that prompted them is rolled back.

“Database-level” triggers

The v.2.1 release introduced triggers whose scope is broader than the table-level trigger. The term “database-level” is not really appropriate, since these higher-level triggers are written with an explicit duration in view. A “connection-level” trigger takes effect when a client session begins and its effects last until the session ends. A “transaction-level” trigger takes effect when a transaction begins and its effects end when the transaction is committed or rolled back.

Such triggers have several uses, including the ability to restrict connections to specific users and the ability to block specified actions if some criteria are not met. They can be used to set and get context variables, a feature that was also enhanced at v.2.1 to enable user-defined context variables.

User-defined functions

By design, in order to preserve its small footprint, Firebird in many of its older versions comes with a very modest arsenal of internally-defined (native) data transformation functions. Developers can write their own very precise functions in familiar host-language code such as C/C++, Pascal or Object Pascal to accept arguments and return a single result. Once an external function—UDF—is declared to a database, it becomes available as a valid SQL function to applications, stored procedures and triggers.

Firebird supplies two libraries of ready-to-use UDFs: `ib_udf` and `fbudf`, available for both Windows and POSIX.



Many of the functions in these libraries have been superseded in v.2.1 and higher by a substantial collection of internal functions. Where both an internal implementation and an external version of the same function exist, the internal function is preferred, both to reduce the vulnerability of the engine to outside interference and to make use of the greater standards compliance of the internal versions.

UDF functions are adjuncts to the Firebird engine, not to the client libraries. Firebird looks for UDFs in libraries stored in the `/udf` directory of its installation or in other directories configured in `firebird.conf` by the **UdfAccess** parameter.

v.1.0.x *In the v.1.0.x configuration file, the name of the parameter is **external_function_directory**.*

Multi-database Applications

Firebird applications can be connected to more than one database simultaneously, with any number of databases open and accessible to the client application at the same time. Tables from separate databases can not be joined to return linked sets, but cursors can be used to combine information.

If consistency across database boundaries is required, Firebird can manage output sets from querying multiple databases inside a single transaction. Firebird implements automatic two-phase commit when data changes occur, to ensure that changes cannot be committed in one database if changes in another database, within the same transaction context, are rolled back or lost through a network failure.

From v.2.5 forward, external databases can be updated from within a single database connection, using autonomous transactions in procedural SQL.

Server Security

For controlling user access to the server, the Firebird server creates and maintains the security database, a database of users known to the server. For Firebird 2.0 and higher server versions, the name of this database is `security2.fdb`. For v.1.5, it is `security.fdb`, whilst its name in v.1.0.x is `isc4.gdb`. At installation time, this database contains one user: SYSDBA.

- In Windows installations, the SYSDBA password is *masterkey*. It is strongly recommended that you run the program `gsec.exe` (in the installation `/bin` directory) immediately after installing and change this password. This is one of the best-known passwords in the database world!
- From v.1.5 forward, the Linux RPM installers generate a random password for SYSDBA and update the database to replace *masterkey*. This password is stored in the installation root in a text file named `SYSDBA.password`. If you want to keep using the password, delete this file!

The SYSDBA has full privileges to every database on the server and, in the current security model, this can not be changed. The root user on Linux/UNIX gets SYSDBA privileges automatically in all versions. A parallel policy was introduced for Windows in v.2.1 for network administrators and was modified in v.2.5. The database Owner (the user which created the database) has full privileges to that database.

For all other users, access to objects in a database is by “opt-in” SQL privileges only—mentioned next.

At v.2.5, some mechanisms were introduced to make it possible for the SYSDBA to confer administrative rights on ordinary users via a privileged SQL role, `RDB$ADMIN`.

Database Security

All users except those with full privileges must be granted rights to each object to which they are to be allowed access. The SQL `GRANT` statement is used for assigning privileges.

Firebird supports the SQL `ROLE`. A user-defined role must first be created using a `CREATE ROLE` statement, and committing it. Groups of privileges can be granted to a role and then the role can be granted to a user. In order to use those privileges, the user must log in to the database using both user name and role name.

Databases created under Firebird 2.5+ (ODS 11.2 or higher) come with an inbuilt role, `RDB$ADMIN`, which the SYSDBA can grant to any user, to confer SYSDBA rights to that user in the current database.

Refer to Chapter 37, *Database-Level Security*.

The Firebird Server Models

Firebird servers come in three flavours—Superserver, Classic server and Supersclassic—to cater for differing user demand. Any can be scaled up or down to handle the simplest to the most complex configurations.

Resource Usage

Firebird server software makes efficient use of system resources on the host computer.

- The Superserver process uses approximately 2MB of memory. Each Superserver client connection is likely to add about 200KB to server memory consumption, more or less, according to the characteristics of the client applications and the database design.
- Each Classic connection starts its own server process—or server thread in Superclassic mode—consuming about 2 Mb.
- Superclassic, introduced with Firebird 2.5, runs as a parent process that launches a thread for each connection. Each thread will consume approximately the same amount of RAM as a single Classic process.

Server cache memory consumed depends on database page size, configuration and the process model chosen. Cache is configured in database pages, so memory usage by the cache is calculated by multiplying the number of pages by the page size. The default page size is 4 Mb (4096 bytes) for v.2.0+, 2Mb for v.1.5 and 1 Mb for v.1.0.

- On Superserver, a common cache configuration for a network of 20-40 concurrent users is likely to be in the range of 8 to 64 Mb per database. For each database, the cache is shared as a pool by all connections.
- Each Classic server process or Superclassic thread is assigned a static cache—the default is 75 pages per user. Multiply this by the page size to calculate the amount of RAM allocated for each attachment.
- Like a Classic process, each Superclassic attachment uses an individual database cache..

Servers other than v.1.0.x will also use RAM to speed up sorting, if it is available.



Don't overlook the memory limitations of 32-bit processes: a 32-bit process cannot use more than 2 Gb of RAM, even on a 64-bit machine, regardless of how much is installed.

Disk space for a minimal Firebird installation ranges from 9MB to 12MB, depending on platform. Additional disk space is required for temporary storage during operation and additional memory could be needed for database page caching. Both are configurable according to performance demands and the likely volume and type of data to be handled.

Running Superserver as an application on Windows

On Windows server platforms the normal mode of operation for a Firebird server is to run as a service. One advantage of this is that nobody needs to be logged in to the host machine in order to make the service available to remote clients.

It is possible on Windows to run the server executable as an application. The executable runs in the user's application space, which means that the user must be logged in with the fbserver application running. The implications for physical security are obvious.

Older versions of Firebird can run on very old Windows versions, viz., Windows 9x, ME, NT4, Windows 2000. Whilst the last two provide service capability, the others do not. Running Firebird as an application is the only option on Windows 9x and ME hosts.



It is a mistake to assume that Firebird will continue to support obsolete operating systems for ever. Support for Windows 9x and NT4 was dropped at v.2.0 and for Windows 2000 at v.2.1. It does not imply that newer versions will not run on those platforms, but it does mean that no QA testing is performed on them.

Comparing Superserver, Superclassic and Classic Architectures

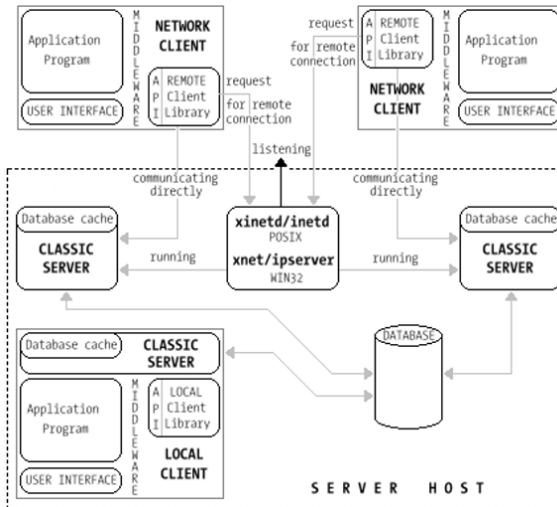
While Superserver and Classic/Superclassic share many common characteristics—indeed, they are built from the same code-base—they present quite distinct models of operation “under the hood”.

Executable and processes

Classic—Runs one server process per connection, on demand. When a client attempts to connect to a Firebird database, an instance of the *fb_inet_server* process is initiated and remains dedicated to that client connection for the duration of the connection. When the client detaches from the database, the server process instance ends.

- On POSIX, the the *[x]inetd* daemon monitors the network for attachment requests and starts each *fb_inet_server* process
- On Windows, a parent instance of the *fb_inet_server* service monitors for attachment requests.

Figure 1.1 The Classic network and process model

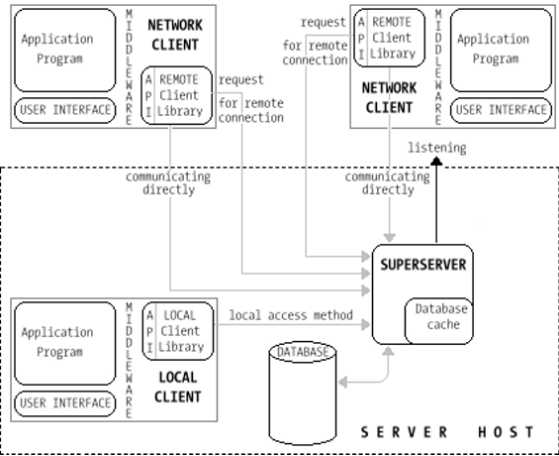


Superclassic—Similar to Classic except that, to attach clients, the listening process launches threads, rather than separate processes.

- On POSIX, the Superclassic listener is the executable *fb_smp_server*, a special build of *fb_inet_server* that does not use *[x]inetd* at all.

Superserver—Runs as a single invocation of the *fbserver* executable. The *fbserver* process is started once, by a system boot script or by the system administrator, and stays running, waiting for connection requests. Threads are created or re-used for attachments. The process is terminated by an explicit shutdown.

Figure 1.2 The Superserver network and process model



Lock management

Classic/Superclassic—Each client's server process has its own, dedicated database cache and multiple processes contend for access to the database. A Lock Manager subsystem—*fb_lock_mgr*—uses inter-process communication (IPC) methods to arbitrate and synchronize concurrent page access among the processes.

SuperServer—The lock manager is implemented as a thread within the *fbserver* process and uses inter-thread communication mechanisms instead of POSIX signaling.

Resource use

Classic/Superclassic—Each instance of *fb_inet_server* is allocated a static cache of database pages in its memory space. Resource growth per additional client connection is therefore linear. However, when the number of concurrent connections is relatively low, Classic uses fewer overall resources than Superserver.

Superclassic—Makes better use of resources overall than Classic on a well-provisioned 64-bit system, since it is cheaper to thread process instances than to fork them. When a Superclassic server is serving clients to multiple databases, the resource load can be spread across multiple CPUs on multi-core systems.



Deploying Superclassic on 32-bit servers is not recommended.

Superserver—Employs one single cache space which is shared by client attachments, allowing more efficient use and management of cache memory when the number of simultaneous connections grows larger.

Multiple processors

Although Firebird does not support splitting the load from one process or thread across multiple CPUs or cores, its Classic and Superclassic models can make use of the capabilities of multi-processor host machines for load distribution.

SMP with Superserver on Windows

On Windows, SMP machines often exhibit problems with Superserver, specifically a “see-saw” effect, whereby Windows will shift the entire process from one CPU or core to another. For this reason, Superserver for Windows uses only the first processor or core in your computer, by default. The **CpuAffinityMask** parameter in `firebird.conf` allows you to change it so Firebird affines to a different CPU or so that you can test whether your particular Windows/hardware combination suffers from the “see-saw” effect. Unfortunately, most do.

For the v.2.5 release, Superserver was improved to the extent that the process is able to utilise multiple CPUs or cores if it is accessing multiple databases concurrently. To enable SMP support for Superserver for a multi-database host environment, adjust **CpuAffinityMask** and test thoroughly for both good and (known) bad effects.

For a single-database system, the situation is unchanged—CPU affinity should be restricted to one CPU or core.



*All other servers (including Superserver for Linux) ignore **CpuAffinityMask**.*

Crash behaviour

Both Superserver and Superclassic use a single process and launch threads for client sessions. A server crash therefore takes out all client sessions with it. If a Classic server process crashes, the other connections continue unaffected.

Firebird Guardian—fbguard

If the optional Guardian service is used with Superserver (any platform) or Superclassic (POSIX only) it will attempt to restart the `fbserver` or `fb_smp_server` process if it should terminate abnormally.

The Guardian gives some extra reliability because it automatically restarts a crashed Superserver. On Windows server platforms, services can be configured at OS level to behave this way, which means the Guardian can be regarded as superfluous for that environment.

The obsolete Windows 9x and ME platforms do not provide services capability. There and on other Windows platforms where Firebird must run as an application, Superserver gains a benefit from running under the Guardian.



The Guardian should not be used with Classic as it can cause the untoward side effect of starting “ghost processes” after abnormal termination of a bona fide Classic process.

Local access method

Superserver—Expects applications to use a network method for I/O requests and satisfies those requests by proxy.

- On POSIX, SuperServer does not support direct local access—local connections to Superserver are made through the localhost server (at IP address 127.0.0.1, by convention).
- A Windows server and local client can simulate a network connection in the shared inter-process communication (IPC) space. This mechanism—referred to as the “local protocol” and `ipserver`—can not handle multiple connections safely. Beyond Firebird 1.5, an improved local access method for `ipserver` uses the XNET subsystem.

Classic/Superclassic—On POSIX, application processes that are running on the same machine as the server and databases can perform I/O on database files directly, using the integrated client/server library `libfbembed.so`.

- On Windows, in versions prior to v.2.0, “Windows local protocol” is not available on Classic.

Performance

As a general observation, the three models perform equally well on equivalent hardware. However, no deployment scenario is exactly like another. Classic uses fewer resources if the number of connections is low. On well-resourced 32-bit hosts it is most recommended for environments that would gain from being able to use multiple CPUs and have adequate resources to support the private database caches. Superclassic may perform better than Classic on larger sites running 64-bit servers.

Superclassic and Superserver use system resources more efficiently than Classic when the number of simultaneous connections grows into the hundreds. Superserver is the most efficient of the three, because all connections share the same cache. However, Superserver does not support SMP and is more limited than Classic/Superserver in the number of simultaneous connections it can carry.

If you are unsure which model is going to perform best for your conditions, you might consider starting with Superclassic for your 64-bit host machine or Classic if you are constrained to 32-bit. You can always switch to another model later: the internal differences are in the server, not in your databases.

Page Cache size differences

The page cache is a dedicated block of RAM where a Firebird process accumulates an image of the database pages it visits. When properly configured, the cache progressively reduces the amount of disk reads it has to do, particularly on frequently accessed tables.

The different server models use cache differently. If you are swapping back and forth between Superserver and Classic/Superclassic, it is essential to get acquainted with the page cache size differences and to understand how page size affects the amount of RAM reserved for the cache.

The default 2048 pages assigned for Superserver’s shared cache is far too large for a Classic/Superclassic setup, where each client connection uses a private cache, whilst the default Classic cache of 75 pages is of little use to Superserver’s cache sharers. Inattention to this one detail of difference accounts for the majority of “performance problems” that appear in the support lists!

You should also bear in mind that, if you are running a 32-bit Firebird server process, the entire RAM usage of the process is limited to 2GB, regardless of how much RAM is installed.

Events

Firebird servers can capture events in the execution of stored procedures and triggers and, on committal of the transaction, return messages to client applications that are listening for specified events. A TCP/IP network channel is created for event traffic.

The traditional architectural differences between the models has meant that the full features of the event mechanism are not available in some topographies where Classic is the model of choice. As versions progress, the conditions improve but only in v 2.5 and higher versions are events available to all topographies regardless of which model is in use.

Events port

By default, the server chooses an available TCP/IP port at random for event traffic. If the server is behind a firewall or if connections are made through a secure tunnel, a specific events port must be assigned and the tunnel or firewall configured to enable traffic. On the Firebird side, the `firebird.conf` configuration parameter for configuring the port is

RemoteAuxPort.

Embedded Server

An embedded server comprises a single-user server folded together with a client instance, providing direct, stand-alone access to databases directly from applications. On Linux, it was always available by attaching using the “direct local” client for Classic, named `libfbembed.so`. On Windows, a functionally similar option was introduced in Firebird 1.5, but embedding Superserver with a single client.

In v.2.5, the architectures of the embedded servers on all platforms were unified so that all use the Superclassic model, enabling multi-threading for the embedded client.

The embedded servers are described in more detail later in this chapter, on [page 18](#), in the topic ***Typical Deployment Topologies.***

Introduction to Client/Server

Generically, a client/server system is a pair of software modules designed to communicate with each other across a network using an agreed protocol. The client module makes requests across the network to a listening server program and the server responds to the requests.

If you are new to client/server systems, the rest of this chapter may help you to understand the concept behind the architecture and how radically different it is to desktop or file-served data storage systems.

Servers and Clients

To take an example of a client/server arrangement that is not primarily a database management system, an e-mail client dispatches a message across a network to an e-mail server, with a request directing the server to send the message to an address on a server somewhere. If the request complies with the agreed protocol and the destination address is valid, the server responds by repackaging the message and dispatching it, returning an acknowledgment to the client.

The key principle is that the task is split—or distributed—between two separate software components that are running independently on physically separate computers. The model does not even require that the components be running on compatible operating or file systems. The e-mail client could be any email client program that runs on a Windows, Mac or any other system and the e-mail server is often running on a UNIX or Linux system. The client and the server programs are able to cooperate successfully because they have been designed to be *interoperable*.

In a client/server database system the model is no different. A host machine in a network is running a program that manages databases and client connections—a database server. It occupies a node that is known to client programs running on machines at other nodes in

the network. It listens for requests from the network, from clients that want to attach to a database and from other clients that are already attached to databases.

As with the e-mail example, the protocol for communication is at two levels. Like the e-mail system, it uses a standard network protocol and overlays it with other, purpose-specific protocols. For e-mail the overlay will be POP3, IMAP, SMTP; for the database system, it takes form at several levels, as protocols for database connection, security, database transfer and language.

Client/Server vs File-Served Databases

File-sharing systems could be cited as another example of client/server systems. File servers and filesystem servers serve client requests for access to files and file systems, sometimes in very sophisticated ways. NFS, SMB and the Windows native networking services are examples. The file server gives clients access to files that the client machine can read into its own memory and write to, as though it were performing I/O on its own, local storage system.

A desktop-based data management system, lacking its own, internal provisions to manage I/O requests from a network, is itself a client of the file-server. When it receives I/O requests from its own clients, it depends on operating system controls to provide the central locking and queuing system necessary to manage conflicting requests.

These file-served DBM systems are not client/server database systems. Both the client and the DBMS software are clients to a file-sharing server. Whilst the flow of input and, often, output are to some extent managed by the DBMS program, physical data integrity is under the control of the filesystem services.

In a client/server database relationship, clients—even if located on the same machine as the server—never get closer to the physical data than sending messages to the server about what they want to do. The server processes the messages and executes the requests using its own code and, in advanced systems like Firebird, its own disk management and accounting system. The server program performs all of the physical changes to metadata and data storage structures within a physical on-disk structure that is independent of the host's filesystem-level I/O layer and inaccessible to it.

Characteristics of a Client/Server DBMS

A fully-featured client/server database management system is realised by the degree to which it meets the five recognised reliability criteria for robust data storage and retrieval systems, viz., scalability, interoperability, data protection, distribution of function and standardization.

Scalability

The advent of comparatively inexpensive PC networks throughout the 1980s and '90s provoked an increasing demand for scalable information systems with user-friendly human interfaces. Spreadsheet and desktop database software and graphical interfaces gave non-technical users a taste for the power of desktop computing. Once the sharing of files across networks and amongst different types of software became standard practice in large enterprises, the customers demanded more. Desktop and LAN-based data management also came within reach of the smallest businesses. Today, it is almost unthinkable to design an enterprise information system under the monolithic model of the mainframe and the text terminal.

Scalability is judged in two dimensions: horizontal and vertical. Horizontal scalability is the capacity of the system to accommodate additional users without impact on the capability of the software or resources. Vertical scalability corresponds with what it takes to reproduce the system on simpler or more complex platforms and hardware configurations in response to variations in load and access requirements. It ranges from the low end—making the system available to users of mobile devices, for example—to a high end that has no conceptual limit.

Interoperability

Client/server architecture for database systems evolved as a response to the fragility, low load capacity and speed limitations of the file-sharing database model in PC networks as multi-user demand increased. Its emergence coincided with the parallel development of the SQL language. Both were, in part, strategies to neutralize the dependency on mainframe hardware and software that prevailed in the 1980s. True client/server architecture for databases is heterogeneous and interoperable: it is not restricted to a single hardware platform or a single operating system. This model allows clients and servers to be placed independently on nodes in a network, on hardware and operating systems appropriate to their function. Client applications can communicate simultaneously with multiple servers running on disparate operating systems.

Data protection

The great flaw with file-served database systems is the vulnerability of data to error, damage and corruption when they are physically accessible to file-sharing clients and placed under the direct control of humans. In the client/server database model, client applications never touch the physical data. When clients request changes to data state, the server subjects the requests to rigorous validation. It rejects requests that fail to comply with internal rules or metadata rules. When a write request is successful, the actual change of database state is executed entirely by code resident in the server module and within disk structures that are under the server's control.

Distribution of function

The client/server model allows the areas of work in the system to be distributed appropriately and effectively amongst hardware and software components. The database server takes care of storing, managing and retrieving data and, through stored procedures, triggers or other callable processes, provides the bulk of data processing capacity for the system. The client process manages the "sharp end" for applications by translating their requests into the communication structures that form the protocols for database and data access.

Applications are the dynamic layer in the model. They provide a potentially infinite variety of interfaces through which humans, machines and external software processes interact with the client process. For its part, the client module exposes itself to applications via a comprehensive, preferably standardized, language-neutral application programming interface (API).

In some systems, it is possible for applications to act almost entirely as deliverers of information and receptors of input, delegating virtually all data manipulation to the server's processing engine. This is an ideal for client/server systems because it locates CPU-intensive tasks where processing power is concentrated and leaves applications to utilize the capacity of the workstation to deliver the best-performing user interfaces.

At the other end of the scale are systems in which, through poor design or an impractical concern for interoperability, virtually all data processing is performed on client

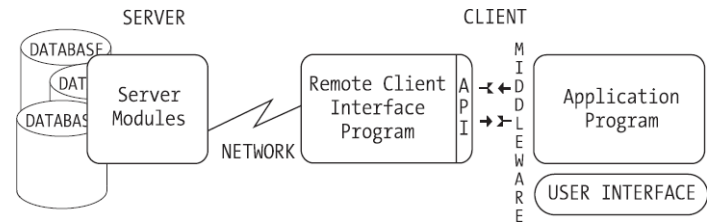
workstations. Such systems are often characterized by poorly performing user interfaces, delays in synchronizing database state and unresponsive networks.

Between heaven and hell are well-performing client/server database systems that make good use of processing capacity on servers while retaining some data processing functions on workstations where it is justified to reduce network traffic or improve the flexibility of task implementations.

The two-tier model

The following diagram conceptualizes the classic two-tier client/server model. The middleware layer, which may or may not be present, represents a driver, such as ODBC, JDBC or PHP, or a data access component layer that is integrated with the application program code. Other layerings at the client side are possible. Applications may also be written to access the API directly, with no middleware.

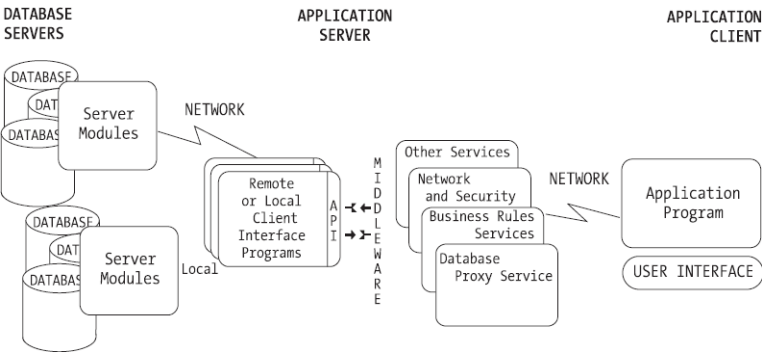
Figure 1.3 Client/server: the two-tier model



The n-tier model

Upscaling and requirements for greater interoperability give rise to a model with more layers. The client interface moves to the center of the model and is joined by one or more application server layers. In this central complex will be located middleware and network modules. The application layer becomes a kind of database superclient—being served by multiple database servers sometimes—and itself becomes a proxy server for database requests from applications. It may be located on the same hardware as the database server but it could just as well be running on its own hardware.

Figure 1.4 Client/server: an n-tier model



outsource most of the administrative functions. Systems like this have capacity for growth without major upheaval.

The single-user model

All Firebird servers can accept local clients. The connection protocols and options vary according to the server model you choose. Single-user installations fall into two categories:

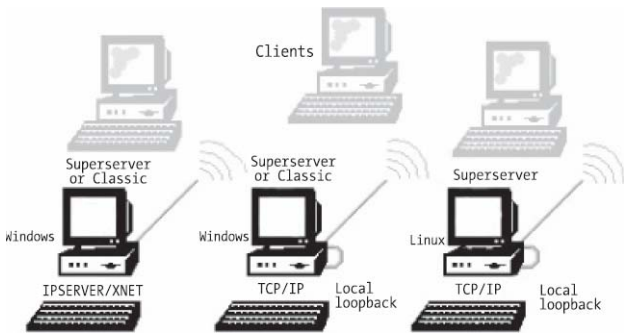
Stand-alone server: in this model, the server is installed and running on the machine. Local attachments are made via network-style protocols, using the normal client libraries.

Embedded server: no server is installed. The client and server programs are rolled into a single dynamic library or shared object that is invoked by the application and starts a single, exclusive server process on attachment. When the application program terminates, the server process is unloaded.

Stand-alone server

In the stand-alone client/server model, the localized client attaches to the running server using a local protocol. The server can listen for connections from remote clients whilst a local client is attached. Figure 1.6 illustrates the options.

Figure 1.6 Stand-alone servers



The first example shows the “local connect” model. Up to and including Firebird 1.5, the *ipserver* subsystem simulates a network connection within the same block of inter-process communication space. Beyond v.1.5, the local protocol uses the faster, more robust, native XNET subsystem. A “local connect” is supported for Classic on POSIX, in the form of a client embedded with a single-client server instance.

In the other two, on Windows, Linux or any other supported platform, the TCP/IP local loopback protocol is used with the Superserver. It is a regular TCP/IP attachment to the special IP address 127.0.0.1 which most TCP/IP subsystems install by default as localhost. On Linux, the v.1.5 Classic server can be used in this mode, provided the client library *libfbclient.so* is used.

Embedded server

Embedded servers are supported on both Windows and Linux/UNIX platforms, although, prior to v.2.5, the implementation models are different. In v.2.5, the platforms are more integrated with one another architecturally, with the embedded server implemented as a flavour of Superclassic and the embedded client being threadable. It also became possible to attach to the same database simultaneously using both embedded and a Classic thread.

For the older versions, the Windows library, `fbembded.dll`, embeds a single-client Superserver, which runs an exclusive Superserver process, while on Linux, the `libfbembded.so` client referred to in the previous sub-topic embeds a Classic server that connects directly to databases. On Linux it is not exclusive: remote clients using `fbclient.so`, another `libfbembded.so` instance or `fbclient.dll` can connect concurrently to `fb_inet_server` instances through the `[x/inetd]` network daemon. (*launchd*, in the case of MacOSX).

Embedded servers are a deployment option, intended for single-user, stand-alone use on a single workstation. As such, they have some limitations that will get in your way, especially if you are developing applications on Windows.

- The embedded client/server arrangement on Windows can use only the “Windows local” access method.
- In versions prior to v.2.5, the embedded server engine, being Superserver, employs a write lock on a database after the initial attachment, to protect it from a Windows path anomaly that can cause corruption. The older `fbembded.11` supports one and only one connection to each local database.
- Client applications connect to the embedded servers without the degree of authentication that challenges remote clients. Prior to v.2.5, server-level authentication on Windows is bypassed completely. It is always essential to protect servers from unauthorised physical access, of course, but workstations are often running embedded applications in public areas, making them especially vulnerable.

The Embedded Client as a Network Client

Besides its ability to connect to and access one or more local databases exclusively through the embedded server, the embedded client in `fbembded.dll` or `libfbembded.so` can also connect as a regular network client to databases on other servers.

Firebird servers in distributed environments

A detailed discussion of distributed transaction processing (DTP) environments is beyond the scope of this guide. However, suffice it to say, the Firebird full servers sit well in a variety of DTP scenarios.

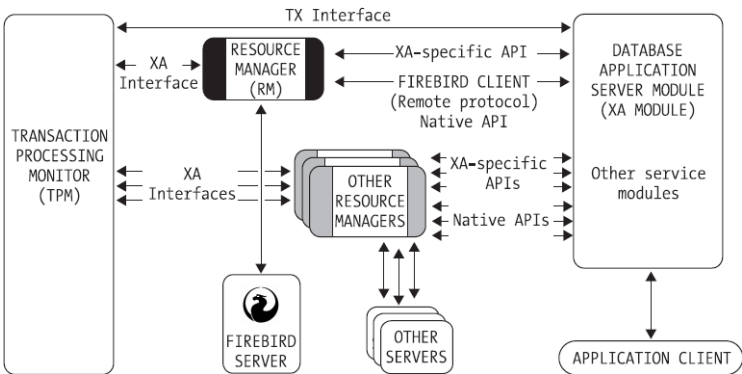
The Open Group defined the X/Open standard for DTP, envisioning three software components in a DTP system. The XA specification defines the interface between two of them, the transaction manager and the resource manager (RM). The system has one RM module for each server and each RM is required to register with the transaction manager.

Figure 1.7 illustrates how a Firebird server could be slotted into an XA-compliant DTP environment. The database application server module provides a bridge between high-level user applications and the resource manager, encapsulating the XA connection. The resource manager (RM) performs a client-like role to negotiate with the database server for access to data storage.

The encapsulation of the XA connection enables the application developer to build and execute SQL statements against the resource manager. Transaction demarcation—which requires two-phase commit capability on the part of all servers—is moderated by the global transaction processing monitor (TPM). Cross-database transactions under the management of a transaction manager are made through a two-phase commit process. In the first phase, transactions are prepared to commit; in the second, the transaction is either fully committed or rolled back. The TPM will alert the calling module when a transaction does not complete for some reason.

The TPM coordinates distributed transactions in multiple database systems so that, for example, one transaction can involve one or more processes and update one or more databases. It keeps account of which resource managers are available and involved in transactions.

Figure 1.7 Firebird in distributed transaction processing environments



The framework supports multiple databases per server and multiple servers, which need not be all Firebird. Up to and including Version 2.5.x, Firebird does not support spanning a single database across multiple servers or serving a database that is not controlled by its host machine.

Transaction Server frameworks

Microsoft Transaction Server (MTS) with COM+ is one such scenario. MTS/COM+ provides a process-pooling environment that encompasses the deployment and management of business logic components, including auditing control, security and performance monitoring. One of its most significant features is declarative transaction management. Transactions initiated by MTS/COM+ are controlled by the Microsoft DTC (Distributed Transactions Coordinator), an XA resource manager. The native Firebird interface requires an ODBC or OLE-DB provider that both supports both Firebird's two-phase commit capability and the MTS/COM+ call context.

Terminal Servers

Firebird is successfully deployed in Citrix and Microsoft Terminal Server frameworks. Protocol is TCP/IP with attachments connecting to network IP addresses in all cases.



It is strongly inadvisable to install the terminal server and the database server on the same node. However, in any situation where the application server is running on the same physical node as the database server, the connection must be to the IP address of the node. TCP/IP local loopback (localhost as server) is not possible.

Firebird Clients

A client on a remote workstation requires a client library and an application program that can interact with the application programming interface published in the library. A client library provides the wire protocol and the transport layer that your client application uses to communicate with the server. The standard shared library for Windows

clients is a Windows DLL. For POSIX clients, it is a shared object (.so library) or, for MacOSX, a .dylib library. The size of the standard client library is approximately 350 Kb.

Some access layers, such as the Firebird .NET provider and the JayBird Java drivers, replace the standard client library and implement the Firebird wire protocol directly. Another option is an embedded server—a library that merges both a client and a server instance—for single-user use.

A client workstation can also, optionally, have a copy of the current `firebird.msg` file, or a localized version, to ensure that correct server messages are displayed when applications include calls to the Services API.

Generally, you would install a copy of the client library on the host server, for use with several of the Firebird command-line utilities and/or any server-based management programs you might wish to use locally. Many of these utilities can be run remotely, however. A remote system administrator can manage some of the essential services provided by these utilities by accessing them through a host service controller interface.

A host-based client application, such as a web application, needs the client layers present.

What is a Firebird Client?

A Firebird client is an application, usually written in a high-level language, that provides end-user access to the features and tools of the Firebird database management system and to data stored in databases. The *isql*/interactive SQL utility and the other command-line utilities in your `$firebird$/bin` directory are examples of client applications.

Firebird clients typically reside on remote workstations and connect to a Firebird server running on a host node in a network. Firebird also supports a stand-alone model allowing client applications, the Firebird client library and the Firebird server to execute on the same physical box.

Client applications need not involve end-users at all. Daemons, scripts and services can be clients.

Firebird is designed for heterogeneous networks. Clients running under one operating system can access a server on a different operating system platform. A common arrangement is to have Windows XP/7 and Linux workstations concurrently accessing a departmental server running Windows Server 20xx, or any of several flavors of UNIX or Linux.

In the client/server model, applications never touch the database directly. Any application process converses with the server through the Firebird client library, a copy of which must be installed on each client workstation. The Firebird client library provides the application programming interface (API) through which programs make function calls to retrieve, store and manipulate data and metadata. Generally, other layers are also involved in the interface between the application program and the Firebird client, that surface generic or application-language-specific mechanisms for populating and calling the API functions.

For Java development, Firebird's stable of supported drivers includes the Jaybird JDBC/JCA-compliant Java driver for flexible, platform-independent application interfacing between many open source and commercial Java development systems and Firebird databases.

Open source and third-party interfacing components and drivers for many other development platforms, including Embarcadero Delphi®, Kylix® and C++Builder®, commercial and open source C++ variants, Python, PHP and DBI::Perl are available. For .NET development, a Firebird .NET provider is under constant development, keeping pace

with Microsoft’s rapid scene changes. Contact and other information can be found in Appendix XIII, Application Interfaces.

The Firebird Client Libraries

The Firebird client library comes in a number of variants, all of which surface an identical API to applications, for the server release version to which they apply.

The client library uses—in most cases—the operating system’s client network protocols to communicate with one or more Firebird servers, implementing a special Firebird client/server application-layer interface on top of the network protocol.



It is important not to mismatch the client library release version with the release version of the server. Use a v.2.5 client with a v.2.5 server. It is normally fine to use a higher client with a server of a lower version. A mismatch is likely to occur when a lower client accesses a higher server, owing to potential API enhancements as major versions rise. Realize too that the client for one version may or may not be installed to the same location as the client for another. When redeploying with a new version, be sure to study the readme and installation documents—located in the Firebird root directory and /doc subdirectory of the server—for information that may outdate the information in this guide.

The Firebird API

All client applications and middleware must use the API in some way to access Firebird databases. The Firebird API is backwardly compatible with the API from old InterBase versions. The **InterBase 6.0 API Guide** (ApiGuide.pdf) is downloadable from many places on the Web, including the InterBase archives on the IBPhoenix website, <http://www.ibphoenix.com>. It provides reference documentation and guidelines for using the API to develop language interfaces and even applications, if you want to do things the hard way.

Enhancements made to the API since the Firebird fork are documented in the Firebird release notes and, to a limited extent, in the header files distributed with Firebird.

More About Clients

Client-only installs—next chapter, [Performing a Client-Only Install](#)
Location of libraries—Appendix X, [Default Disk Locations](#)
Application interfaces—Appendix XIII, [Application Interfaces](#)

INSTALLATION

This chapter describes how to obtain an installation kit for the platform and version of Firebird server that you want to install on your server machine. The full installers install both the server and the client on a single machine.

Remote clients do not require the server at all. The procedure for installing the Firebird client varies somewhat, according to platform. For instructions, refer to the topic *Performing a Client-Only Install* in this chapter.



If you are new to Firebird, do not attempt a client-only install until you have worked out how all the pieces fit together in the default installation.

System Requirements

- Server memory
- Installation drives and disk space (*Installation Drives*)
- *Minimum Machine Specifications*
- *Operating System*

Server Memory (All Platforms)

Estimating server memory involves a number of factors:

- Firebird server process: The Firebird server process makes efficient use of the server's resources. The Superserver utilizes around 2MB of memory. On POSIX, the Classic server uses no memory until a client connection is made. On Classic for Windows, a small utility service is listening for connection requests.
- Client connections: Each connection to the Superserver adds approximately 115KB, more or less, according to the style and characteristics of client applications and the design of the database schema. Each connection to the Classic server uses about 2MB.

- Database cache: The default is configurable, in database pages. The Superserver shares a single cache (with a default size of 2,048 pages) among all connections and increases cache automatically when required. The Classic server creates an individual cache (with a default of 75 pages) per connection.

Databases with large page sizes consume resources in larger chunks than do those with smaller page sizes. Resource usage on the Classic server grows by a fixed amount per client attachment; on Superserver, resources are shared and will grow dynamically as needed. Extra available RAM will be used for sorting if enough is available to accommodate one or more of the intermediate sets; otherwise those sets are written to temporary disk space.

Installation Drives

Firebird server—and any databases you create or connect to—must reside on a hard drive that is physically connected to the host machine. You cannot locate components of the server, or any database, on a mapped drive, a filesystem share, or a network filesystem.

Disk Space

When estimating the disk space required for an installation, consider the sizes of the following executables. Disk space, over and above these minimum estimates, must also be available for database files, shadows (if used), sort files, logs, backups.

- Server: A minimal server installation requires disk space ranging from 9MB to 12MB, depending on platform and architecture.
- Client library: Allow 350KB (embedded: 1.4MB–2MB).
- Command-line tools: Allow ~900KB.
- DB administration utility: Allow 1MB–6MB, depending on the utility selected. For a list of free and commercial utilities available, refer to Appendix XIV, **Resources**.

CD-ROM

You cannot run a Firebird server from a CD-ROM. However, you can attach to a read-only database on a CD-ROM drive that is physically attached to the server.

Default Disk Locations

Default disk locations for the standard Firebird installations are listed in Appendix X, *Default Disk Locations*.

Minimum Machine Specifications

Minimum specifications depend on how you plan to use the system. You can run a server and develop database schemas on a minimally-configured PC—for v.1.5 and later, a 586 with 128MB should be regarded as minimum. Windows is more demanding on CPU and memory than a Linux server running at the console level. Operating system versions will influence the requirements: some UNIX platforms are expected to demand more resources at both server and client, and the requirements of some Windows versions push the baseline out, independent of any software requirements.

SMP and Hyperthreading Support

Firebird Superserver, Superclassic and Classic can use shared memory multiprocessors on Linux. On Windows, SMP support is available only for Classic.

Hyperthreading is uncertain and seems to depend on several variables, including operating system platform, hardware vendor, and server version. Some users have reported success; others have had problems. If you have a machine with this feature, try your selected server with it enabled initially, and be prepared to disable it at the BIOS level if performance appears slow or unstable.

On Windows, for Superserver and Superclassic, processor affinity can be configured at the server level in `firebird.conf` (v.1.5 and above) or `ibconfig/isc_config` (v.1.0.x). The CPU affinity mask should be set to a single CPU on a SMP machine. For instructions, refer to the entry [CpuAffinityMask](#) in Chapter 34, **Configuration Parameters in Detail**.

Operating System

Table 2.1 shows the minimum operating system requirements for running Firebird servers. However, always check the README files in the `/doc/` directory of your kit for late-breaking information about operating system issues.

Table 2.1 Firebird Minimum Operating System Requirements

Operating System	Version	Requirements/Recommendations
Microsoft Windows	All	Visual C++ Runtime libraries –msvcp60.dll or higher required for v.1.5 (or C Runtime library msvcrt.dll v.6. or higher required for v.1.0) –msvcp70.dll or higher required for v.2.0 –msvcp80.dll or higher required for v.2.1 and 2.5 Database files should not use the extension “.gdb”.
	Server 2003 and 2008 versions and spin-offs, 64-bit	Recommended for deploying Firebird to medium to large enterprise sites, if Windows is a requirement. Databases should be on partitions that have the VSS (volume shadowing) feature disabled.
	Server 2003 and 2008 versions and spin-offs, 32-bit	Usually suitable for deploying any Firebird version ¹ to small enterprise sites of up to 200 users. Databases should be on partitions that have the VSS (volume shadowing) feature disabled.
	Win 7 64-bit	Should be suitable as a server for small to medium sites, any Firebird version 2.1 or higher

Operating System	Version	Requirements/Recommendations
	XP 64-bit, Vista 64-bit	Fully patched installation may be suitable as a server for small sites, any Firebird version 2.1 or higher. Databases should be on partitions that have System Restore disabled.
	XP 32-bit, Vista 32-bit, Win 7 32-bit	OK for single-user and very small workgroups, using any version in Classic or Superserver models; not suitable as a server otherwise. Service pack 3 required. Databases should be on partitions that have System Restore disabled.
	Windows 2000	Service pack 4. OS not recommended for Firebird versions higher than v.1.5.x
	NT 4.0	Requires Service Pack 6a. OS not recommended for Firebird versions higher than v.1.5.x
	95/98/ME	Not recommended for Firebird versions higher than v.1.5.x.
Linux	Any	Many Linuxen now have their own Firebird package maintainers with most or all released versions available from distro repositories Firebird 2.5 Classic and Superclassic may not run satisfactorily with glibc runtimes 2.7 or lower. Use glibc 2.9 or higher. Kernels that do not support the New POSIX Threading model (NPTL) will not work with Superserver Firebird 2.1 and higher or Superclassic (Firebird 2.5 and higher).
	OpenSuse & family	v.10.1 or higher for Firebird 2 and above, v.8.10 or higher for Firebird 1.5, v.7.2 or higher for Firebird 1.0.
	Fedora & Family Debian, Ubuntu and Family Mandriva & Family	
MacOSX/Darwin		Firebird 2.5 will run only on MacOSX 10.6 (Snow Leopard) or higher versions.
Other platforms	Solaris (Intel, Sparc), FreeBSD, HP-UX 10+	Refer to the relevant Firebird distribution kits for details

1. Because of the memory-addressing limitations on 32-bit systems, Superclassic may be unsuitable if demand for resources becomes sufficient to terminate the server process. Classic will be more robust under these conditions. On the other hand, one over-demanding Classic process could bring the host to its knees with continual swapping.

32-bit or 64-bit

An application compiled as 32-bit can be installed on either a 32-bit or a 64-bit operating system. A 64-bit application cannot be installed on a 32-bit operating system.

A 32-bit Firebird client can connect to a 64-bit Firebird server and a 64-bit client can connect to a 32-bit server. It is the application environment that determines which client library must be used.

- if the application is 32-bit, the 32-bit client and any driver layers (ODBC, .NET provider, et al.) must be 32-bit versions
- if the application is 64-bit, the 64-bit Firebird client library and drivers must be 64-bit
- If the application (32-bit or 64-bit) calls external functions (“UDFs”) and the Firebird server is 64-bit, then the UDF library must be compiled as 64-bit.



Citrix or other terminal server installations are themselves client applications. The client you install in this environment is governed by the architecture of that tier, even if it is running on the same host machine as the server. For example, a 32-bit tier running on a 64-bit operating system that is hosting 64-bit Firebird must use the 32-bit versions of all connecting layers, including the Firebird client.

How to Get an Installation Kit

The main download area for Firebird release kits can be linked from the main Firebird website, <http://firebirdsql.org>. Links on the download pages will take you to <http://sourceforge.net/projects/firebird/files/>.

The main page of the Firebird site usually displays a list of links to the latest releases for Linux and Windows. Other links will point you to distributions for other platforms, that may be in repositories other than Firebird’s Sourceforge repository.



If a file has “src” in its name, it is buildable source code, not an installable binary kit.

Kit Contents

All of the kits contain all of the components needed to install the Firebird server:

- The Firebird server executable
- A number of other executables needed during installation and/or runtime
- Shell scripts or batch files needed during installation, which may also be available as server utilities
- The security database (security2.fdb for “2” series versions, security.fdb for v.1.5.x or isc4.gdb for v.1.0.x;)
- One or more versions of the client library for installing on both the server and the client workstations
- The command-line tools
- The standard external function libraries and their declaration scripts (*.sql)
- A sample database
- The C header files (not needed by beginners!)
- Text files containing up-to-the-minute notes for use during installation and configuration

- Release notes, Quick Start Guide and various README files (these are essential reading)

Kit Naming Conventions

File naming of kits across platforms is not consistent. Alas, it is not even “consistently inconsistent,” with builders often needing to conform to platform-specific conventions or simply making their own rules. However, certain elements in the file names will help you to identify the kit you want.

Classic or Superserver?

In general, the first part of the name string is “Firebird.”

- If a Windows release supports the Classic server (`fb_inet_server.exe`), it will be included in the same kit as the Superserver (`fbserver.exe`). Both 64-bit and 32-bit kits are available as either executable installers or zip archives.
- For POSIX platforms that support both models, separate installers are provided for the Classic/Superclassic server and Superserver in x86 and AMD64 flavours. The kit name will begin with “FirebirdCS” (for Classic/Superclassic server) or “FirebirdSS” (for Superserver).
- For MacOSX, there are different packages for Classic/Superclassic and for Superserver. Recent releases have followed the same convention as the other POSIX kits, of prefacing the package name with “FirebirdCS” and “FirebirdSS”, respectively. Each has at least an Intel x86 platform package, while some versions come in 64-bit flavours. Some versions of Firebird come with PowerPC packages and “lipo” fat client packages as well.
- For minor platforms, the architecture might be less obvious and the first part of the name might be that of the OS or hardware platform.

Version Numbers

All release kit names should contain a dot-separated string of numbers in the following order: version number, release number, subrelease number. For example, “2.0.6” is the seventh subrelease of the “Firebird 2.0” series, while “2.5” or “2.5.0” is the initial release of the “Firebird 2.5” series. Most kits also include the absolute build number (e.g., “2.5.0.26074”). For some Linuxen and some minor platforms, especially those that impose their own naming rules and build on different compilers, version numbers may be less obvious.

Architecture Designators

Where builds are offered for both 32-bit and 64-bit platforms, the infix varies according to chipset:

- Windows: **Win32** and **x64**
- Linux: **i686** and **amd64**
- MacOSX and Darwin: **i386** and **x86-64** (for Intel platforms); **ppc** for PowerPC. The “fat client” packs for x86-64 have the infix **lipo**.
- In the past, some platforms that required a special build to support 64-bit I/O carried the infix “64IO” somewhere in the name string.
- Assume Solaris kits are for Intel unless the “SPARC” tag is present in the kit name.

The download page at <http://firebirdsql.org>, for the particular version you are interested in, is generally the most useful indicator of the minimum chipset supported by the distribution and of any “gotchas” regarding OS and hardware with a specific build.



Do not try to install a 64-bit kit on a version of that OS or hardware that does not support 64-bit I/O.

Installing a Server

Where practicable, executable installers have been provided for all binaries distributed by the Firebird Project. Where there are issues that need your attention, you will find detailed instructions in the release notes, README files, and distribution notes.

The kits install a running Firebird server. “Newbies” take note that there is no visible interface—“GUI”—that pops up when installation is complete. A later topic in this chapter explains how to test that your installation worked.



*If you need the release notes before installing your kit, go to the Downloads > Firebird Database Engine page at the Firebird website at <http://firebirdsql.org> and read or download a copy from there. The **Documentation Index** at the Firebird site also gives access to these downloads.*

The FIREBIRD Variable

FIREBIRD is an optional environment variable that provides a system-level pointer to the root directory of the Firebird installation. If it exists, it is available everywhere in the scope for which the variable was defined.



The FIREBIRD variable is not removed by scripted uninstalls and it is not updated by the installer scripts. If you leave it defined to point to the root directory of a different installation, there will be situations where the Firebird engine, command-line tools, cron scripts, batch files, installers, etc., will not work as expected.

Unless you are very clear about the effects of having a wrong value in this variable, you should remove or update it before you begin installing Firebird 2.1. After doing so, you should also check that the old value is no longer visible in the workspace where you are installing Firebird—use the SET FIREBIRD command in a Windows shell or printenv FIREBIRD in a POSIX shell.

Finalise your Server Choice

Before you begin your installation, you must finalise your choice of which model of the Firebird server you want. If you are still unclear about it, refer back to Chapter 1, **Choosing a Server**.

Reminder: Superclassic is for 64-bit systems

Superclassic was introduced at Firebird 2.5 as a precursor to a new, SMP-aware architecture proposed for Firebird 3. It is not an option with any lower versions.

Superclassic is intended to improve the utilisation of resources on 64-bit servers with high loads, especially those with multiple CPUs and large amounts of RAM. It consumes considerably more machine resource than multiple Classic processes.



Superclassic may be an unsuitable choice if you are installing Firebird on a 32-bit production server.

Uninstall and remove other Firebird servers

If your new installation is to replace an existing version, uninstall and remove the artefacts of the existing server before you begin installing this one.

- If you are unsure about how to do a clean uninstall, see the topic **Uninstalling Firebird** on [page 46](#) of this chapter.

If you intend to run this server concurrently with another Firebird server or an InterBase server, your task will not be quite so straightforward. Visit the topic [Configuring the TCP/IP Port Service](#) in Chapter 33, **Configuring Firebird and Its Environment**, before proceeding with a zip kit installation.

- Before you start, make sure that any application or server instance of Firebird running on the host machine is shut down.

The Firebird Variable

If the Windows installer program finds a value for the optional %FIREBIRD% variable, it will make that path the default location that it offers, instead of c:\Program Files\Firebird\Firebird_2_5. Take care about this: it may or may not be what you desire!

Windows

The official kits for Windows are distributed both as executable installers and as compressed (.zip) ready-built filesystem frameworks. If you are new to Firebird, it is strongly recommended that you use the installer in preference to dealing with the subsequent scripting required for a manual installation from a .zip kit.

The Windows install kits include executables and associated files for both server models: Superserver and Classic/Superclassic. The installer sets default parameters with a number of option fields. A dialog box in the installer will prompt you for the model you want to install (Superserver or Classic) and the other one will not be copied to your disk. If you choose Classic you will be given the option to enable Superclassic.

Do not try to install a Classic or Superclassic server if you already have a Superserver installed, or vice versa. You should be logged in as an Administrator for a normal installation.

V.1.0.x For Firebird 1.0.x, the only model supported on Windows is Superserver.

Windows Platforms

On service-capable platforms—Windows 7, Vista, XP, 2000 and NT—the installer installs Firebird, by default, to run as a service. The service will be installed and started automatically at the end of the installation process and, subsequently, each time you boot up your server machine. To find out how to stop and start the server manually, see Chapter 4, **Operating Basics**.

The very old low-end Windows platforms—Windows 95, 98, and ME—do not support services. The installation will start Superserver as an application, protected by the Guardian

program. If the server application should terminate abnormally for some reason, the Guardian will attempt to restart it.

If you are installing Superserver on a services-capable system, it is up to you whether you take the option to use the Guardian. On those platforms you have the option to configure services, including the Firebird service, to have the operating system attempt to restart on failure.



Do not try to install the Guardian with a Firebird Classic server.

Running the Installer

If you are running Windows XP/Server 2003 or an older version of Windows, check the version of the Windows installer installed on your machine by running *msiexec.exe* from a command prompt. A help screen will be displayed that shows the version. If it is earlier than v.3.0 you must update.

To start the Firebird installer, double-click on the “.exe” file for your chosen version and follow the prompts. If you are studious in following the advice given so far, taking care that the server model you select (Classic/Superclassic/Superserver) is the one you really want, the install should be a walk in the park.

- Make sure the Guardian option is unselected if you choose Classic.

Installing from a ‘zip’ kit

To use a “.zip” kit, you will require a utility, such as 7zip, WinZip, PKZip or WinRAR, to inspect and/or extract the contents before you begin.

Extract the contents to the root location where you want to install Firebird. Let’s suppose you have decided to create a folder for it called C:\Programs. You should end up with a file structure along these lines:

```
C:\Programs\Firebird\Firebird_2_5
C:\Programs\Firebird\Firebird_2_5\bin
C:\Programs\Firebird\Firebird_2_5\doc
C:\Programs\Firebird\Firebird_2_5\include
```

..and so on.

If the kit you are installing is v.2.5 then the Firebird root directory will be C:\Programs\Firebird\Firebird_2_5; similarly, ..\Firebird_2_1, ..\Firebird_2_0 or ..\Firebird_1_5 if you are installing an older version. From here on, we’ll refer to your equivalent of C:\Programs\Firebird\Firebird_2_5 as \$firebird\$.

The Microsoft C and C++ Runtimes

Neither the Firebird server nor a Firebird client will run without (as a minimum) the Microsoft C and C++ runtimes that were present in Microsoft Visual Studio on the computer the binary was built on. The Firebird installer programs for versions 2.1.3 and higher take care of installing the appropriate run-time assemblies—MSVCRT8 at the time of this writing.



By default, the v.2.5 installer and possibly the latter sub-releases of v.2.1 will create a “global assembly” in your \$system\$\winSXS folder, in a sub-folder with a name of the pattern xnn_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.6195_x-ww_44262b86. The characters represented by “nn” will be either “86” for a 32-bit installation or “64” for 64-bit. The sequence after the “_” character in the folder name will

be unique on your machine. The folder contains the files `msvcm80.dll`, `mscvp80.dll` and `msvcr80.dll`.

Manual install of the MSVCRT runtime assemblies

For your zip-kit install and for older versions, you must take care of installing the run time libraries yourself. The Microsoft redistributable packs are distributed in all of the zip kits (including the ones for Firebird Embedded). In the server installation kits, you will find them here:

- For 32-bit: `$firebird$\system32\vcrt8_win32.msi`
- For 64-bit: `$firebird$\system32\vcrt8_x64.msi`



In the older v.2.1.x kits, you may find the equivalent installers in your zip kit as executables with the names `vcredist_x86.exe` and `vcredist_x64.exe`, respectively.

The runtimes for v.2.0 and v.1.5 are `msvcp70.dll` and `msvcrt.dll`. The latter, which is the C runtime, may have a numeric infix. They are usually present in established Windows systems but may be found in the `$firebird$\bin` directory if needed.

Running the Setup Programs

Unzipping the kit does not install Firebird.

After checking and, if necessary, installing the runtimes, you still have some programs to run from the command window. Run them in the order indicated in Table 2.2, Commands for Installing Firebird Manually on Windows.



All commands must be run from the `$firebird$\bin\` directory.

Table 2.2 Commands for Installing Firebird Manually on Windows

Program	Switches	Comments
<code>instreg.exe</code>	<code>install</code>	Optional command, causes the installation path of the directory above the current directory to be written into the registry (HKLM\Software\Firebird Project\Firebird Server\Instances\DefaultInstance)
<code>instsvc.exe</code>	<code>install</code>	Installs Superserver (<code>fbserver.exe</code>) to run as a service
	<code>-d[emand]</code>	Use this switch if you don't want the service to start automatically.
	<code>-g[uardian]</code>	Use if you want the Guardian to start the service and restart it in the event of failure.
<code>instsvc.exe</code>	<code>install</code>	Installs the Classic listener (<code>fb_inet_server.exe</code>) to run as a service
	<code>-c[lassic]</code>	
	<code>-d[emand]</code>	Use this switch if you don't want the service to start automatically

Program	Switches	Comments
instsvc.exe	install	Installs the Classic listener (fb_inet_server.exe) to run in multi-threaded mode (“Superclassic”) as a service. Not always an optimal choice for 32-bit servers and not available in versions prior to Firebird 2.5.
	-m[ultithreaded]	
	-d[emand]	Use this switch if you don’t want the service to start automatically.
instclient.exe	install f[bclient] g[ds32]	Generates a copy of the client library in Windows\system32 using the library name you specify—it can only be fbclient or gds32.
	-force	-force switch can optionally precede the library argument to force overwriting existing library of the same name



Be sure to configure the firebird.conf parameter DefaultDbCachePages appropriately for the server model you have chosen to install.

Installing Superserver to run as an application

It is possible to install Superserver to run as an application, either on demand or automatically at startup. In that case, do not run the *instsvc.exe* command at all. Be aware that, when Firebird runs as an application, the Windows user who started it must be logged in to make remote connections to databases possible.



If you want to make a Windows shortcut to start Superserver as an application, the “run” command is
`$firebird$bin\fbserver.exe -a`
where (as before) *\$firebird\$* is the entire path of your firebird root directory. You can copy this shortcut to wherever you like, including the Startup folder of the user who will use it.

Linux and Many Other POSIX

Both RPM installers and tarballs are available for the “main line” Linux distributions of Firebird versions up to v.2.5.0. Many distributions also have custom, distro-specific packages available in their repositories.

RPM (Red Hat Package Manager)

If your Linux distribution supports RPM installers, you may prefer to choose an RPM kit. Running *rpm* with --i switch will create the directories and install everything required, prompt you to set a password for the SYSDBA user, and start your chosen server.



Do not try to use *rpm --update* to bring any existing Firebird package installation up to date. The Firebird RPM packages do not support it.

Note that not all AMD chipsets are recognised by all Linuxen as compatible with Firebird’s i686 RPM installers. For example, the Firebird RPM kits will not install on a 32-bit Mandriva that is running on AMD Athlon 64. However, the tarballs install flawlessly.



RPM installer packages are to be discontinued at some point during the v.2.5.x series. The reason? Distribution-specific rules have become so diverse over the years that it has

become virtually impossible to produce a RPM package that is sufficiently generic to avoid bumping into them.

glibc Issues (and libstdc++.so.5)

Installation of Firebird versions prior to v.2.1.4 on Linux requires a *glibc* package installed that is equal to or greater than glibc-2.2.5. However, to enable support for some older distros, the generic binaries are built in a compiler environment that will ensure compatibility with the v.2.2.5 kernel. For this reason, the runtime library `libstdc++.so.5` must be present in your system before you attempt to install those older versions of Firebird.

It can be achieved in various ways, viz.,

- by installing a *compat-glibc* package (RedHat, CentOS, OpenSuse, Debian) or a *libstdc++5* package (Mandriva)
- by using a Firebird RPM kit (or other, appropriate package type) provided by your distro instead of the generic one provided by the Firebird Project
- by compiling Firebird yourself, on the same system that you are going to run it on!

Distribution-specific builds

In recent years, much effort has gone into assisting Firebird into the package management systems of a variety of Linux distributions. These days, it is very likely that the PM system of your Linux installation will be able to find and install a distro-specific package of a version of Firebird that you want. Don't overlook the benefits of this method of installation, particularly the automated installation of other packages that Firebird depends on and the relocation of Firebird file assets to comply with the rules of the particular distro.

The instructions here do not necessarily apply to installations from distro repositories, although they might be helpful as a troubleshooting aid if things fail at some point.

Compressed Files (Tarballs)

For Linux distributions that cannot process RPM packages, and for the various UNIX flavors, use the tarball kit (usually `.tar.gz` or `.bz2`), as it will give the experienced Linux hand more control over the installation process. The appropriate decompression utility will be needed on your server for decompressing and “untarring” the kit into the filesystem.

Shell scripts have been provided in the `/bin/` directory of the tarball. In some cases, the distribution notes may instruct you to modify the scripts and make some manual adjustments. Skilled users can also study and adjust the installation scripts to make structures that work on less common distros.

Where possible, the build engineers for each Firebird version and sub-release attempt to document, in release notes and ReadMe texts, any known issues with various kernel versions and distributions. In all cases, read any distributed text files, along with any specific topics in the official release notes that pertain to the version of Firebird that you are going to install.

SYSDBA Password

During the install, a temporary SYSDBA password will be created in `/opt/firebird/SYSDBA.password`, a plain text file. You will need a SYSDBA password in order to attach to any database initially, so write it down somewhere as soon as your install completes. You can change it later, using the script `changedBAPassword.sh` that is stored in `/opt/firebird/bin/`.

Installing a Tarball

Download the compressed tarball you want to install, e.g., FirebirdCS-2.5.0.26074-0.i686.tar.gz, into your home directory, say, ~/Downloads. The extraction and untarring procedure is the same for either a Classic or Superserver tarball kit, viz.,

```
$ cd ~/Downloads
$ tar -xzf FirebirdCS-2.5.0.26074-0.i686.tar.gz
```

The new directory FirebirdCS-2.5.0.26074-0.i686 will appear.



It is fine to simplify the name of the directory, e.g.,

```
$ mv FirebirdCS-2.5.0.26074-0.i686 fireball
```

Classic or Superclassic

Firebird Classic is installed so that the `[x]inetd` daemon listens for remote connection requests and operates one instance of the `fb_inet_server` process for each connection. Each connection therefore has its own copy of the database cache.

Ensure that you have the appropriate daemon installed and running before you install your Firebird Classic kit.

Now, you can run the install script, which will take care of everything to complete the install. You need root privileges to run the script:

```
$ sudo fireball/install.sh
```

Installing and Configuring Superclassic

Superclassic is an alternative mode of operation of the Firebird Classic model that was introduced with the v.2.5 Classic servers. Older versions do not have Superclassic.

In this mode, a specialised process called `fb_smp_server` becomes the listener that is responsible for spawning threaded instances of `fb_inet_server` for each successful connection request. The `[x]inetd` daemon is not required at all for Superclassic.



Although Superclassic is available as a 32-bit application, it is not recommended to try to run it on 32-bit production installations of more than a few users, due to the resource limitations that apply to 32-bit applications. Classic remains the recommended model to use on well-resourced 32-bit servers.

The Superclassic Enabling Script

Do not try to do this while any `fb_smp_server` or `fb_inet_server` process is running.

The interactive script `changeMultiConnectMode.sh` is provided in the `/bin/` directory beneath the Firebird root directory. To complete the installation of the Superclassic server requires running this script, regardless of whether you installed the Classic package from the RPM kit or the tarball.

In the `$firebird-root$/bin` directory, run the script as follows and you will be prompted to choose the mode:

```
$ ./changeMultiConnectMode.sh
Which option would you like to choose: multi-(process|thread) [process] ?
```

- To select Superclassic, enter “m” (without the quotes)
- If Superclassic is already enabled and you are running this script to revert to Classic, enter “p” (without the quotes)

Configuration

Several parameters in the `firebird.conf` configuration file in Firebird's root directory will affect the way the server runs in Superclassic mode.

Pay particular attention to **DefaultDbCachePages**. Unlike Superserver, Superclassic does NOT provide a shared cache for all its process threads. Each thread has an individual cache so it will be important to allocate cache wisely, just as one would for Classic.

Parameters pertaining to locking and shared memory (names starting with **Lock--**) may also prove to be of interest as you test Superclassic under production conditions.



Restart the Superclassic server after changing parameters in `firebird.conf`.

Superserver

The Superserver binary is called *fbserver*. You can set it up to run with or without the Guardian, which is enabled on POSIX by supply the `-f[orever]` switch when starting the executable.

NPTL Implementation

The so-called “new” Native POSIX Thread Library (NPTL), that replaced *pthreads*, has been around for much of Firebird’s life. Buggy kernel implementations can cause problems for Firebird Superserver and locally-compiled programs in old versions of Red Hat (9.x and below) and possibly some later Linux distributions. The *gbak* utility was reported to throw a “broken pipe” error in these conditions. If you cannot avoid using the old or buggy kernel version, you will be limited to using a version of Firebird that supports the old *pthreads* threading model (search the Firebird download pages for a “non-NPTL” kit).

Follow these steps if you need to enforce the use of *pthreads* on a Linux that has a buggy implementation of NPTL:

- 1 Take care of the server instance. In `/etc/init.d/firebird`,

```
LD_ASSUME_KERNEL=2.2.5
export LD_ASSUME_KERNEL
```
- 2 You need to have the environment variable set up within the local environment as well, so add the following to `/etc/profile`, to ensure every user picks it up for the command line utilities. After

```
HISTSIZE=1000
```

```
add
```

```
LD_ASSUME_KERNEL=2.25
```

On the following line, export it:

```
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUT_RC LD_ASSUME_KERNEL
```



*Firebird 2.0.5 was the last build that supported the option to use the old *pthreads* threading model. V.2.0.6 and all successive “2 series” Firebird Superserver versions do not come with kits capable of running on systems where the NPTL is disabled.*

MacOSX/Darwin

The official kits for MacOSX are distributed as zipped .pkg (package) installer files that create a framework. To unzip the package contents, double click the file icon to unzip automatically. Using *unzip* is also an option.

From the operator’s point of view, all the kits install in the same way. Just double click on the package icon in the Finder window and the installer will take care of everything, including permissions, environment variables and so on.

Alternatively, run the *installer* utility as from the command line, e.g.

```
sudo installer -verbose -dumplog -pkg nameofpkg -target /
```

The root for Firebird (any model) will be `/Library/Frameworks/Firebird.framework/`.



In order for multi-threading conditions to work properly, the Firebird 2.5 engine uses Grand Central Dispatch, which was first released in MacOSX 10.6 (Snow Leopard). If you want to use an earlier version of OSX you will need to use an earlier version of Firebird.

Classic or Superclassic

Classic (*fb_inet_server* in Classic mode) runs by way of a daemon—*launchd* if the MacOSX version is 10.5 or higher, *inetd* or *xinetd* on lower versions. Ensure that you have the appropriate daemon installed and running. Once the framework is in place, simply connecting to a database will start an instance of *fb_inet_server* for the requesting client.

Do not try to install the Guardian with a Classic server..

Enabling Superclassic

Superclassic is *fb_inet_server* in threaded mode. On this platform, as with other POSIX, it uses a binary named *fb_smp_server*. It does not use *launchd*/[*x*]*inetd* but runs as a self-contained “listener” daemon itself, that is started by way of *StartupItems*.

To convert from *fb_inet_server* to *fb_smp_server*, change to `/Library/Frameworks/Firebird.framework/Resources/bin/` and find the script named `ChangeMultiConnectionMode.sh`. Run this script. Your *fb_smp_server* daemon will start listening for connections.

Superserver

The installation procedure for Superserver is the same as for Classic.

SuperServer (*fbserver*) starts via *StartupItems*. If the **-forever** option is used, the Guardian (*fbguard*) will start first and kick off the *fbserver* process.

The Start and Stop commands are the same as for Superclassic.

“lipo” Packages

The “lipo” packages provide both 32-bit and 64-bit client libraries (“fat-libs”), useful if you want to be able to use a 32-bit client—such as the open source Flamerobin database management toolset—locally with a 64-bit server.

Install a “lipo” package in just the same way as before. The installation will be the same as its regular 64-bit counterpart except that your Mac will accept either 64-bit or 32-bit client requests in accord with the requesting application.

Other Host Platforms

If you are installing Firebird on a POSIX platform that is not covered here, be sure to study any README and other information texts that are in the root of the tarball or in the documentation (`/doc/`) directory of the installation.

Testing Your Installation

If everything works as designed, the Firebird server process will be running on your server when the installation finishes. You can run some tests to verify the installation and work out any adjustments you might need in your configuration.

Network Protocol

At this point, it is assumed that you will use the recommended TCP/IP protocol for your Firebird client/server network, to take advantage of the benefits of platform-independent networking.



In an all-Windows environment, you also have the alternative of using Named Pipes protocol (wrongly referred to as “NetBEUI” in Delphi programming environments), although it is not recommended except possibly within a very small office LAN. For more information, refer to the [Network Protocols](#) topic at the beginning of the next chapter.

Pinging the Server

Usually, the first thing you will want to do once installation is complete is ping the server. This just gives you a reality check, to ensure that your client machine is able to see the host machine in your network. For example, if your server’s IP address in the domain that is visible to your client is 192.13.14.1, go to a command shell and type the following command:

```
ping 192.13.14.1
```

Substitute this example IP address for the IP address that your server is broadcasting.



*If you get a timeout message, refer the next chapter, **Network Setup and Initial Configuration**, for further instructions. If you need more information about how to set up or find out your server’s IP address, see the topic [A Network Address for the Server](#) in the next chapter.*

TCP/IP Local Loopback

If you are connecting to the server from a local client—that is, a client running on the same machine as the server—you can ping the virtual TCP/IP loopback server:

```
ping localhost
```

or

```
ping 127.0.0.1
```

Checking That the Firebird Server Is Running

Techniques for checking whether the server is running vary according to the operating system and Firebird model.

Classic Server on POSIX, including MacOSX

Use the **ps** command in a command shell to inspect the running processes. If any clients are connected to a Firebird Classic process, you should see one process named `fb_inet_server` for each connected client. The **ps** command has several switches, but the following will provide a satisfactory list. The **grep** command will filter the output so you only see the Firebird processes. Look for processes named `fb_inet_server`:

On Linux:

```
[xxx]$ ps aux | grep fb
```

On MacOSX:

```
ps ax | grep fb
```

V.1.0.x The “fb” prefix belongs to Firebird versions 1.5 and higher, while “gds” and “ib” belong to Firebird 1.0.x. If you are running v.1.0.x use `ps aux | grep gds`.

Superserver on POSIX, including MacOSX

Because Superserver forks off a thread for each connection, it is interesting to throw the `-f[ork]` switch into the mix for examining its processes and threads. You get a formatted display of the forking processes.

```
[xxx]$ ps auxf | grep fb
```

The same `ps` command should show one process named *fbguard* if the server was started with the `-f[orever]` switch, and one main process named *fbserver*. There will be at least one child process thread named *fbserver* forking off one more such process thread. This first group is “the running server,” sans any client connections except those that the server uses for listening on ports and for garbage collection. Beyond that will be a group of threads for each connection.

Windows Server Platforms

For the Windows server platforms, start the Firebird Server Control applet from the Control Panel.

Server Control Applet

Figure 2.1 shows the Firebird Server Control applet display on a Windows Server. If you used the installer, this applet will have been installed to your Control Panel. Its appearance may vary from one Windows server edition to another.

Figure 2.1 Server Control applet



You can use the applet to start and stop the service and to modify the start and run options. It is not recommended to change to “Run as an application” for multi-user use, for security reasons—you have to leave the server logged in to keep the server running.

Services Applet

If you have no Control Panel applet, you can inspect the Services applet (see Figure 2.2) in the Administration Tools display. On most Windows platforms, you can access this applet from the Control Panel, under Administrative Tools.

Figure 2.2 Services applet on Windows server platforms

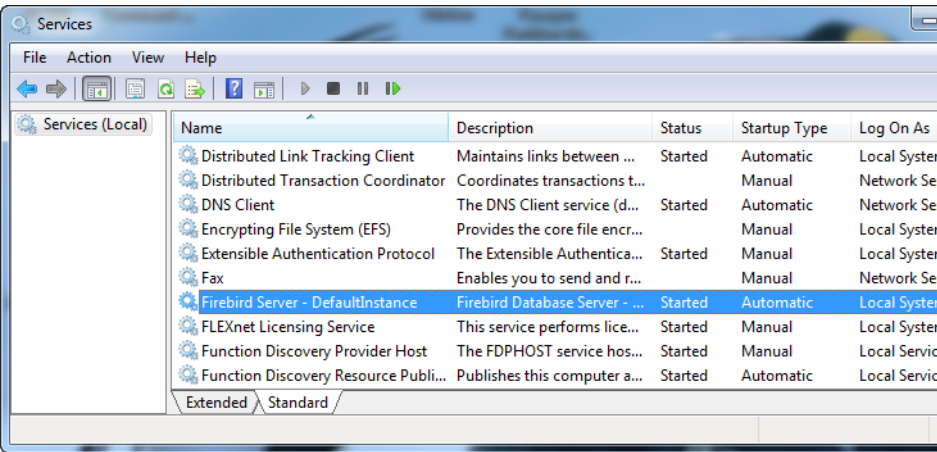


Figure 2.2 shows the service names for the Guardian and the Superserver. They may have different service names because of version changes; the Guardian may not appear at all. A user with Administrator privileges can right-click the service name to stop or restart the service. If you are using the Guardian, stop that service to stop both Guardian and server.

On the Windows server platforms, the Guardian is a convenience rather than a necessity, since these operating systems have the facility to watch and restart services.

Windows 9x, ME, and XP Home

Windows 9x and ME do not support services. Firebird server should be running as an application, monitored by the Guardian. If you used an installation kit that installed but did not automatically start the Guardian and the Firebird server, you can set it up manually, as follows:

- 1 Locate the executable file for the Guardian program (ibguard.exe) and create a shortcut for it in the Startup area of your machine's Start menu.
- 2 Open the Properties dialog box of the shortcut and go to the field where the command line is.
- 3 Edit the command line so it reads as follows:
`fbguard.exe -a`
- 4 Save and close the Properties dialog box.
- 5 Double-click the shortcut to start the Guardian. The Guardian will proceed to start *fbserver.exe*.

The Guardian should start up automatically the next time you boot your machine.

The Client Libraries

All installation kits are distributed with the client libraries included. Since the principal use of a client library is to provide the programming interface between remote client applications and the server, its presence is required as a component of any remote client application.

Of course, some applications live on the server. A copy of one or more of the client libraries native to the server platform is installed in an appropriate location for use by server-local applications such as the command-line tools, database management or monitoring applications and server-based components of n-tier applications.

Windows

On Windows, the client is named `fbclient.dll` and it is installed by default into the `\bin\directory` beneath the Firebird root directory. By default, the command-line utilities load the client from there, not the System directory.

The installer gives you the option of placing `fbclient.dll` in the system directory, if you need it there for compatibility with application code.

An option exists also to generate a version of the client library that is named `gds32.dll` and contains an identity string that will make the Embarcadero “InterBase Express” components recognise it.

V.1.0.x For Firebird 1.0.x, the name of the client library is `gds32.dll` and it is installed into the system directory. Command-line utilities load the client from there.

MacOSX/Darwin

On MacOSX/Darwin the client libraries are `libfbclient.dylib` (the remote client) and `libfbembed.dylib` (the embedded server, used for direct access to databases).

After installation, the embedded server library—which works only on the host machine—moves into `/Library/Frameworks/Firebird.framework/Firebird Classic`, while the remote client will be in `/Library/Frameworks/Firebird.framework/Libraries`.

Performing a Client-Only Install

Installing remote clients is an essential part of deploying your database applications in a client/server network.

Each remote client machine needs to have the client library that matches the release version of the Firebird server it will attach to. In general, it will be safe to use a client library from a different build of a release, as long as the version numbers match. However, when upgrading the server, do read any readme documents that come with a point release, to determine whether any “gotchas” exist regarding lower client versions.

Search carefully in the system path of any client workstation on which you want to deploy a Firebird client, to find and, if necessary, disable existing client installations for InterBase® or older Firebird versions.



Newer client versions should be able to connect to databases running under an old server. Firebird 1.0.x and 1.5 client applications usually have no problem with legacy applications written for Interbase databases running under the very ancient InterBase 6.0 or lower. The exception is applications built with Embarcadero's InterBaseXpress (IBX) components, which are hard-wired to look for a certain version string in the file. All of the

Firebird installers, along with the executable `instclient.exe`, provide the option to generate a Firebird client named `gds32.dll` that has a version string that IBX likes.

Firebird 1.5 and later versions on Windows can co-exist with InterBase® or Firebird 1.0.x on both server and client. It is still a matter of setting things up manually, however, and tools are available to assist with the task.

Installing a Linux/UNIX client

POSIX operating system layouts are famously idiosyncratic. The suggestions below should be helpful as a guide to installing clients on many common Linux and UNIX flavors but this is an area where uncertainty is the only certainty!

Log into the client machine as root and look for the client library in the server installation.

The binary for remote clients is `libfbclient.so.n.n.n`, where “n’s” are numbers corresponding to the major and minor release numbers of the server. The RPM and tarball scripts install it by default in `/opt/firebird/lib` and symlink it (see below).

v.1.0.x The client libraries for Firebird 1.0.x inherit from the InterBase ancestor the confusing habit of distributing both the local client (the embedded version, that comes with the Classic package) and the remote client, that comes with the Superserver package, with the same name, `libgds.so.n.n.n`, where the “n’s” might be a variety of numbers. Installation symlinks either client with `libgds.so.0`. The default location is `/usr/lib`.

Step 1: Symbolic linking

Copy the library to `/usr/lib` on the client and create symbolic links for it, using the following commands:

```
ln -s /usr/lib/libfbclient.so.1.5 /usr/lib/libfbclient.so.0
ln -s /usr/lib/libfbclient.so.0 /usr/lib/libfbclient.so
```

v.1.0.x: `ln -s /usr/lib/libgds.so.0 /usr/lib/libgds.so`



The embedded library, `libfbembed.so`, has the capability to connect to a remote server, as well as to the embedded server. The client component in it will work just as the regular `libfbclient.so` does and will connect client applications to databases on either Classic/Superclassic or Superserver servers.

Step 2: Position other Firebird components

Create a directory `/opt/firebird` on the client for the message file (`firebird.msg`) and copy the file into it from the firebird root on server to the client.

v.1.0.x Create `/opt/interbase` on the client and copy the file `interbase.msg` into it.

Step 3: Set the FIREBIRD environment variable

In system-wide default shell profile, or using `setenv()` from a shell, create the environment variable that will enable the API routines to find the messages:

```
setenv FIREBIRD "/opt/firebird/"
```

v.1.0.x `setenv INTERBASE "/opt/interbase/"`

Installing a Windows client

You can install a Windows client using the same installer that you use for installing a server or you can extract the files from the matching ZIP kit and install the client manually. The

latter method may suit you if you want to script a client install for use on multiple workstations.

The recommended way to install a client is to use the installer program.



Although you are not going to install the server, allowing the installer to install to a root location will ensure that optional pieces—including the Registry key—that are needed by some software products are available on the client machine. Later, you can customize the setup manually, if necessary.

If you are installing the command-line tools for remote use, a properly installed root location is essential.

Choosing a root location for the client components

The first choice you need to make is where the root of the client installation is to be located. It is recommended that you take the default (c:\Program Files\Firebird\Firebird_n_n), since it will simplify future upgrades. The “n_n” part here should be replaced by the “major version” numbers of your server version, e.g. “_2_5” if your server is v.2.5.

However, you can specify a custom location if necessary.

Microsoft run-time libraries

The client libraries, like the server, are compiled on a specific version of Microsoft Visual Studio. This means that, also like the server, they need to have the correct Microsoft C and C++ run-time libraries available. Newer versions of the runtimes should work satisfactorily with older Firebird binaries but, sadly, it does not always work that way. To review the information about the runtimes, see **The Microsoft C++ Run-times** on [page 31](#).

Using the Firebird Installer

The simplest way to install a Firebird Windows client is to copy the Firebird installer to a portable storage device and run it on the client machine, selecting a “Client-only” install option when prompted by the installer dialog. You can choose whether to install the client with or without the command-line tools.



Most clients will not need the tools and it is not recommended to install them on client workstations that do not need admin access to the server.

The installer will create the default root directory in c:\Program Files\Firebird, which you can vary when it prompts for a location. Here, it will write the message file—firebird.msg—and, if you selected to install the tools, it will create a \bin\ directory beneath the root and install the tools there.

If you have asked it to, it writes fbclient.dll or gds32.dll to the system directory, provided it is valid to do so for your Windows version. If the Microsoft C/C++ run-time libraries are too old, or absent, it may write the correct ones there also.

Finally, it runs a program named instreg.exe to install the Registry key, whose value is the absolute path to where firebird.msg is. If you chose the default installation directory, the key is HKLM\Software\Firebird Project\Instances. If any running DLLs were overwritten during the installation, you will be prompted to reboot the machine.

v.1.0.x The old InterBase client library was always installed by default in the system directory, c:\windows\system32 on WindowsXP and Server 2003, for example. Firebird 1.0.x followed suit, retaining the old names and locations. The library is gds32.dll.

Installing a client manually

Installing a client manually requires all of the above steps.

You will need to copy the following files from your server installation, or extract them from the ZIP kit, to portable storage or a shared folder:

```
firebird.msg
bin\fbclient.dll
bin\{Microsoft Visual C run-time DLL} (if needed)
bin\{Microsoft Visual C++ run-time DLL} (if needed)
bin\{Manifest file for the local assembly} (if present and if needed)
bin\instreg.exe
bin\instclient.exe
```



For Firebird 2.1.x and 2.5.x clients, the whole Microsoft Visual C/C++ run-time assembly comprises msvc80.dll, msvcp80.dll and Microsoft.VC80.CRT.manifest.

For older versions, check the names and versions of the run-times that are present in the \bin\ folder of the server install structure for that version of the server. In the v.2.1 and 2.0 kits, you may find there is a “.msi” redistributable installer present.

Once you have created your Firebird “root” directory on the client machine, copy firebird.msg there.

Create a \bin\ folder beneath the “root” directory and copy into it all of the files from the \bin\ folder on your storage device.

In a command window, change to the \bin\ folder that you created and run instreg.exe from there:

```
c:\Program Files\Firebird\Firebird_2_5\bin>instreg.exe
```

Alternatively, you could run instreg.exe from your portable storage device, e.g., from a flash drive assigned as drive K:

```
K:> instreg.exe 'C:\Program Files\Firebird\Firebird_2_5'
```

If you placed the root directory for your Firebird client installation somewhere different, use that path as the root directory argument.

Running instclient.exe

The program instclient.exe can be run when you need a client version that can be accessed and used by existing software, drivers or components that expect the client to be named gds32.dll and/or to be located in the Windows system path.

Still in the command window, from the \bin\ location, run the command according to the following syntax:

```
instclient.exe {i[nstall]} [-f[orce]] {fbclient | gds32}
```

The parameters **i** (or **install**) and either **fbclient** or **gds32** are required.

If the program finds there is already file in the system directory with the name of the file you are trying to install (fbclient.dll or gds32.dll) it will not proceed. To have the program write the file even if it finds a pre-existing copy, use the **-f** (or **-force**) switch.

Your operating system may require you to reboot the machine to complete the installation.

Querying the installed client

The `instclient.exe` program can be used for querying about Firebird running on the machine. The syntax for querying about the client is

```
instclient.exe {q[query] fbclient | gds32}
```

For example,

```
instclient.exe q fbclient
```

returns

```
Installed FBCLIENT.DLL version : 2.5.0.26074 (shared DLL count 1)
```

Using `instclient.exe` to uninstall a client DLL

To remove a Firebird client that is installed in the system directory, use the following syntax:

```
instclient.exe {r[emove] fbclient | gds32}
```

For example, the following command removes the client reported in the query example above:

```
instclient.exe r fbclient
```

Installing an Embedded Server

Embedded Server on Windows

If you have not used Firebird before, it is strongly recommended that you bypass this option until you have gained some experience of working with one of the server versions and the “regular” client. You will not lose anything by attempting your first applications under the normal client/server model; they will work just fine with the embedded server.

Download Kit

The files for the embedded server are distributed in a separate kit from the server models. Look for downloads with the infix “embed”.

V.1.0.x There is no embedded server version for v.1.0.x.

The merged server and client are in the dynamic link library `fbembed.dll`, which you will find in the `\bin\` directory of your regular Firebird server installation. You can install an embedded server if you already have a full server or other embedded servers installed.

For each embedded server application, the home directory of your application executable becomes the root directory for that embedded server application. To set up an embedded Firebird server installation with your application, do as follows:

- Copy `fbembed.dll` to the home directory and rename it to either `fbclient.dll` or `gds32.dll`, according to the client file name your database connectivity software expects.
- Copy the files `firebird.msg` and `firebird.conf` to the home directory.
- If you want to use the database aliasing feature (recommended), copy `aliases.conf` to the home directory and configure it for any databases this particular application will connect to.
- If external libraries are required for your application, such as international language support (`fbintl.dll` and the `icu*.dll` files), UDF libraries, or blob filter libraries,

create folders for them (./intl, ../UDF) directly beneath your application home directory and copy them to these locations.

- If you want any of the command-line tools available to the application user, place the required executables in the home directory, alongside your own executable.

Example of an Embedded Installation Structure

The following is an example of the directory structure and configuration for an installed embedded server application:

```
home = D:\my_app
D:\my_app\MyApp.exe
D:\my_app\fbclient.dll
D:\my_app\firebird.conf
D:\my_app\aliases.conf
D:\my_app\firebird.msg
D:\my_app\firebird.conf
D:\my_app\gbak.exe
D:\my_app\intl\fbintl.dll
D:\my_app\UDF\fbudf.dll
MyDatabase = D:\databases\MyDB.fdb
```

Uninstalling Firebird

As with installation, uninstalling Firebird is specific to the operating system platform and the integration of the original installation with the software inventory tools of the platform.

Linux

If you need to uninstall Firebird, you must do so as root. How you go about it depends on how it was installed, viz.

- from a RPM kit
- from a tarball
- by a distribution-specific method

The following examples use the context of a Classic installation, but the same holds true for Superserver by replacing the ‘CS’ in the package name with ‘SS’.

Uninstalling a RPM package

As root, query the list of installed RPM packages to verify the correct name of the package that was installed, if indeed the installation was performed by RPM:

```
rpm -qa Firebird
```

Once you have verified the correct package name, e.g., FirebirdCS-2.5.0.26074-0, use the -e switch to remove (erase) the package:

```
$rpm -e FirebirdCS-2.5.0.26074-0
```



If the RPM query does not return any package names, that should tell you that Firebird was installed on your server by another means.

Uninstalling a tarball installation

The tarball installations come with an uninstall script in the `/bin/` directory. As root, simply do

```
$/opt/firebird/bin/uninstall.sh
```

Check any symlinks that you might have created yourself and remove them.

Uninstalling a distribution-specific installation

If you installed Firebird from a repository for your Linux distribution using `apt` or some like tool, use the same tool to uninstall it.

Windows

When Firebird was installed on your Windows host, it would have been done either by running the installer executable or by unzipping the zip kit and completing the installation manually.

Uninstalling Using Windows Native Tools

If you used the installer program to install Firebird there should be a record of it in the Registry and it will appear in the pick list when you run Add/Remove Programs applet from the Control Panel.

First, get all connections off-line, if possible, before shutting down each database that is potentially being served by the Firebird server that you want to uninstall.

Next, shut down the Guardian (if running) and Firebird services from the Services applet in the Administrative Tools section of the Control Panel. If the server and/or Guardian are running as applications, use the Firebird Server Manager Control Panel applet to stop them, if it is available.

Now, use the Add/Remove Programs applet to remove Firebird.

What is left after an uninstall

The uninstall routine preserves some files that you may need, in case you intend to install an upgrade or to reinstall on another machine:

```
security.fdb or security2.fdb (depends on version)
firebird.log
firebird.conf
aliases.conf
```

Any files that were added to the server's directory structure after the original installation are left untouched.

Shared files such as `fbclient.dll` (or `gds32.dll`) and the `icu*` libraries are deleted only if the share count would become 0 by their removal.

The Registry keys are removed.

Uninstalling a zip kit install

To remove Firebird without a Windows Uninstaller profile in the Registry database, do the following steps in order:

- 1 get all users off-line if possible and shut down all databases
- 2 stop the server as described in the previous section
- 3 open a command window and cd to the Firebird installation's \bin\ folder, then run
 - a instreg.exe remove
 - b instsvc.exe remove
 - c instclient.exe remove fbclient.dll
 - d instclient.exe remove gds32.dll

Finally, backup any files you want to preserve and delete the Firebird installation directory.



Do not delete the client libraries from system locations, such as the system32 directory, manually, because it is likely to make the system's shared library count inaccurate. An important task of instclient.exe is to enable the installation and removal of the client library into and from system locations without breaking the shared library count.

MacOSX/Darwin

To remove Firebird from a Mac system cleanly, use the following shell script:

```
#!/bin/sh
echo "Clean User"
dscl localhost -delete /Local/Default/Users/firebird echo "Clean Group"
dscl localhost -delete /Local/Default/Groups/firebird if [ -f
"/Library/StartupItems/Firebird" ]; then echo "Remove Superserver StartupItem"
rm -fr /Library/StartupItems/Firebird
fi
if [ -f "/Library/LaunchDaemons/org.firebird.gds.plist" ]; then echo "Remove Launchd"
launchctl unload /Library/LaunchDaemons/org.firebird.gds.plist rm
/Library/LaunchDaemons/org.firebird.gds.plist fi
echo "Remove Framework"
rm -fr /Library/Frameworks/Firebird.framework echo "Remove Receipt"
rm -fr /Library/Receipts/Firebird*.pkg
echo "Remove /tmp/firebird"
rm -fr /tmp/firebird
```

Other Things You Need to Know

We finish this chapter with a few items that you will need to know if you are new to Firebird.

Default User Name and Password

The SYSDBA user has all privileges on the server. The installation program will install the SYSDBA user into the security database (security2.fdb, or security.fdb if you are installing Firebird 1.5).

V.1.0.x In Firebird 1.0, the security database is named `isc4.gdb`.

On Windows and in the v.1.0.x Linux versions, the password is **masterkey**. Actually, the password is **masterke**, since all characters after the eighth one are ignored.

On the v.1.5 and higher Linux versions, the installer generates a random password during installation, sets it in the security database, and stores it in clear in the text file `SYSDBA.password`. Either memorize it or use it to get access to the security database and change it to something easier to remember.



If your server is exposed to the Internet at all, you should change this password immediately.

How to Change the SYSDBA Password

If you are on a Linux or other system that can run a bash script, cd to the `/bin/` directory of your installation and find the script named `changeDBAPassword.sh`. All you need to do is run this script and respond to the prompts. The first time you run the script, you will need to enter the password that the installer wrote in `SYSDBA.password`—it is in the Firebird root directory:

```
[bin]# sh changeDBAPassword.sh
```

or

```
[bin]# ./changeDBAPassword.sh
```

Using *gsec* directly

The command-line and shell utility *gsec* exists solely for maintaining login credentials for Firebird users. The following procedure will work on both Windows and Linux.

On Linux, you need to be logged into the operating system as Superuser (root) to run *gsec*. Let's say you decide to change the SYSDBA password from **masterkey** to **icuryy4me** (although, on Linux, the installed password will not be “masterkey”, but something much more obscure!). You would need to follow these steps to do so:

- 1 Go to a command shell on your server and change to the directory where the command-line utilities are located. Refer to Appendix X to find this location.

- 2 Type the following on Windows, treating it as case-sensitive:

```
gsec -user sysdba -password masterkey
```

On POSIX platforms type as follows:

```
./gsec -user sysdba -password masterkey ()
```

You should now see the shell prompt for the *gsec* utility:

```
GSEC>
```

- 3 Type this command:

```
GSEC> modify sysdba -pw icuryy4me
```

- 4 Press Enter. The new password **icuryy4m** (note, 8 characters!) is now encrypted and saved, and **masterke[y]** is no longer valid.

- 5 Now quit the *gsec* shell:

```
GSEC> quit
```

Because Firebird ignores all characters in a password past the eighth character, `icuryy4m` will work, as will `icuryy4monkeys`.

Full instructions for using *gsec* are in Chapter 36, *Protecting the Server and its Environment*.

Linux/UNIX Users and Groups

From Firebird 1.5, the **firebird** user is the default user that runs the server software. This means you need to put non-root users into the **firebird** group to enable them to access databases using the embedded model.

To add a user (for example, “sparky”) to the firebird group, the root user needs to enter:

```
$ usermod -G firebird sparky
```

The next time sparky logs on, s/he can start working with Firebird databases.

To list the groups that a user belongs to, type the following at the command line:

```
$ groups
```



*The **firebird** user will also need read-write privileges to all databases and read-write-execute privileges to all directories where databases are located.*

Admin Tools

The Firebird kits do not come with GUI admin tools. The excellent third-party GUI tools available for use with a Windows client machine are too numerous to describe here.

Readers are recommended to look at the cross-platform, open source GUI toolset named FlameRobin, at its project website <http://www.flamerobin.org>.

Firebird does have a set of multi-platform command-line tools, executable programs that are located in the `/bin/` directory of your Firebird installation. They are summarised in Chapter 4, **Operating Basics**, in *Summary of Command-line Tools*, where you will find links to detailed usage instructions in later chapters.

A list of the better-known third-party database administration tools for Firebird appears in Appendix 14, **Resources**, in *Third-Party Tools*. For up-to-date and more comprehensive listings, visit <http://www.ibphoenix.com>, select the Contributed link from the Downloads area, and then choose the Administration Tools link.



Because of the heterogenous nature of Firebird, you can use a Windows admin client to access a Linux or MacOSX server and all permutations of mixed platforms.

NETWORK SETUP AND INITIAL CONFIGURATION

As a relational database management system (RDBMS) purposely built for client/server deployment, Firebird is designed to allow remote and local clients to connect concurrently to the server using a small range of network protocols.

The default installers will set up a default TCP/IP configuration for connecting the server-based client to your server and for receiving connections from clients using a default TCP port. Unless there is some external reason to use a custom network configuration, it should not be necessary to configure anything in order to get your first installation of Firebird up and running.

Network Protocols

Firebird supports TCP/IP as the protocol of choice for client/server communications. Support for Novell was dropped early in Firebird's life, reflecting lack of demand for it. It continues to support the deprecated "Named Pipes" transport on Windows, although it will go the same way as Novell eventually.

The Firebird API, or its implementation in driver layers, deduces the protocol to use by interpreting the connection string passed to it by the client application.

TCP/IP

Firebird supports TCP/IP for all combinations of client and server platforms. The default port is 3050. In the event that the host network is already using port 3050, the Firebird server, the client application, or both, can be configured to use another port.



Make sure that port 3050 (or whatever port Firebird is configured to listen on) is opened with the appropriate sub-net scope in the firewall.

Named Pipes

Firebird supports the deprecated Microsoft WNet Named Pipes protocol for Windows server platforms and Windows clients. The default pipe name is `interbas`.

As noted earlier, this protocol is often wrongly referred to as “NetBEUI”, especially by the documentation and property sheets for Delphi driver products. NetBEUI is actually not a protocol but an old transport layer used for networking in Windows 3.1 and 9X.



Windows 9x and ME do not have the capability to be WNet servers.

Local Access

Although Firebird is designed to be a database server for remote clients, it offers a number of options for local access.

Client/server

TCP/IP local loopback: For n-tier server application layers and other clients accessing a local server on any supported platform using TCP/IP, even without a network card, connection can be made through the special localhost server at IP address 127.0.0.1.

The localhost connection is not possible for embedded server applications.

“Windows local” connection mode: For Windows clients running the Firebird Superserver on the same physical machine, Firebird supports a local connection mode involving inter-process communication to simulate a network connection without a physical network interface or wire protocol. It is useful for access to a database during development, for embedded server applications and for console-tool clients, but it does not support Firebird's event mechanism.

- From the “2” series onward, the subsystem that provides this “Windows local connect” is XNET. The name of the shared memory area used for this transport channel is FIREBIRD. If the host operating system is Windows 7 or Vista, or you are running Windows Server 2003 or XP with terminal services enabled, you will probably need to configure this namespace explicitly to get “Windows local” to work. The entry in `firebird.conf` is ***IpcName*** and you should change it to `Global\FIREBIRD`



The prefix Global is case-sensitive.

- In versions 1.X, it is a less functional transport channel referred to as “IPServer” or “IPCServer”. XNET and IPCServer are not compatible.

In Firebird 1.5.x, the default value of ***IpcName*** is `FirebirdIPI`. In Firebird 1.0.x, it is `InterbaseIPI`.

Direct local connect on POSIX: Whether a local client can connect to a database on Linux and some other POSIX systems depends primarily on the server mode you have installed (Classic or Superserver) and, secondarily, on the type of client connection.

- Superserver on POSIX does not accept local connections through the normal API clients at all. The connection path must always include the TCP/IP host name. However, it does accept local connections from “embedded applications”, i.e. applications written using embedded SQL (ESQL). The command-line tools, *gsec*, *gfix*, *gbak* and *gstat*, which are embedded applications, can make local connections to a Superserver.
- If you are running the Classic server, direct connection is possible from a local client.

Embedded server

On Linux, the client library *libfbembed.so* that is a component of a Classic installation is a fully functional embedded server (client and server merged as a dynamic library) that dates back to pre-Firebird times. A client application running on the host machine in this environment can connect directly to a database. The client/server connection is entirely self-contained: no network resources are involved and no daemon intercedes in the client attachment. Normal server-level authentication applies.

From Firebird 1.5 onward, the Firebird Windows offerings have included a functionally similar embedded library, named *fbembed.dll*. For application development, its features are identical to those of the normal client/server model except that there is no network protocol support: connection must be via the “Windows local” style of emulated network connection.



On Windows, connections between the embedded client and the embedded server bypass server authentication.

Mixed Platforms

Firebird’s design enables clients running one operating system to access a Firebird server that is running on a different platform and operating system to the client. A common arrangement, for example, is to run several inexpensive Windows workstations concurrently accessing a departmental server running on a host that might be Windows, Linux, or any of several brands of UNIX.

A database that was built for access by one server model can be served by any of them. For example, when an embedded server application is off-line, its database can be under the control of a different embedded server application or a full Firebird server serving remote clients.

Porting a Database

The same database can be ported from platform to platform without modification. It is strongly recommended that the database be backed up with *gbak* on the old platform and restored with *gbak* on the new. File-copying databases is not recommended, especially with databases that were created using a Firebird version lower than 2.0.x and still have an old on-disk structure (“ODS”).

ODS 11.1 and 11.2 databases should be completely portable between Intel platforms, even when porting from i86 to i64 operating systems. Still, if a *gbak* backup/restore is practicable for such migrations, it should be in the plan. If not, be certain to test and verify copies thoroughly.



Never file-copy a database that has active connections. This warning applies, even if the copy is not going to be used. File-copying applications can be aggressive about overriding locks that databases rely on for data integrity.

Besides the Windows and Linux platforms, current server implementations of Firebird (Classic, Superclassic, Superserver) are available also for Mac OSX (Darwin). Older versions of Classic and, in some cases, Superserver, are available for FreeBSD, Sun Solaris (Intel and Sparc), HP-UX and potentially AIX, and can be built for a number of additional UNIX platforms.

Old Netware

Firebird does not currently have platform support for any version of Netware by Novell, nor for any other species of networking that accords to the largely obsolete IPX/SPX protocol. With the demise of Novell support for this protocol, sites often operate Firebird server on a Linux system, with network clients connecting to this subsystem via TCP/IP.

A Network Address for the Server

The host on which the Firebird server is running needs a network address in order for network clients to connect to it.

- If you are on a managed network, get the server's IP address from your system administrator
- If you have a simple network of two machines linked by a crossover cable, or a small, switched network, you can set up your server with any suitable unique IP address you like except 127.0.0.1 (which is reserved for a local loopback server)
- If you know the “native” IP addresses of your network cards, and they are different, you can simply use one of those that the remote Firebird client applications can “see”
- If you know the TCP/IP host name of the server, you can use that. On POSIX, the host name must be resolved in `/etc/hosts`. Note that the network name of a Windows server will resolve automatically to both its TCP/IP host name and its NetBIOS destination name.
- If you are intending to try out a single-machine installation of both client and server, you should use the local loopback server address—localhost, or its IP address 127.0.0.1



It is possible for a single user to connect locally to a server, without using a local TCP/IP loopback, either as an or embedded client or, on Windows, as an ipServer client (“Windows local” protocol). A server address is neither required nor valid for such a connection and cannot be used to verify that TCP/IP is working properly in your setup.

Hosts File

When setting up Firebird nodes for TCP/IP networking, it is recommended that you configure the host name files on clients and use these, rather than the IP addresses directly, for attaching to the server. Whilst most up-to-date operating systems can use the host's IP address in your client connection string in lieu of the host name, connection through a host name ensures that the server's address remains static, regardless of dynamic address changes in the network.

If your TCP/IP network is not running a domain name service (DNS), or your network's dynamic host addressing is sometimes unreliable, it may be advisable to inform each node individually of the mappings of IP addresses to host names in your network. To enable this, update the HOSTS file of each node (server and client).

Locating the HOSTS file

on Linux and many UNIX versions, the hosts file is normally located in `/etc/`. Note that a file name is case-sensitive on the Unix family of platforms.

- on Windows server platforms the HOSTS file is located in `c:\windows\system32\drivers\etc\` (or `c:\winnt\system32\drivers\etc\` on Windows 2000 or a host that was upgraded from Windows 2000)
- on Windows 95/98/ME it is located in `c:\windows\`

If HOSTS is not present, a file named `Hosts.SAM` will be found in the same location. Copy this file and rename it to `HOSTS`. On Windows, a file name is not case-sensitive.

Examples of hosts file entries

```
10.12.13.2 db_server # Firebird server (in your LAN)
216.34.181.60 sourceforge.net # (server on a WAN)
127.0.0.1 localhost # local loopback server (on Windows)
127.0.0.1 localhost.localdomain # local loopback server (on Linux)
```

Open and edit the hosts file with a plain text editor.

- The IP address, if not 127.0.0.1 (localhost) must be the valid one configured for the host in your network.
- `Server_name` can be any unique name in your network.
- The comments following the hash (`#`) symbol are optional, but recommended
- The text format is identical, regardless of whether the host is running on Windows or a POSIX operating system. However, if you are copying a hosts file around a mixed network, be aware of the differences in end-of-line characters used on Windows, MacOSX and Linux/other POSIX.

After editing, check that your editor saves `HOSTS` with no file extension. If necessary, remove any extension by renaming the file.



On Windows 95 and early Windows 98 the network support does not recognize an IP address in the connection string at all because they were installed with Winsock 1. Installing Winsock 2 is effectively a requirement for current Firebird versions. Free upgrade packs may still be available from Microsoft customer support sites.

Server Name and Database Path

When you create or relocate a database, ensure that it is on a hard disk that is physically attached to your server machine. Database files located on shares, mapped drives or (on Unix) mounted as SMB (Samba) or NFS filesystems can not be seen by the server.



SAN (storage area network) devices and NAS/iSCSI boxes (network-attached storage with iSCSI control) are seen as locally attached.

From Firebird 1.5 forward, you have the option of storing the paths to databases on your server as database path aliases. This not only simplifies the business of setting up portable connection strings in your applications, but adds an extra layer of security to your remote

1. Unless the database is read-only and *RemoteFileOpenAbility* is configured “on” in `firebird.conf`. For normal, read-write databases, here be dragons!

connections. Database path aliasing prevents sniffer programs determining where your databases are located and using that information to compromise your files.
Refer to *Database aliasing* in Chapter 4, *Operating Basics*, for information.

Connection string syntax

These are the “in-clear” connection strings for each platform, which you need for configuring aliases and for attaching clients to databases running on Firebird 1.0, which does not support database aliasing.

TCP/IP

A TCP/IP connection string has two elements: the server name and the absolute disk/filesystem path as seen from the server. It may optionally include a third element, the TCP port number, only necessary if the server is listening on a port other than 3050. Its format is as follows:

For connecting to a Linux server

hostname:/filesystem-path/database-file
or, with the optional TCP port number included:
hostname/port-number:/filesystem-path/database-file

Examples Examples for connecting to a Linux or other Unix-family server named “hotchicken”:
hotchicken:/opt/firebird15/examples/LeisureStore.fdb
hotchicken/103050:/opt/firebird15/examples/LeisureStore.fdb
All file names are case-sensitive on these platforms.



For connecting to a Windows server

hostname:Drive:\filesystem-path\database-file
or, with the optional TCP port number included:
hostname/port-number:Drive:\filesystem-path\database-file

Examples Examples for connecting to a Windows server named “winserver”:
winserver:C:\Program Files\Firebird15\examples\LeisureStore.fdb
winserver/103050:C:\Program Files\Firebird15\examples\LeisureStore.fdb
Forward slashes are also valid on Windows:
winserver:C:/Program Files/Firebird15/examples/LeisureStore.fdb

Windows Networking (Named Pipes/WNet)

For connecting a remote client to a Windows server using the Named Pipes protocol, an UNC-style notation is used:

\\servername\filesystem-path\database-file

where **\\Servername** must be the properly-identified node name of the server machine on the Windows network, not a share or a mapped drive.

For example,
\\winserver\c:\databases\LeisureStore.fdb

Windows local connection

For connecting an embedded client or a local external client in Windows local mode:

`C:\Program Files\Firebird15\examples\LeisureStore.fdb`

Inconsistent connection strings for Windows connections

To protect databases from a long-standing bug in Windows, Superserver or Superclassic on Windows establishes an exclusive lock on the database file when the first client connection is activated.

The connection path bug

Windows will accept two forms of the absolute local path to a file: one (correct according to the DOS standard) being `Drive:\path-to-database` and the other, optionally omitting the backslash following the drive designator, `Drive:path-to-database`.

If the server were to receive two client connection requests, the first using the standard path form and the subsequent one using the optional form, it would treat the two attachments as though they were connected to two different databases. The effects of any concurrent DML operations would cause destructive corruption the database.

For connections to Superserver or Superclassic, the exclusive lock pre-empts the problem by requiring all connections to use the same path format as was used by the first connection.

Classic

The same solution cannot be applied to Classic server process because each connection works with its own instance of the server. Take care to ensure that your application always passes consistent path strings.



It is highly recommended that you use the database aliasing feature for all connections and also ensure that `aliases.conf` contains one and only one alias for each database. See the topic [Database aliasing](#) in the next chapter.

Testing Connections

All things being equal, the last reality check is to make sure your client machine is communicating with the server host. You can quickly test whether a TCP/IP client can reach the server, using the ping command in the command shell. Usage is:

```
ping server_name
```

substituting the name you entered in the `hosts` file, of course.

If the connection is good and everything is properly configured, you should see something like this:

```
Pinging hotchicken [10.10.0.2] with 32 bytes of data
reply from 10.10.0.2: bytes=32 time<10ms TTL=128
reply from 10.10.0.2: bytes=32 time<10ms TTL=128
reply from 10.10.0.2: bytes=32 time<10ms TTL=128
reply from 10.10.0.2: bytes=32 time<10ms TTL=128
```

Press Ctrl–C to stop the ping responses.

If ping fails

If you get something like

Bad IP address hotchicken

then your host name file entry for the `server_name` (in this example, *hotchicken*, may be missing or wrongly spelt. For example, all identifiers on Linux/UNIX are case sensitive. Another cause might be simply that your server machine's host name has not been configured.

If you see

Request timed out

it means that the IP address referred to in your host name file cannot be found in the subnet. Check that—

- there are no typos in the host name file entry, remembering that a Linux host name is case-sensitive
- the network cable is plugged in and the wires and contacts are free from damage and corrosion
- the network configuration allows you to route network traffic between the client and server in question. Subnet or firewall restrictions may be preventing the host server from receiving the ping from the client.

Another possibility is that the host machine may be firewalled against ICMP requests, in which case ping will be blocked.

Use the *netstat* tool

netstat (network statistics) is a command-line tool that displays both incoming and outgoing network connections, routing tables, and a number of network interface statistics. It is available on both POSIX and Windows.

A useful *netstat* call is *netstat -a* which shows all sockets from all connections. If the Firebird server is listening for connections, it will be listed with the local port address it is configured to listen on (3050 by default).

On Windows the output of *netstat -a* might look something like this:

Active Connections

Proto	Local Address	Foreign Address	State
TCP	dev1:http	dev1:0	LISTENING
TCP	dev1:epmap	dev1:0	LISTENING
TCP	dev1:https	dev1:0	LISTENING
TCP	dev1:microsoft-ds	dev1:0	LISTENING
TCP	dev1:2869	dev1:0	LISTENING
TCP	dev1:3050	dev1:0	LISTENING
TCP	dev1:17500	dev1:0	LISTENING

On POSIX

Active Internet Connections

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(State)
-------	--------	--------	---------------	-----------------	---------

tcp	0	0 *:domain	*:*	LISTEN
tcp	0	0 *:time	*:*	LISTEN
tcp	0	0 *:3050	*:*	LISTEN
tcp	0	0 *:telnet	*:*	LISTEN
tcp	0	0 *:chargen	*:*	LISTEN
tcp	0	0 *:daytime	*:*	LISTEN
tcp	0	0 *:discard	*:*	LISTEN
tcp	0	0 *:echo	*:*	LISTEN
tcp	0	0 *:shell	*:*	LISTEN
tcp	0	0 *:login	*:*	LISTEN

For more information about the command switches available for *netstat* on your platform, refer to the on-line documentation (`man netstat` on POSIX, `netstat /?` on Windows).

Try Telnet

If ping is not working and you know, from running *netstat*, that Firebird's service port is open, you can try using Telnet to test the connection. In a command shell, type:

```
telnet [hostname|IPaddress] <service-port>
```

For example,

```
telnet dbhost 3050
```

If the screen goes blank, with no messages, then Telnet has found the port.

Firewalls

Your connection test might fail if the database server is behind a software or hardware firewall that blocks port 3050 or your reconfigured port.

Subnet restrictions

TCP/IP can be configured to restrict traffic between subnets. If your client machine is part of a complex network of subnets, check with the network administrator that it has unrestricted access to the host server.



WNet cannot route network traffic between subnets.

Problems with events

Although each client connects with the server through a single pipe, Firebird events—a callback mechanism that can channel event notifications back to clients from triggers and stored procedures—use *random* available ports. On static, self-contained networks without internal firewalls, this usually causes no problems. On networks where there are multiple subnets, dynamic IP addressing and tightly configured firewalls, the event channels can fail.

It is possible to configure an IP address in the network explicitly for events traffic. Use the *firebird.conf* parameter ***RemoteAuxPort*** to set statically a port number on the server that is available for event routing.

v.1.0.x

In Firebird 1.0.x, the network administrator needs to configure some way to ensure that a port is available that is always free, open and static. It can be solved in one way or another on most networks.

For more information about Firebird events, refer to Chapter 32, ***Error Handling and Events***.

Initial Configuration

Firebird does not require the intense and constant reconfiguration that many other heavy-duty RDBMS systems do. To begin with—and possibly, for all time—you can go with the defaults and operate Firebird effectively.

A range of configuration options is available for customizing a Firebird server and the host system on which it runs for your particular needs. We will not look at most of them here; they are described in detail in Part VII, **Configuring Firebird**. This topic serves just as an introduction to a few of the parameters that may affect the particular conditions you encounter with your hardware or network.

The Firebird Root Directory

The root directory of your Firebird installation is used in many ways, both during installation and as an attribute that server routines, configuration parameters and clients depend on. If you installed Firebird with all defaults, everything should be where Firebird's components expect it to be. If something about your installation is different, it will be helpful to know about the different ways the location can be configured and the precedence trail that the server follows at startup, to determine it correctly. It is described in Chapter 33, **Configuring Firebird and Its Environment**, in the topic [Finding the Firebird Root Directory](#).

The Firebird Configuration File

When a Firebird server process starts up, it reads the configuration file and adjusts its runtime flags to any non-default values contained in the configuration file. The file will not be read again until next time the server is restarted. The default configuration parameters and their values are listed in the configuration file, commented out by "#" comment markers. It is not necessary to uncomment the defaults in order to make them visible to the server's startup procedure.

The Firebird configuration file is named `firebird.conf` on all platforms and models since Firebird 1.5.

v.1.0.x In Firebird 1.0.x, the configuration file's format and the names of the parameters are different, more restrictive and there are fewer of them. The name of the old configuration file depends on the operating system:

- On Linux/UNIX the name is `isc_config`
- On Windows, the name is `ibconfig`.

Changing configuration parameters

It should be unnecessary to change any defaults until and unless you need to customize something. It is not recommended to do so if you lack a clear understanding of the effects.

A handful of default configuration settings that may be showstoppers for some legacy applications or non-default installations are discussed briefly here in case you suspect you cannot proceed without looking at them. The configuration file can be edited with any plain text editor, e.g. *vim* or *nano* (Linux or MacOSX) or *Notepad* (Windows).



Do not copy the file between platforms without being aware of the conversions you need to do to address the differences in the ways line breaks are stored on each platform! The

free, open source editor **Notepad++** enables cross-platform text conversions and is available on both POSIX and Windows.

Parameter entries in `firebird.conf` are in the form:

```
parameter_name = value
```

- **parameter_name** is a string that contains no whitespace and names a property of the server being configured.
- **value** is a number, Boolean (1=True, 0=False) or string that specifies the value for the parameter

To set any parameter to a non-default setting, delete the comment (`#`) marker and edit the value. You can edit the configuration file while the server is running.



You might prefer to retain the default value beside the '#' marker as documentation and insert a fresh line below it to set your non-default value.

To activate configuration changes, it is necessary to stop and restart the service.

On Linux, you should assume that parameter names are case-sensitive.

v.1.0.x The *ibconfig/isc_config* format is

```
parameter_name    value
```

where the white space between the name and the value can be tabs or spaces, as desired, to please the eye. Each line of the file is limited to 80 characters. Unused parameters and installation defaults are commented with `'#'`.

Parameters relating to file access

Firebird has several parameters for protecting its files and databases from accidents and unauthorized access. If you are porting an old database application or admin tool to Firebird 1.5 or higher, it may be important for you to refer to Chapter 34 for detailed information about these parameters:

- **RootDirectory** can be used to configure the absolute path to a directory root on the local filesystem. It should remain commented unless you want to force the startup procedure to override the path to the root directory of the Firebird server installation, that it would otherwise detect for itself.
- **DatabaseAccess** parameter can be used to provide tighter security controls on access to database files and to support the database-aliasing feature. The default installation configures this to Full—the server can attach to any database in its local filesystem and is always accessed by applications passing the file's absolute filesystem path. Alternative options can restrict the server's access to aliased databases only, or to databases located in specified filesystem trees.

From v.2.5 forward, if Restrict is configured, Firebird uses the first tree root in the list as the default location for creating a database.

It is strongly recommended that you set this option and make use of the database-aliasing feature. For information about database aliasing, refer to the next chapter.

See also **UdfAccess** and **ExternalFileAccess** in Chapter 34, **Configuration Parameters in Detail**, if your new installation will be using an existing database that relies on external functions (“UDFs”) or has database tables defining external text files as the source of data.

TempDirectories (*tmp_directory* in v.1.0.x) can be configured as one of the ways to allocate temporary sort space for the server in a specific disk location.

Other parameters of interest

- **CpuAffinityMask** (*cpu_affinity* in v.1.0.x) solves a problem affecting many, if not most, SMP host machines running Firebird on Windows. The operating system continually swaps the entire Superserver process back and forth between processors, resulting in a system stall whenever CPU usage becomes heavy. It is known as “the see-saw effect” and, if manifest on your system, requires the CPU affinity of the Superserver process to be restricted to a single CPU.

By default, the affinity mask is set to use the first CPU in the array. Refer to the instructions in Chapter 34, **Configuration Parameters in Detail** if you need to change it to a different CPU.

- **LockMemSize** is specific to Classic servers and represents the number of bytes of shared memory allocated to the memory table used by the lock manager. You may need to adjust this if you encounter the error “Lock manager is out of room” on Classic. Refer also to the **LockHashSlots** parameter in relation to this problem.
- **TempBlockSize** and **TempCacheLimit** (**SortMemBlockSize** and **SortMemUpperLimit**, respectively, in v.1.5 and v.2.0) are two parameters that enable you to set and limit the amount of RAM the server uses for in-memory sorting. For Classic servers on hosts with low resources, the default settings may be too large to sustain more than a few connections.
- **DummyPacketInterval** (*dummy_packet_interval* on v.1.0.x) is a relic from 16-bit systems that causes problems on all Windows systems. It should be kept at the default setting of zero on all platforms, to disable it.
- **RemoteBindAddress**: By default, clients may connect from any network interface through which the host server accepts traffic. This parameter allows you to bind the Firebird Superserver or Superclassic service, or Classic on Windows, to incoming requests through one single IP address (e.g. network card) and to reject connection requests from any other network interfaces. This helps to solve problems in some networks when the server is hosting multiple subnets.

For Classic on a POSIX platform, you can include the optional “bind” parameter in [x]inetd.conf:

```
service gds_db
{
    disable      = no
    bind         = 192.150.22.1
    ...
}
```

v.1.0.x **RemoteBindAddress** is not supported on v.1.0.x.

Environment Variables

A number of environment variables can be set, as required, to configure a variety of non-default conditions. At this point, you may be interested in reading about them, their effects

and any associated “do’s” and “don’ts. They are enumerated in Chapter 33, **Configuring Firebird**, in the topic Environment Variables.

OPERATING BASICS

Once you have Firebird server installed on your host server machine, what then?

The server you installed should be running when the installation completes. In this chapter you will find enough to get you going on the basics.

Looking for a User Interface

If you are a complete newcomer to client/server systems, you might expect to see some kind of graphical user interface (“GUI”) that indicates the server is running and offers you ways to do things with databases. The Firebird servers do not interface directly with humans at all. All of their interaction with the world is through the application programming interface—API—that is realised in the client libraries.

Several excellent utility client programs are available in the marketplace for accessing Firebird databases. Many are commercial products, with or without stripped-down free versions. One open source product whose developers work closely with the Firebird Project is FlameRobin—see <http://www.flamerobin.org> for details and downloads. It is available for multiple client platforms, including Linux, Windows and MacOSX.

All Firebird versions are distributed with a command-line SQL query and general database information tool named *isql*. You will find this program in the `/bin/` directory of your Firebird installation. Near the end of this chapter is a brief introduction to *isql*. Full coverage of this tool can be found in Chapter 24.

Running Firebird on POSIX

The administration differences between the Super* servers (Superserver, Superclassic) are much more distinct on POSIX platforms than on Windows. Hence, it avoids confusion to introduce their operating basics separately. However, the newcomer may rest assured that the internals of databases running under different Firebird models are not affected by the architectural differences.

Superserver and Superclassic

The default installation directory for either Superserver (executable: *fbserver*) or Superclassic (*fb_smp_server*) is `/opt/firebird`. The binary itself, found in the `/bin` subdirectory, runs as a daemon process on Linux/UNIX. It is started automatically at the end of a RPM or script installation and whenever the host server is rebooted, by running the daemon script *firebird*, residing in `/etc/rc.d/init.d` in most distributions. This script calls the command-line Firebird Manager utility *fbmgr*.bin. Firebird Manager is surfaced as *fbmgr*, which can be used from a shell to start and stop the process manually.

Superclassic, offered from v.2.5 forward, is designed to improve the utilisation of resources, especially large RAM configurations and multiple processors, on well-resourced 64-bit operating systems. It is not recommended for production deployment on a 32-bit server.

- v.1.0** The default installation directory is `/usr/firebird`. The executable is named *ibserver*, whilst the Firebird Manager is *ibmgr*.

Starting the server: *fbmgr*

If you have to start the Firebird server manually for some reason, log in as root or as the *firebird* user. Take care about which account you use when starting *fbserver* because, once it has been started, all of the objects created belong to that account. If another user later starts the process using one of the other special user accounts, those objects will be inaccessible.



*It is strongly recommended to create a system user named **firebird** and run the Firebird server process under that account.*

To start the process, execute the following command from a shell:

```
./fbmgr -start -forever
```

- v.1.0** `./ibmgr -start -forever`

The **-forever** switch causes the Guardian monitor process (*fbguard*) to start. Under Guardian, if the *fbserver* process should terminate for some reason, it will be restarted automatically.

To start the server without the Guardian, enter

```
./fbmg -start -once
```

- v.1.0** `./ibmgr -start -once`

The **-once** switch makes it so that, if the server crashes, it stays down until manually restarted.

Stopping the server

For safety, if possible, ensure that all attachments to databases have been disconnected before you stop the server.

The **-shut** switch rolls back all current transactions and shuts down the server immediately.

You do not need to be logged in as root to stop the Firebird server with *fbmgr*, but you do need SYSDBA authority. Execute the following command:

```
./fbmgr -shut -password <SYSDBA password>
```

v.1.0 `./ibmgr -shut -password <SYSDBA password>`

Controlled shutdown

If you need to allow clients an interval to complete work and detach gracefully, shut down individual databases using the *gfix* tool with **-shut** and one of a range of available arguments to control detachment. See the topic [Shutting Down a Database](#) in Chapter 35, **Configuring and Managing Databases**.



In versions 1.x on this platform, Firebird does not provide a utility for counting users connected to a database on the Superserver

Other *fbmgr* commands

The general syntax for invoking *fbmgr* from a command shell is:

```
./fbmgr -command [-option [parameter] ...]
```

Alternatively, you can start an interactive *fbmgr* shell session, i.e. go into “prompt mode”.

Type:

```
./fbmgr <press Return/Enter>
```

to bring up the prompt:

```
FBMGR>
```

In prompt mode, the syntax for commands is:

```
FBMGR> command [-option [parameter] ...]
```

For example, you can start the server in either of the following ways:

From the command shell:

```
./fbmgr -start -password password
```

In prompt mode:

```
FBMGR> start -password password
```

Switches for use with *fbmgr*

Following is a summary of the switches available for *fbmgr* and *ibmgr* in either shell or prompt mode.

Table 4.1 Switches for *fbmgr*/*ibmgr*

Switch	Argument	Other switches	Description
<code>-start</code>	<code>-forever</code> <code>-once</code>	<code>-user</code> , <code>-password</code>	Starts the fbserver process if it is not already running
<code>-shut</code>	-	<code>-user</code> , <code>-password</code>	Stops the fbserver process
<code>-show</code>	-	-	Shows host and user
<code>-user</code>	<i>user-name</i> (Recommended: use SYSDBA)	-	Used with <code>-start</code> and <code>-stop</code> switches if system user is not root or equivalent

Switch	Argument	Other switches	Description
–password	SYSDBA password, or password of <i>user-name</i> if not SYSDBA	-	Used with –start and –stop switches if system user is not root or equivalent
–help	-	-	Prints brief help text for <i>fbmgr</i>
–quit	-	-	Use to quit Prompt mode

Classic server

Firebird Classic uses the *xinetd* or *inetd* process to handle incoming requests. (The process it uses depends which is in your Linux version.) There is no need to start the server explicitly. Provided *xinetd*/*inetd* is installed and running, the process runs automatically. When it accepts a request from a Firebird client to attach, it forks off a process named *fb_inet_server* for that client.

How the server listens for connection requests

If Firebird Classic was installed using a scripted or RPM installer, a startup configuration file for *fb_inet_server*, named *firebird*, should have been added to the services that *[x]inetd* knows about. On most Linux distributions, the location of this file is the directory */etc/xinetd.d*. To have *[x]inetd* “listen” for connection requests from clients to your Firebird Classic server, the *firebird* script must be in this directory when the *[x]inetd* process starts.

If *[x]inetd* is running and no client connection requests succeed at all, check whether the *firebird* script is actually where it is supposed to be. If not, the script *firebird.xinetd* may be extracted from the compressed install kit, copied to the correct location and renamed to *firebird*. To make *[x]inetd* “see” the Firebird service:

```
~ $ service xinetd status
xinetd start/running, process nnnn
~ $ sudo service xinetd reload
```

If you see a message indicating that *[x]inetd* is stopped, just start it:

```
~ $ service xinetd start
```



Substitute *inetd* for *xinetd* if your system does not support *xinetd*.

Stopping and starting *[x]inetd* and its services

The *[x]inetd* daemon is itself a service which manages on-demand services like the Firebird Classic attachments. Stopping *[x]inetd* will cause each of the processes which it manages to stop also. Starting or restarting it will cause it to resume listening for requests to start any of its managed processes.

If all of the services in the *./rc.d* root (see above) are safe to shut down, log in as root and stop *x[inetd]* with the following shell command:

```
# service xinetd stop
```

or, as appropriate,

```
# service xinetd stop
```

If $x[inetd]$ has not been configured to restart automatically when shut down, restart it with

```
# service xinetd restart
```

Stopping a Firebird process

If you need stop a runaway Firebird process, in Classic you can do it for any version. Find the offending process by running the `top` command from a shell. This utility displays a list of the most CPU-intensive running processes and updates it constantly. Any *fb_inet_server* instances with extraordinary resource usage should appear in this list.

Get the PID (process ID) of the offending *fb_inet_server* process from the leftmost column of the display. You can use this PID with the `kill` command to send a signal to an errant process. For example, for a PID of 12345, you can attempt a controlled shutdown with

```
# kill 12345
```

If the process remains visible in the `top` display, you can attempt a forced shutdown with

```
# kill -9 12345
```



Exercise great care with kill, especially if you are logged in as root.

Running Firebird on Windows

The notes in this section refer to all models of Firebird for Windows hosts—Superserver, Superclassic and Classic—except where differences are noted.

Servers and the Guardian

On Windows, a Firebird server can run as a stand-alone program or it can be monitored by the Guardian program, *fbguard.exe*. The Guardian provides a capability that emulates the auto-restart capabilities of Windows services and those of POSIX services running with the `-forever` switch. If the *fbserver.exe* (Superserver) or *fb_inet_server.exe* (Superclassic) application should terminate abnormally, the Guardian will attempt to restart it.

It is recommended that you use the Guardian option on hosts running Firebird Superserver as an application. When running Superserver as a service, it is not really necessary, since services can be configured to restart automatically.

- On the services-capable Windows versions, the Superserver program or the Classic listener program can run as a service or as an application. The default installation on a services-capable Windows installs the Firebird server—and the Guardian, if selected—to run automatically as services. Both can be changed to run instead as applications.
- On Windows 95/98, ME and XP Home Edition, Firebird can run only as an application.



When Firebird runs as an application, an icon appears in the system tray. Limited administration tasks can be done manually by right-clicking on the tray icon.

Classic and the Guardian

Do not try to use the Guardian if you have installed the Classic server. Apart from the fact that it is redundant to the architecture of Classic, it can cause “ghost processes” to be created when clients have lost connections through network faults or have terminated their sessions abnormally. These ghosts can accumulate progressively, appropriating resources but doing nothing.

Running as a service

Unless you have a special contrary requirement, it is strongly recommended to keep Firebird server running as a service on a services-capable host.

Stopping and starting the service manually

To stop the service manually, first open a command shell window. Because the NET commands return messages to the command shell, do not try to run them directly using the Run.. option on the Start Menu.

The example here uses the default service name that is installed by the Firebird installers. If you are using v.2.5 or higher, the service might have been installed using a custom service name. If so, use that name instead as the argument to these commands.

Start the shell first by invoking *cmd.exe*.

In the shell, enter the command

```
NET STOP FirebirdServerDefaultInstance
```

To start or restart the server manually:

```
NET START FirebirdServerDefaultInstance
```



The NET STOP and NET START commands do not work on a Firebird server running as (or to be run as) an application.

Stopping and restarting the service using instsvc

The alternative, “native Firebird” way to start and stop the Firebird and Guardian services is to use the *instsvc.exe* utility, which is located in the /bin folder beneath the Firebird root directory. *Instsvc.exe* is used by the system to install the Firebird service—and Guardian, if selected—when the host server is booted up. Because it was not originally intended for general use by humans, it is a DOS-style command with switches.

Open a command shell and navigate to the rootdirectory/bin folder.

To stop the Firebird service:

```
C:\Program Files\Firebird\Firebird_2_5\bin> instsvc stop
```

To (re)start the Firebird service:

```
C:\Program Files\Firebird\Firebird_2_5\bin> instsvc start
```



Using instsvc with these switches does not respectively unload and reinstall the service.

Firebird Manager applets

When Firebird runs as a service, a small degree of administration, including stopping and restarting, can be done through the Firebird Manager Control Panel applet that is

optionally installed by the installer into the system folder appropriate to the host machine's Windows version. The name of the applet in Firebird 2 and higher installations is `Firebird2Control.cpl`.



On older versions, the name of the “official” applet was `FBControl.cpl`. Several third-party applets were in circulation in those days, too, so it would not be unusual to find an older Firebird installation with a differently named applet.

Be aware that old control panel applets will not work on Vista or Windows 7. They may actually break the Control Panel altogether. In reality, the Control Panel applet is superfluous if you are not intending to run Firebird as an application. Administrators usually find it quicker to manage Firebird directly from the Services applet.

Running as an application

If Superserver or the Classic listener is running as an application, you should see an icon in the system tray of the server machine. You won't see a tray icon if the server has not been started. Unless you checked the installation option to start the server automatically, you will need to start it manually.

The appearance of the tray icon depends on whether you are running the server “stand-alone” or you have it under the Guardian's control.



It is recommended to use the Guardian when running Superserver as an application and to avoid it when running Classic.

Figure 4.1 Application tray icon



Starting the server as an application manually

If the Superserver is not running, it can be started or restarted manually by selecting it from the Start | Programs | Firebird menu.

Alternatively, Guardian or the server can be started from the command prompt: Invoke the command shell window and change to the `\bin` folder of your Firebird installation.

Superserver

The Guardian program is called `fbguard.exe` (`ibguard.exe` on v.1.0.x). Use the following command to start it:

```
fbguard.exe -a
```

v1.0 `ibguard.exe -a`

The Guardian places its icon in the tray and automatically starts the Superserver.

The name of the Superserver program is `fbserver.exe` (`ibserver.exe` on v.1.0.x). To start the Superserver directly yourself, without Guardian protection, use this command instead:

```
fbserver.exe -a
```

v.1.0 `ibserver.exe -a`

The server starts and places its own icon in the tray.

Classic server

These notes apply to Firebird 1.5 and later. Classic on Windows is not supported in older versions.

The process that is the “ears” for clients requesting attachment to a Classic server is an initial instance of a program named *fb_inet_server.exe*. If this initial instance of *fb_inet_server* is not running, it will not be possible for a client/server connection to be made and you will get the error “Unable to connect to server. Database cannot be found.”

To start the initializing instance of the Classic server manually as an application, go to a command window, cd to your Firebird\bin directory and type:

```
fb_inet_server.exe -a
```

The server icon should appear in the tray and your server is ready to start receiving connections.

Stopping the server

If you need to stop a server that is running as an application, it is an operation that affects Superserver and Classic differently.

Superserver

Right-click the Firebird Guardian or server icon and choose Shutdown from the context menu. If Guardian is running, it first stops the server and then shuts down itself. Users currently logged on will lose any uncommitted work.

Classic

Taking the Shutdown option from the Server tray icon will prevent any further connections to the server but it does not affect any processes that are currently connected. Under most conditions, it should be unnecessary to “stop” the Classic server.

It is rarely, if ever, necessary to shut down Classic processes manually, since closing a client connection terminates its process instance cleanly and correctly. The only way to stop a Classic process that is running as an application is by applying brute force, via the Task List.

Running Firebird on MacOSX

Operating basics for Firebird on MacOSX are very similar to those on any other POSIX platform. Some variations that are due to vernacular differences should be noted.

The Super* Models

The Start and Stop commands for Superserver and Superclassic are:

```
/Library/StartupItems/Firebird/Firebird start
/Library/StartupItems/Firebird/Firebird stop
```

Classic

Classic on MacOSX uses the *launchd* daemon, which is similar to *[x]inetd* on Linux and related daemons on some other POSIX (described above). As with processes launched by

[x] *intended* on those platforms, you can stop (unload) and [re]start (load) the Classic listener from the command shell.

To disable the listener:

```
$sudo launchctl unload /Library/LaunchDemons/org.firebird.gds.plist
```

To enable (or re-enable) it:

```
$sudo launchctl load /Library/LaunchDemons/org.firebird.gds.plist
```

Mixed Platforms

If your Firebird server is running on Linux, databases must be on Linux, too, on storage devices that are under the direct physical control of the host machine. The same host-database relationship goes for a host running Windows, MacOSX or any other operating system.

A Firebird server cannot serve requests for operations on a database that resides on another box, or even on a VM that is running on the same box. This rules out databases that are located on logical devices in the host's file system that are physically located on another box.

SAN (storage area network) devices and NAS/iSCSI boxes (network-attached storage with iSCSI control) that are visible to the server are seen as locally attached.



On all POSIX versions and on Windows from v.2.5, it is possible to configure the Firebird server to enable reading a database that is on a networked partition. The mechanism that enables this is the configuration parameter `RemoteFileOpenAbility`. Be strictly warned that such a facility should be restricted to databases that have the “read-only” attribute set in the header by SYSDBA or the database owner, using the `gfix` utility or a Services API call.

On the other hand, Firebird is neutral about the client application platform. You can run a client application on Windows workstation, for example, that connects to a database on a Mac OSX host across a TCP/IP network. The client needs a copy of the client library that is built for the platform that the application runs on and, usually, one or more translation layers that “hook up” the application to the client library.

Database aliasing

The concept of database aliasing is not just to relieve keyboard-weary developers. It improves the portability of applications and tightens up control of both internal and external database file access.

Aliases.conf

With database aliasing comes the configuration file `aliases.conf`. It is located in the root directory of your Firebird server installation and should not be moved from there.

Portability

Client applications connect to the server using a connection string which includes the absolute path to the server. The format of the absolute path varies according to whether the server is running on Windows or a POSIX-compliant platform (Linux, Unix, etc.) and,

with a Windows server, whether the clients are using TCP/IP or NetBeui for network connectivity.

For example, suppose we have a server named “hotchicken”.

With the server running on a POSIX-compliant platform, TCP/IP clients would connect to databases using a string of this format:

```
hotchicken:/opt/databases/Employee.fdb
```

With the server running on Windows, TCP/IP clients would connect with a different path format:

```
hotchicken:D:\databases\Employee.fdb
```

Database aliasing makes it so that, for TCP/IP clients, these differences become transparent. The absolute path portion of the connection string goes into the alias file, associated with a simple alias name. For example, in `aliases.conf` on a Linux server, the example could be stored as

```
db1 = /opt/databases/Employee.fdb
```

On a Windows server installation with TCP/IP or NetBEUI (Named Pipes) clients, it could be

```
db1 = D:\databases\Employee.fdb
```

Regardless of whether the server is POSIX or Windows, the TCP/IP connection string becomes

```
hotchicken:db1
```

It is not quite so neat if you want to make it so your application’s connection string to a Windows host is transparent across either a TCP/IP or a NetBEUI connection, however. The UNC notation for a Windows host server to NetBEUI clients means that, although the database alias would be identical, the server portion is not portable:

```
\\hotchicken\db1 versus hotchicken:db1
```



Make sure that all of the entries in `aliases.conf` point to local paths. If you port a database from Windows to POSIX, or vice versa, don’t forget to edit the alias entries.

Access control

The principal benefit of the aliasing option is that it can be used, in combination with the `firebird.conf` parameter ***DatabaseAccess = NONE***, to restrict the server to opening only a specific set of named database files, viz. those which are identified in `aliases.conf`.

To implement database aliasing, edit the file `aliases.conf` in the root directory of your Firebird installation, using a plain text editor such as Notepad (on Windows) or `vi` (on Linux).

aliases.conf

The installed `aliases.conf` looks similar to this:

```
#
# List of known database aliases
# -----
#
# Examples:
#
# dummy = c:\data\dummy.fdb
#
```

As in all of Firebird's configuration files, the '#' symbols are comment markers. To configure an alias, simply delete the '#' and change the dummy line to the appropriate database path:

```
# fdbb1 is on a Windows server:
fdbb1 = c:\Programs\Firebird\Firebird_2_5\examples\empbuild\Employee.fdb
# fdbb2 is on a Linux server
fdbb2 = /opt/databases/killergames.fdb
#
```



These examples would not be in the same aliases.conf file! The first would be in a file on a Windows server, while the second would be on a Linux server.

Each connection request containing a path using the alias format causes the server to read aliases.conf. You can edit aliases.conf whilst the server is running. The server does not have to be restarted for a new entry to take effect: just make sure you save the edits.



Changing the alias for a database will not affect current connections but future connections will use the new or modified alias. Thus, if you change the alias during a process that makes multiple connections—such as gbak—you are likely to cause the process to end with errors.

Connecting using an aliased database path

For TCP/IP connections, using the alias.conf examples above, the modified connection string in your application looks like this:

```
Server_name:aliasname
```

For example,

```
hotchicken:fdbb2
```

For Windows Named Pipes connections:

```
\\hotchicken\fdbb2
```

For a local connection, simply use the alias on its own.

The SYSDBA User and Password

The SYSDBA user is the only Firebird user installed in the security database at installation. The SYSDBA can access any database with full, destructive¹ privileges. At installation time, Windows and MacOSX platform versions have a password assigned that is known to everyone who ever used Firebird or InterBase. It is masterke. (or masterkey, if you prefer: only the first eight characters are significant).

It is imperative that you change the password at the earliest opportunity, using the *gsec* utility that is in Firebird's \bin\ directory:

```
..\BIN> gsec -modify SYSDBA -pw newpassw -user SYSDBA -password masterke
```



*Notice that this command contains two password switches: **-pw**, which takes the new password as its argument, and **-password**, which takes the current SYSDBA password for authentication.*

1. “Destructive” in the sense that SYSDBA or a user with equivalent privileges can alter or delete objects owned by anyone—including the database itself.

Linux

On Linux, the SYSDBA password that is stored in the security database is generated during the installation process and stored in a text file in Firebird's root directory. If your installation used the default file structure, you can find this password in the file `/opt/firebird/SYSDBA.password`.

- If you would like to keep the generated password, write it down somewhere, then delete the file
- If you want to change it to something else, either use *gsec* as above, or run the script `/opt/firebird/bin/changeDBAPassword.sh` and follow the prompts.

Administering Databases

The command-line utility *isql* incorporates tools and techniques for using SQL to maintain database objects, manage transactions, display metadata and manage database definition scripts. A shell interface is available that is consistent across all platforms. This brief introduction will get you started on the basics of connecting (attaching) to a database and creating your first database.

The employee.fdb Database

You can begin practising your Firebird SQL skills right away, using the “old faithful” `employee.fdb` database that is installed in the `../examples/empbuild` subdirectory of your Firebird installation. This database predates Firebird by many years: in InterBase installations its name was `employee.gdb`. While the `employee` database will never win any prizes for excellence in database design, it is a useful sampler for getting started with Firebird's SQL language.



You might like to create an alias for `employee.fdb` if you plan to use it for experimenting with the language features of Firebird. For example, you might add an entry like this to `aliases.conf` on a Windows installation in a folder called `Programs`:

```
emp25 = c:\Programs\Firebird\Firebird_2_5\examples\empbuild\employee.fdb
```

Starting isql

There are several different ways to connect to a database using *isql*. One way is to start its interactive shell. To begin, in a command shell, go to the `/bin` directory of your Firebird installation, where the *isql* program is installed, and start *isql* as follows:

For a POSIX server:

```
[chick@hotchicken]# ./isql
```

For a Windows server:

```
C:\Program Files\Firebird\Firebird_2_5\bin>isql
```

Press Return/Enter to open the program. The *isql* interactive shell should now open, displaying this message:

```
Use CONNECT or CREATE DATABASE to specify a database
```

Using isql

The characters 'isql' stand for Interactive SQL [utility]. Once connected to a database, you can query its data and metadata by entering regular dynamic SQL statements as well as a special subset of statements that work only in the isql environment.

Semi-colon terminators

Each statement you want to execute in isql must be terminated with a designated character. The default terminator is the semi-colon (;). This terminator is not a syntax element of the SQL but a convention that is specific to the usage of this application.

Make sure you terminate each SQL command with a semi-colon. If you forget, the next thing you will see is isql's continuation prompt:

```
CON>
```

The actual purpose of **CON>** (continuation) is to enable the convenient entry of long commands, phrase by phrase. *isql* will just keep filling the command buffer until it encounters a terminator. Thus, if you see the continuation prompt and have already completed your command, simply type a semi-colon and press Enter/Return.



You can change the designated terminator character. In certain isql tasks there is a compelling reason why you would need to. If you are preparing to run a sequence of commands, interactively or as a script, that will contain procedural SQL (PSQL) you might consider changing it at the beginning of the work. For more information, look up SET TERM in Chapter 24.

The CONNECT statement

The **CONNECT** statement is a standard SQL command statement for connecting to a database. Here it's assumed you haven't changed the SYSDBA password yet. If you have done so (recommended!) then use your own password.

For connecting to a Linux/Unix server

All in one line:

```
SQL> CONNECT 'hotchicken:/opt/firebird/examples/empbuild/employee.fdb' user 'sysdba'
password 'masterkey';
```

For connecting to a Windows server

All in one line:

```
SQL> CONNECT 'WINSERVER:C:\Program
Files\Firebird\Firebird_2_5\examples\empbuild\employee.fdb' user 'SYSDBA' password
'masterkey';
```



*On Linux Classic and Windows Superserver it is possible to connect to a database locally, e.g. **CONNECT '/opt/firebird/examples/employee.fdb'** on Linux Classic or **CONNECT 'c:\Program Files\Firebird\Firebird_1_5\examples\employee.fdb'** on Windows Superserver.*

At this point, isql will inform you that you are connected:

```
DATABASE 'hotchicken:/opt/firebird/examples/empbuild/employee.fdb', User: sysdba
SQL>
```

If the server is on Windows:

```
DATABASE "WINSERVER:C:\Program
Files\Firebird\Firebird_2_5\examples\empbuild\employee.fdb", User: sysdba
SQL>
```

Continue to play about with the `employee.fdb` database. You can use `isql` for querying data, getting information about the metadata, creating database objects, running data definition scripts and much more.

To get back to the command prompt type

```
SQL> QUIT;
```

Creating a database using isql

There is more than one way to create a database using `isql`. Here, we will look at one simple way to create a database interactively—although, for your serious database definition work, you should create and maintain your metadata objects using data definition scripts (also known as DDL scripts, SQL scripts, metadata scripts, schema scripts). This topic is covered in detail in Chapter 24

If you are currently logged into a database through the `isql` shell, leave it now with the following command:

```
SQL> QUIT;
```

Next, restart it, without connecting to a database:

For a Linux server:

```
[chick@hotchicken]# ./isql
```

Use `CONNECT` or `CREATE DATABASE` to specify a database

For a Windows server:

```
C:\Program Files\Firebird\Firebird_2_5\bin>isql
```

Use `CONNECT` or `CREATE DATABASE` to specify a database

The `CREATE DATABASE` statement

Now, you can create your new database interactively. Let's suppose that you want to create a database named `test.fdb` on your Windows server and store it a directory named “data” on your D drive:

```
SQL> CREATE DATABASE 'D:\data\test.fdb' user 'SYSDBA' password 'masterkey';
```

Press Enter. The database will be created and, after a few moments, the SQL prompt will reappear. You are now connected to the new database and can proceed to create some test objects in it.

To verify that there really is a database there, type in this query and press Enter:

```
SQL> SELECT * FROM RDB$RELATIONS;
```

The screen will fill up with a large amount of data! This query selects all of the rows in the system table where Firebird stores the metadata for tables. An “empty” database is not empty—it contains a set of system tables which will become populated with metadata as you begin creating objects in your database.



Almost all metadata objects in Firebird databases have identifiers prefixed with “RDB\$”. From v.2.1 forward, another series of system tables, for database monitoring, have the prefix “MON\$”.

To get back to the command prompt type

```
SQL> QUIT;
```

For full information about using `isql`, refer to Chapter 24.

Summary of Command-line Tools

The command-line tools that are built from the Firebird sources and distributed with the official binary kits are summarised in Table 4.1, below.

Table 4.2 Summary of Command-line Tools

Tool	Purpose	Details
isql	SQL client. Comprises an interactive interface with some internal language extensions and shell access and a command-only mode for use in a shell or for automation scripting.	In Chapter 24, Interactive SQL Utility on page 449
gfix	Utility toolset for configuring database attributes and for analysing and repairing some corruptions.	In Chapter 32, Configuring Databases, The gfix Toolset , on page 701
gsec	Utility comprising an interactive shell and a command-only mode for use in a shell, for maintaining server users in the authentication database.	In Chapter 36, Protecting the Server and Its Environment, The gsec Utility , on page 734
fbtracemgr	Utility for working interactively with trace services.	In Chapter 38, Monitoring and Logging Features, Command-line Utility fbtracemgr , on page 778
gstat	Statistics-gathering tool for collecting statistics about the transaction inventory, index distribution and behaviour—can output reports to display screen or files.	In Chapter 38, Monitoring and Logging Features, Collecting Database Statistics —gstat, on page 780
fb_lock_print	Utility that extracts the lock table statistics as either static reports or samplings done interactively at specified intervals.	In Chapter 38, Monitoring and Logging Features, The Lock Print Utility , on page 794
gbak	Backup and restore utility, used for regular backups, for on-disk structure (ODS) upgrades and also, with <i>gfix</i> , in database repair procedures.	In Chapter 39, Backing Up Databases, The gbak Utility on page 818
nBackup	Incremental backup facility for scheduled or ad hoc backups of database page images as the page inventory grows. Comprises both command-line and SQL interfaces.	In Chapter 39, Backing Up Databases, Incremental Backup , on page 837
fbsvcmgr	Simple utility for accessing Services API functions from a shell.	In Chapter 40, The Services Manager, The fbsvcmgr Utility , on page 854

CHAPTER

5

MIGRATION NOTES

Not surprisingly, things do change—for the better, always!—as the project’s talented team of developers introduces new features to solve old problems and meet new challenges. In this chapter, we look at the changes and how they affect you as you upgrade existing databases to take advantage of the improvements.

Version Lineage

If you are planning a migration, it is essential to get acquainted with the “trail” of Firebird development through the versions. The descriptions below are fairly terse, since the larger the gap between the version you migrate from to the one you migrate to, the more complex the issues. You should read, cumulatively, the release notes for each version in your migration trail.

Firebird 1.0

Last sub-release: v.1.0.3

The first official release series of Firebird was v.1.0 in March, 2002. It was still written in the original C language of the InterBase 6.0 beta source code that was released as open source in July, 2000 by a company that was named “Inprise” at the time. Most people knew the company by its old name, “Borland”, whose software products, including InterBase, passed latterly into the new ownership of Embarcadero Inc. Any connection between Firebird and subsequent, closed-source releases of InterBase was one-way, since the proprietors of InterBase have not been participants in Firebird development at all.

That first release had a long beta cycle and constituted a vast number of bug fixes and improvements to the inherited, unfinished code. Firebird 1.0 was distributed for 32-bit Windows, 32-bit Linux and a handful of other POSIX platforms. For Windows, only the Superserver model was distributed. For Linux, both Classic and Superserver were distributed.

Although the Firebird 1.0 series was stable and popular in its day, there was little reason to stay with it once it was succeeded by Firebird 1.5 in the following year. Today, there is no reason why anyone should consider using v.1.0 for new work.

ODS: 10.0 The On-disk Structure version of databases created or restored under Firebird 1.0.x is 10.0. The last sub-release of the v.1.0 series was v.1.0.3, in April, 2003. By that time, Firebird 1.5 was well into its late betas.

Firebird 1.5

Last sub-release: v.1.5.6

Firebird 1.5 was developed as a full recoding of the Firebird sources into the C++ language. Its release was announced at the first international Firebird conference in Fulda, Germany, in May, 2003.

The Classic server model for Windows came in v.1.5, providing a way to utilise multiple processors on Windows host machines. The embedded server for Windows also arrived in this release, enabling compact, self-contained deployments of Firebird and an application for use on stand-alone machines and for web applications.

The server configuration file `firebird.conf` was first served with this release and, with it, ways to make server-based files and executable modules more secure from accidental or intentional interference. The database aliasing feature also came with this version. The security database was renamed, from `isc4.gdb` to `security.fdb`.

EXECUTE STATEMENT was amongst many improvements added to Firebird’s procedural language, PSQL.

ODS: 10.1 The On-disk Structure version of databases created or restored under Firebird 1.5.x is 10.1.

Firebird 2.0

Last sub-release v.2.0.6

The “under-the-hood” enhancements that came with Firebird 2.0 improved performance, security and international language support. Incremental backup (nBackup) came with this release.

The revised INTL (international language) interface for non-ASCII character sets appeared, along with a large number of new character sets and collations and the mechanisms for describing and including others as “plug-ins”.

The old “252 bytes or less” limit on index size was replaced by limits that vary according to page size: specifically, one-quarter of page-size. Calculation of index statistics was improved and many new additions were made to the SQL language, including support for derived tables and the EXECUTE BLOCK syntax for executing PSQL blocks in dynamic SQL.

Firebird 2.0 brought the first 64-bit implementations, but only on Linux. The 64-bit builds for Windows were unstable and would not be introduced until v.2.1.

It was discovered that a bug in Linux meant that Forced Writes had never worked on Linux at all. All Firebird versions and sub-releases prior to v.2.0.4 had this fault, which could allow databases to be corrupted by a system failure.

An important change in Firebird 2.0 on Windows was a complete re-implementation of the “server-only” local protocol, using the XNET subsystem. The XNET and the old IPServer implementations are not compatible with each other, necessitating careful matching of the client library (`fbclient.dll`) and server versions (`fbserver.exe` or `fb_inet_server.exe`). It is impossible to establish a local connection if this detail is overlooked. In a Delphi environment, where InterBase client libraries are often installed by default, this will need special attention.

Garbage collection was reworked and the capability provided to perform “cooperative” garbage collection on Superserver model servers.. Refer to the parameter ***Gcpolicy*** in `firebird.conf`.

The structure of the security database—renamed `security2.fdb`—changed to prevent direct access to the table of users by anyone, even the SYSDBA. From v.2.0 forward, users access the user records by way of a view and ordinary users can change their own passwords.

Implementation of the Services API in the Classic server was completed in this release.



For migrators, it is important to note the following:

*Because the security database changed at Firebird 2, `security2.fdb` is not interchangeable with Firebird 1.5's `security.fdb` nor Firebird 1.0's `isc4.gdb`. A script must be run in order to use an existing, older security database with Firebird 2.0 and higher. See details below in the topic **Security Database**. The upgrade script can be found in Appendix XII.*

ODS: 11.0 The On-disk Structure version of databases created or restored under Firebird 2.0 is 11.0.

Firebird 2.1

Latest sub-release v.2.1.4

Firebird 2.1 brought connection-level triggers, global temporary tables and, for executing recursive queries, an extension to the UNION mechanism known as “common table expressions” (CTEs). A large number of internal functions was added, generally obviating the need to use the standard packages of external functions (UDFs) that are distributed in Firebird’s UDF libraries. Many improvements were made to string predicates, including the ability to apply many of them to text BLOBs.

From this version onward, Windows “trusted user authentication” became available as an option configurable in `firebird.conf` (parameter: **Authentication**), as a parallel to the long-time support on Linux for trusted users.

Database monitoring by query was introduced, underpinned by a new set of system tables whose identifiers start with ‘MON\$’. A command-line tool, `fbisvcmgr`, was introduced as a bare-bones interface to the Services API.

The porting of 64-bit support on Windows servers was completed and appeared in this release.

On Linux, Firebird finally dropped support for the old POSIX pthreads threading model. All Firebird binaries forward from this version support only the “new POSIX threading model”, known as NPTL.

ODS 11.1 The On-disk Structure version of databases created or restored under Firebird 2.1 is 11.1

Firebird 2.5

Latest sub-release v.2.5.1

Firebird 2.5 establishes the basics for a new threading architecture that is virtually unified across the Superserver, Classic and Embedded models, taking in lower level synchronization and overall thread safety. The threaded Superclassic model is introduced for 64-bit platforms, whereby a single Classic process runs as a daemon or a Windows service and spawns multiple threads for connections.

SQL enhancements include the ability to maintain users in the security database by way of SQL requests when logged in to a regular database. In PSQL, autonomous transactions are introduced and, with them, the ability to query out to another database from a PSQL procedure using a much enhanced EXECUTE STATEMENT. In DSQL, regular expression support is introduced with the SIMILAR TO predicate.

On the administrative side, v.2.5 brings trace auditing and logging and enhancements to the database monitoring introduced in v.2.1. A new system role RDB\$ADMIN in the ODS 11.2 database allows the SYSDBA to transfer its privileges to an ordinary user, on a per-database basis, or globally for privileged users.



With this feature in place, Windows “trusted user” authentication works somewhat differently in this release from the way it is in v.2.1.

ODS 11.2 The On-disk Structure version of databases created or restored under Firebird 2.5 is 11.2

Backward Compatibility

As a rule of thumb, databases with an older ODS should be backed up on the older server with *gbak* and then restored from that backup on the newer one. The restored database will have the ODS that the newer server builds.

However, databases created on older Firebird server versions can be opened and managed by a newer version, because each successive server version can read the on-disk structure (ODS) information in the database’s header page. Whilst new versions of the server know what to do with old databases, it is usually the case that new features introduced in a later version will be unavailable to databases having an outdated ODS.

Backups

The biggest inhibition when running a database with an outdated ODS is that you will need to maintain a running version of the old server on the same host as the database and the new server version in order to do backups and certain other maintenance. If you have a serious reason not to upgrade the ODS of the database then you should reconsider the idea of upgrading until the “cannot upgrade ODS” reason is removed.



It is not correct to suppose that old server versions know what to do with a database created or restored with a newer server—they don’t! If you try to connect under these conditions, the server will quickly tell you it is encountering an unrecognized on-disk structure.

Downgrading Databases to Older ODS

Once the database has been restored on the newer server version, it can no longer be read by the older server. Before Firebird 2.5, a trick was discovered that can often (but not always) result in a “downgraded” database, as long as no user objects have been created or altered to make use of features introduced in one or more versions newer than the destination server where the downgraded database is required. Here is the procedure for attempting a downgrade:

- 1 Temporarily rename the *gbak* executable in the installation */bin* directory of the newer server, using whatever name you like, e.g., *gbak25* (or *gbak25.exe* on Windows).
- 2 Copy the *gbak* utility program from the older server to the */bin* directory of the newer server installation—the one under which the database was restored to the newer ODS.
- 3 Running the older *gbak* from its new, temporary location, make a **[t]ransportable** backup of the database. If all goes as hoped, the backup will complete without reporting any errors.
- 4 Now, with the older server running, run its own *gbak* from its */bin* directory and attempt to restore from the backup created in the previous step. If you can connect to the newly created database, then your mission is accomplished.

If a downgrade fails

If this downgrade procedure doesn't work on your database, it probably means you have overlooked some usage of a new feature. If you find it and undo it, you can try the downgrade procedure again.

For an example that some have experienced, it is not possible to “round trip” a downgrade from an ODS 11.2 or higher database, i.e., one created or restored under Firebird 2.5+. You should be able to use the described downgrade method—the problem comes when you try to restore a backup of the downgraded database back to ODS 11.2. The upgrade fails with a “Duplicate Index” error when gbak tries to create the system-generated RDB\$ADMIN role: it already exists!



There is a solution to that one: after restoring the downgrade backup under the older server, log in as SYSDBA and drop the RDB\$ADMIN role.

If the culprit is elusive, the only way to downgrade will be to use **isql -extract** (or an equivalent tool) to create a script, edit the script to remove any DDL that the older server doesn't support and then run the script under the older server to create an “empty” database. Use your favourite data-pumping tool to populate the downgraded database.



Don't forget to reinstate the correct version of gbak on the newer server!

Future Downgrade Support

The Firebird Project team has never officially supported the procedure described and does not claim “downward compatibility” for Firebird databases. Implementing a reliable downgrade path was not on the development agenda at the time of this writing. It could change, though—with open source software, you never can tell when a developer might become interested in an objective that includes such a path.

Deprecated Features

Be aware of features that you may have relied on in legacy applications that have been deprecated and may have become, in any new release, unsupported. Consult release notes, migration guides and other official Firebird Project documentation as an important part of your migration plan.

Later in this chapter you will find a non-exhaustive collection of changes likely to affect legacy applications, in the topic **Application Issues** on [page 95](#).

Preparing to Migrate

First, be clear as to whether the upgrade you are about to do is actually a migration—from a major release to a higher major release—or a minor upgrade—from one minor release (a.k.a. “sub-release” or “patch release”) to another in the same series.

Major Releases

The Firebird major release version series, detailed in the previous section, are 1.0, 1.5, 2.0, 2.1 and 2.5. The first two numbers in the pattern indicate the major release version. You might see the original releases cited sometimes as 1.0.0, 1.5.0, 2.0.0, 2.1.0 and 2.5.0: these numberings indicate the same respective versions.

Sub-releases

As time moves on and development continues on the next major release, for example, v.3.0 at the time of this writing, certain bug fixes and, occasionally, minor improvements, may be backported to versions that are still under maintenance. Fixes may also occur directly to the current version and another active version, as a result of reports in the Firebird Tracker. Periodically, when testers are satisfied with the changes, the version is built and distributed as a sub-release. The third number in the version number pattern—the “minor version number”—indicates the ordinality of the sub-release. For example, Firebird 2.1.4 is the fifth release of the 2.1 series and Firebird 2.5.1 is the second of the 2.5 series.

Table 5.1 should help to identify the differences between a migration and a sub-release upgrade.

Table 5.1 Database Migration: Major release vs Minor release

Major Release to Major Release	Minor Release, Same Series
ODS changes	ODS does not change
API may change	API does not change unless the change addresses a previously undiscovered major bug, regression or vulnerability
System tables may change (new ones, new structure or new language for existing ones)	Structure and enumeration of system tables do not change
Some application code may become invalid due to language syntax changes or API changes	Same application code is expected to work as intended on any sub-release in the series
Drivers may become incompatible	Driver functionality should be unaffected
Run-time environment may change	Run-time environment does not change
Client libraries will change	Client libraries should not change
Logic loopholes ¹ and other “undocumented features” may disappear	Generally, loopholes are retained. However, it can happen that a loophole disappears if it is established as a security vulnerability.
Security database ODS should be upgraded; structure may change as well	Security database does not change
Message files and configuration options usually change, outdating older versions	Message file and configuration options do not change except to correct errors or bugs
Provision and default locations of server assets may change	Provision and default locations of server assets do not change

1. A logic loophole is any unplanned implementation effect, whether documented or not, that allows you to achieve a “trick solution” to a problem. Never consider these tricks as privileged, inside knowledge: it will almost certainly bite you, eventually.

In short:

- if you are moving up to a higher ODS then you are doing a migration and you cannot assume it will be merely a matter of installing the new server software.
- if you are moving up to a newer sub-release within the same series, it is not a migration and, normally, the move can be done without drama.



It is unwise to ignore the release notes for minor releases. It does happen sometimes that a security loophole is discovered between one sub-release and another. In those cases,

the need to remove a vulnerability will override the policy to maintain the upward compatibility of sub-releases.

Dialect 1 Databases

“Dialect” in Firebird and InterBase databases is a way to distinguish between very old databases that were created under InterBase servers with versions lower than v.6 and those that were created new (i.e., not restored) under InterBase 6 or higher. The InterBase 6.0 code, which was open-sourced and became the basis of Firebird, supported for the first time several important ISO-standard SQL data types, such as 64-bit integers and the date-only DATE type. The pre-InterBase 6 DATE type was almost exactly similar to the ISO-standard TIMESTAMPTZ. Some floating-point types were implemented differently in the old InterBase versions.

Databases with these old data type definitions were distinguished as being “in SQL dialect 1”. New databases created under Firebird were distinguished as being “in SQL dialect 3”.



A “dialect 2” was intended for use by clients only, when transitioning existing databases from dialect 1 to dialect 3. It never worked as intended and has been ignored all its life, at least by Firebird.

The ability for Firebird to read and work with old dialect 1 InterBase databases of ODS 10.0 and below has always been retained, for compatibility. Multiple dialect support was never intended to be a permanent feature. Many, if not most, of the SQL language enhancements that Firebird has been built upon over the 11 years of its existence are unavailable to a dialect 1 database. Dialect 1 has been deprecated since Day One of Firebird and it will certainly become unsupported in future.

Determining the Dialect of an Existing Database

There is little point in considering migrating a dialect 1 database past the now unmaintained Firebird 1.5. If you are contemplating a migration to Firebird 2 or higher, you should discover the dialect of the database and, if it is 1, do the steps to convert it to dialect 3.

Using gstat -header

In a command shell, go to the bin directory of your Firebird installation and enter the following command:

```
gstat -header mydb
```

This command returns quite a lot of information from the header page of your database, including its dialect:

```
Database "h:\databases\mydatabase.fdb"
Database header page information:
  Flags                      0
  Checksum                   12345
  Generation                 21181
  Page size                  8192
  ODS version                11.1 <-----
  Oldest transaction         11109
  Oldest active              18124
  Oldest snapshot            18124
  Next transaction           18128
  Bumped transaction         1
```

```
Sequence number      0
Next attachment ID    10
Implementation ID     16
Shadow count         0
Page buffers         4092
Next header page      0
Database dialect      1 <-----
Creation date         May 7, 2008 11:13:56
Attributes            force write
```

```
Variable header data:
Sweep interval:      20000
*END*
```



Notice that the database alias (*mydb*) that was used in this command resolved to the actual path of the database file when *gstat* was invoked.

From the above header information, we can guess that this is probably an old InterBase database (dialect 1) that was upgraded to Firebird 2.1.

Using interactive isql

An alternative for getting the SQL dialect of a database is to use the SHOW SQL DIALECT command when using *isql* interactively. For more information about this, refer to Chapter 24, *Interactive SQL Utility (isql)*.

Changing a Database to Dialect 3

If your old database is in dialect 1 and you judge that it’s time to take it into the 21st century, it is highly recommended that you do the conversion before you begin any migration.

To perform the conversion, under the older server, follow the steps below:

- 1 Set up a working directory, away from the location of your production database, where you will create the dialect 3 database.
- 2 Using *isql* or another utility that can extract metadata, extract the full metadata of the database to a file (script), having it write the output to a file in your working directory. You might find it useful to use the “.sql” suffix for the file, for ease of identification in future.
- 3 Using a reliable plain text editor—not a word processor!— edit the metadata script and address the following issues, as required:
 - a Identify any column and domain definitions that are populated by generators and change them to BIGINT type if they are in large tables that take frequent inserts.
 - b In PSQL modules (stored procedures, triggers) find variables declared as INTEGER and change any that will have values from generators, or from fields that store generator values. Change them to BIGINT, too, in step with the changes in (a).
 - c Find columns and domains that use fixed decimal (NUMERIC, DECIMAL) types with precision larger than 9. In a dialect 1 database such columns are stored as DOUBLE PRECISION floating point, even though their definitions appear like genuine fixed decimal numbers. Mark these columns for attention during the subsequent data-pumping operations.

- **TIP** • Consider defining any numeric columns or domains that store “counted” data with precision greater than 9, such as money or usage statistics, as NUMERIC(18,n) rather than using a precision between 10 and 17. You don’t sacrifice any storage from that, since Firebird stores all 64-bit fixed decimals as a 64-bit integer, with scale stored as a SMALLINT in a separate system table field
- d The dialect 1 DATE type is a TIMESTAMP in dialect 3. Additionally, dialect 3 has the SQL DATE type, which is date only.
If you already restored an old ODS 9 or lower InterBase database backup into any Firebird version then all the old DATE columns and domains should already be TIMESTAMP.
- 4 Use *isql* or another utility to create a new database using the amended script. Watch carefully while running the script. If errors occur, drop the database, fix the script and run it again.
- 5 When you are satisfied that you have a good database, use a data-pumping tool, such as the one in the IB_SQL utility (free from www.ibobjects.com) or pick one from the list at <http://www.ibphoenix.com/download/tools/migration>, to “pump” the data from the dialect 1 database to the new, empty dialect 3 one.

Pumping 64-bit numbers

To convert the data in a NUMERIC or DECIMAL column of precision 10 or greater to match the dialect 3 definition, they should be cast explicitly during the data pumping process, to convert them to the dialect 3 64-bit type.

A good data-pumping tool should take care of this for you automatically. If you are “rolling your own” data-pumping routine, your testing will be alert you to the need to cast by the appearance of one or more ISC errors that indicate data type mismatches.

For example, you may see the standard arithmetic overflow exception 335544321, “Arithmetic exception, numeric overflow, or string truncation”. If you have written a stored procedure with input parameters to do the conversion, the error may be 335544512, “Input parameter mismatch for procedure <procedure-name>”.

The *gfx*-sql_dialect Switch

The *gfx* toolset has a switch `-sql_dialect`, whose original purpose was intended to be to change the header page of a database, to identify it as being dialect 3. And it does just that! However, as it does NOT convert any data, it is less than useless. You do not want a database full of dialect 1 data that announces itself as dialect 3!

Don’t use it.

Back Up!

It can’t be emphasised too much that you must make at least *two* backups of your existing database before you begin the work of migrating it.

First, using *gbak* under the older server, make a full backup and restore it to check that it went perfectly.

If the restore fails or the restored database exhibits any problems, resolve any issues and repeat. Do not proceed until you are satisfied that you have a good backup.

Make a copy of the good backup and move it to the location where you will be working on it.

If you intend to use the user data in your old security database, rather than take the empty one that is created in the installation directory of the newer server and populate it all-new from offline data, then do the same thing with the old security database.



*Migrating a security database from v.1.0 or v.1.5 to any version 2 or higher server requires some manual work before it is ready. Study the **Security Database** task below carefully.*

Migration Tasks

- Get the latest sub-release!
- Convert dialect 1 databases to dialect 3 as described above
- Check or unset the FIREBIRD variable
- Upgrade the security database (v. 2.0 and higher)
- Repair metadata text and text BLOBs (v.2.1 and higher)
- Platform issues:
 - For Windows server and clients, check that the correct Microsoft runtime libraries are installed
 - For Linux servers and clients, check your glibc and kernel versions
 - For MacOSX servers and clients, check your OSX version
- Consider application issues

Get the Latest Sub-release!

Ensure that the kit you have for your new server is the latest released minor version for the major version you want to move to.



Do not use kits downloaded from unofficial sites or repositories. All official Firebird kits are at Sourceforge and are linked to from the Download pages of the main Firebird site, approved support sites such as <http://www.ibphoenix.com> and, for Linux distro-specific installs, in the approved repositories of those distributions.

The FIREBIRD Variable

FIREBIRD is an optional environment variable introduced in v.1.5 to replace the old INTERBAS variable. It provides a system-level pointer to the root directory of the Firebird installation. If it exists, it is available everywhere in the scope for which the variable was defined.

The FIREBIRD variable is NOT removed by scripted uninstalls and it is not updated by the installer scripts. If you leave it defined to point to the root directory of an older installation, there will be situations where the Firebird engine, command-line tools, cron scripts, batch files, installers, etc., will not work as expected.

Unless you are very clear about the effects of having a wrong value in this variable, you should remove or update it before you begin installing a new version of Firebird. After doing so, you should also check that the old value is no longer visible in the workspace

where you are installing Firebird—use the `SET FIREBIRD` command in a Windows shell or `printenv FIREBIRD` in a POSIX shell.

Platform Issues

For those migrating to Firebird 2.5, do not consider installing Superclassic on a 32-bit platform for production use with more than a handful of users.

Pay attention to the following known issues that are platform-specific:

C++ Runtime Libraries on Windows

Check that the correct Microsoft Visual C++ runtimes are installed on both clients and servers. It is important to read the release notes and review Chapter 2 carefully on this subject whenever you do a migration, since components of both the server and the client, including the embedded model, call functions in these runtimes.



The runtimes are a particular trap for migrating to v.2.1 or v.2.5, due to the reorganisation that happened in the MSVC8 compiler to accommodate Microsoft's "assemblies". Older versions of Firebird, compiled in MSVC7 or lower, use MS runtimes that tend to be present on servers—although it would be a mistake to assume that they are.

Linux

glibc and Kernel Versions on Linux

If your *glibc* version is v.2.7 or lower, it may need to be upgraded before using Firebird 2.5 Classic/Superclassic. High-load systems have been reported as unstable unless installed with a *glibc* version of 2.9 or higher. The situation with *glibc* 2.8 is unclear.

Linux deployers should also note that versions 2.1 and above will not run on kernels that do not support the new POSIX threading model (NPTL).

Intel Xeon 32-bit Quad core CPUs

On Linux, some older Intel Xeon 32-bit quad cores cause serious resource and performance issues with Firebird 2.1 and higher versions.

MacOSX

If migrating to v.2.5 or higher on MacOSX, ensure that your MacOSX version is 10.6 (Snow Leopard) or higher. If you want to use an earlier version of OSX you will need to migrate to an earlier version of Firebird.

Security Database

In all Firebird versions in the “2” series—2.0, 2.1 and 2.5—the name of the security database is `security2.fdb`. Its structure is different to that of earlier versions. At the same point, server security was tightened in several ways

Migrating the Security Database



Do not try to “upgrade” `security.fdb` by renaming it or copying it as `security2.fdb`. You will be unable to attach to the Firebird server!

If you want to retain the user logins from your old `security.fdb` (from v.1.5) or `isc4.gdb` (from v.1.0), a script, named `security_database.sql` is provided in the `misc/upgrade/security` subdirectory of your installation.



*Download the “.zip” or compressed tarball kit for your OS platform and extract the script. For the reader’s convenience, a copy is included in Appendix XII, **Upgrade Scripts**.*

Begin by making a *gbak* backup of your old `security.fdb` or `isc4.gdb` under the old server, using the old server’s version of *gbak*.

Restore it under the Firebird 2.x server, using the *gbak* that is distributed with the new server, in a location that is NOT Firebird’s root directory. For this operation, use the “empty” `security2.fdb` that is located in the installation directory.



Include the `page_size` parameter and make the page size at least 4 KB. The new `security2.fdb` will not work with a smaller page size.

Using *isql* or another utility that can run script, connect to the restored database as `SYSDBA` and run the script.

Now, stop Firebird (Superserver) or log out of all attachments (Classic) and rename the installed `security2.fdb` to anything you like.

Copy the upgraded database to the Firebird 2.x root directory, renaming it to `security2.fdb`.

Finally, restart Firebird and connect using your old user names and passwords.

Security Changes to Watch

Besides the changed security database format, the “2” series brought changes in security handling that introduce incompatibilities with how your existing applications interface with Firebird’s security. For more details, refer to the later topic in this chapter, **Application Issues**.

Metadata Repair

Firebird versions prior to v.2.1 carry some problems concerning character sets and metadata extraction that were addressed in v.2.1. In summary, if any of the database’s system objects contain metadata that were created with characters not in the ASCII character set, the metadata will be unreadable after upgrading the database to v.2.1.

During object creation or modification in these past versions, the metadata text stored for the source code of procedures and triggers is not transliterated from the client character set to `UNICODE_FSS`, the character set that the engine uses in general for storing metadata in the system tables. Similar omissions occur for constraints, descriptions, text associated with default declarations, and so on.



This behaviour is similar to what can occur in any version of Firebird if objects are created or altered from a client connected with character set `NONE`, or when writing metadata text that is not in character set `UNICODE_FSS` while attached to the server by way of a client with character set `UNICODE_FSS`.

A similar problem is present when the engine reads text BLOBS: it fails to transliterate from the character set in which the BLOB was stored to the character set of the client connection.

Repairing the Metadata Text

If upgrading to v.2.1, use the supplied scripts on a backup restored under the new server version

- If upgrading directly to 2.5 from v.2.0.x or lower, restore the backup using the metadata fix switch `-FIX_FSS_METADATA` that was introduced in v.2.5 to supersede the scripts

The script method

Use this method for repairing a database that has been upgraded to v.2.1.x, i.e., a database with ODS 11.1 that has not been fixed previously. The method requires running scripts and stored procedures over the database in multiple passes. It is strongly recommended that you detach from and reattach to the server *for each pass*.

The scripts, named `metadata_charset_create.sql` and `metadata_charset_drop.sql`, are provided in the `misc/upgrade/metadata` subdirectory of your installation.



Make a file copy of the database first, while no users are connected to it, and store the copy in a safe location.

Step 1: Create the procedures in the database

From Firebird's root directory, connect to the database using `isql`. In the `isql` shell, type

```
SQL> input 'misc/upgrade/metadata/metadata_charset_create.sql';
SQL> COMMIT WORK;
SQL> EXIT;
```

Step 2: Check the database

Calling the selectable stored procedure `rdbs$check_metadata` should list all the objects that are touched by it.

Reconnect to the database using `isql`. In the `isql` shell, type

```
SQL> select * from rdbs$check_metadata;
SQL> COMMIT WORK;
SQL> EXIT;
```

- If no exception is raised, the metadata are in good shape and you can skip to **Step 4: Removing the upgrade procedures**.
- If you encounter an exception, the first bad object is the last one listed before the exception.

Step 3: Repair the metadata

In order to repair the metadata, you need to know the character set in which the bad objects were created.

The `rdbs$fix_metadata` procedure will return the same data as `rdbs$check_metadata`, but the metadata texts will have been changed.¹

1. A Norwegian reviewer noted that the scripts did not work perfectly when migrating Norwegian character metadata from 1.5 to 2.1. Three Norwegian letters – æ-ø-å – and their upper case versions –Æ-Ø-Å – caused problems. He wrote a conversion program for changing occurrences of these characters in some fields in the RDB\$xxx tables in v.1.5 databases to strings like «<xx\$>» and «<\$XX\$>», and another program to change back from these strings in the v.2.1 databases. E.g., ø and Ø became <\$oe\$> and <\$OE\$> for a while.



The upgrade script will not work correctly if your metadata objects were created using multiple character sets. **Run the script only once!**

Reconnect to the database using *isql*. In the *isql* shell, type

```
SQL> input 'misc/upgrade/metadata/metadata_charset_create.sql';
SQL> --for the character set argument, replace 'WIN1252' with the name of the
SQL> --character set used in creating your system objects
SQL> select * from rdb$fix_metadata('WIN1252');
SQL> COMMIT WORK;
SQL> EXIT;
```

Step 4: Removing the upgrade procedures

After Step 3, the upgrade procedures should be removed. Reconnect to the database using *isql*. In the *isql* shell, type

```
SQL> input 'misc/upgrade/metadata/metadata_charset_drop.sql';
SQL> COMMIT WORK;
SQL> EXIT;
```

Fixing User Data

You may also need to repair text BLOBs in your database if the v.2.1 engine fails to transliterate them due to the historical mismatches described in this section. If you see an exception reported as “malformed string” when a BLOB is being read for output then this is the clue that your BLOB text needs repairing.

Use the supplied scripts as a model that you can apply to the affected tables using the character set that ought to match with other character data accessed by your client applications.



Use of the scripts is not recommended for repairing an ODS 11.2 or higher database. Read on for the simplified method introduced with the v.2.5 *gbak* utility.

The *gbak* method

If upgrading directly to 2.5 from v.2.0.x or lower, restore the backup using the metadata fix switch `-FIX_FSS_METADATA` that was introduced in v.2.5 to supersede the scripts and simplify the correction of text in system objects.

The switch `-FIX_FSS_DATA` works similarly for correcting malformed text in text BLOBs in user tables.

Switch `-FIX_FSS_METADATA`

The *gbak* switch `-FIX_FSS_METADATA <character_set>` directly replaces the `rdb$fix_metadata` procedure that would be used to upgrade older databases to ODS 11.1 (Firebird 2.1.x). It is available in *gbak* versions 2.5 and above, with server versions higher than v.2.1.x.

To fix wrongly stored metadata in the system tables, you need to know the character set that was used when the system objects were defined. Metadata read from a backup is treated as being encoded in the character set specified in the character set argument, instead of UNICODE_FSS, and is transliterated for storage in the restored database.

```
gbak -restore path/to/mybackup.fdb mydbalias -FIX_FSS_METADATA WIN1252 [..other options..]
```

As in the previous example, replace the **WIN1252** argument with the name of the character set in which your database was created.



*Using character set **NONE** for this argument has no effect with the **-FIX_FSS_METADATA** option.*

Switch **-FIX_FSS_DATA**

Use the switch **-FIX_FSS_DATA <character_set>** to repair VARCHAR and BLOB columns in user tables during the upgrade restore. Again, you need to know the character set that was used when the objects were defined.

```
gbak -restore path/to/mybackup.fdb mydbalias -FIX_FSS_DATA WIN1252 [..other options..]
```

As in the previous examples, replace the **WIN1252** argument with the name of the character set in which your database was created.

- If the supplied character set is **NONE**, the columns are created with **UNICODE_FSS** as their character set and the data will be stored as-is
- If the supplied character set is anything but **NONE**, **UNICODE_FSS** columns are created and the data read from the backup is treated as being in the encoding of that character set and will be transliterated for storage in the columns.



*When restoring, use the **-v[erify_output]** option to log the restore. Then, if **gbak** encounters a malformed string, it will supply a hint directing you to use the **-FIX_FSS_METADATA** and **-FIX_FSS_DATA** switches.*

*The switches may be abbreviated to **-FIX_FSS_M** (for metadata) and **-FIX_FSS_D** (for data).*

Application Issues

Possibly the larger issues with migration come from the need to modify application code. Legacy code often takes advantage of “undocumented features”, logic loopholes and the tolerance of old server versions towards sloppy SQL syntax.

Connection Parameters

A long-standing, legacy loophole in the handling of DPB parameters—those that are passed as a block by drivers and other connection routines when attaching a client—enabled ordinary users to make connection settings that could lead to database corruptions or give them access to SYSDBA-only operations. This loophole was closed at versions 2.1.2 and 2.0.5. That change that could affect several legacy applications, and old versions of database tools and connectivity layers (drivers, components).

Security

Direct connections to the security database became disallowed from the “2” series forward. The only way to configure user accounts from an application in these newer versions is by using the Services API. The older calls to add, delete and modify users via the DPB do not work any more.



This affects especially legacy Delphi applications and components that were written for InterBase and were ported to early versions of Firebird.

Under the 1.x versions, non-SYSDBA users can see other users' accounts in the security database. In newer versions, a non-privileged user can retrieve or modify only its own account. It can also change its own password.

Remote attachments to the server without a user name and password, even by root or Administrator users, are prohibited in the “2” series.

On Windows in v.2.1.x, operating system users with Local Administrator or Domain Administrator group privileges can attach to databases with a “blank” Firebird user name and password if the Authentication parameter is set to ‘trusted’ or ‘mixed’. From v.2.5, enabling this condition requires explicit configuration. If your v.2.1 applications rely on this feature, study the pieces pertaining to v.2.5 and higher in the section *Managing User Access* in Chapter 36, *Protecting the Server and its Environment*.

SQL Language Changes

Improvements and standardisation of the SQL Language are an ongoing part of Firebird development. It is essential to check the SQL sections of release notes if you are considering a migration. As a general rule, the changes do not break client applications that have been written so as to consciously avoid loopholes and shortcuts that have been picked up in ad hoc ways from user forums and adopted with no mind for their potential to disappear.

However, some changes do affect code that has not been kept up to date as bugs, illogicalities and non-conformances have been addressed.

Keywords, reserved or not, can and do change!

For better or for worse, the lists of keywords that are reserved or not reserved have undergone many changes since 2000, when Firebird began. The objectives of all these changes have been alignment with the ISO standards and reduction of the number or scope of non-standard keywords. The latest list of reserved and non-reserved keywords can be found in Appendix II.

Tighter rules for disambiguation of qualifiers

One such to note carefully is the tightening of the rules in the “2” series regarding ambiguity in qualifying columns in multi-table queries.

Use of an alias makes it invalid to use the table identifier to qualify a column. When an alias is used for a table, that alias, and not the table identifier, must be used to qualify columns. The only alternative is to avoid aliases altogether. For some queries, it is impossible to avoid aliases.



A sounder practice is to avoid table names as qualifiers and use only aliases.

When this disambiguation rule was implemented, it was accompanied by a temporary configuration parameter that enables the rule to be relaxed for a period, to let programmers catch up. The parameter is *RelaxedAliasChecking*, which you can review in Chapter 34, *Configuration Parameters*.

Update or insert to a column not more than once per statement

In the “1” versions you could make multiple “hits” on the same column in an INSERT or UPDATE statement. For example,

```
INSERT INTO ATABLE(COLA, COLB, COLA) ...
```

or

UPDATE ATABLE

SET COLA = x, COLB = y, COLA = z

In the “2” series and above, multiple accesses to the same column are rejected: a single statement strictly updates or inserts to a given column not more than once.

SET clause assignment behaviour changed in v.2.5+

In versions prior to Firebird 2.5, the SET clause of the UPDATE statement assigns columns in the user-defined order, with the NEW column values being immediately accessible to the subsequent assignments. This non-conformant behaviour was changed at v.2.5, with only the OLD column values accessible to any assignment in the SET clause.



*For compatibility with legacy code, the 2.5+ server can be temporarily configured to “misbehave” like older versions by setting the configuration parameter **OldSetClauseSemantics**. That parameter will disappear in a future release.*

Validation of user-specified query plans gets stricter

User-specified plans are validated more strictly in the “2” series than in the “1” series. For example, older versions would accept without complaint plans with missing references—a bug that was fixed in the “2” series. A plan must refer to all tables in query now.

If you encounter an exception related to plans, e.g., “Table ATABLE is not referenced in plan”, you will need to adjust the plan.



Plan errors could show up during the upgrade restore and stop you in your tracks. If you use explicit query plans in client code or in PSQL modules, visit your sources and make them semantically correct before attempting to run the upgrade restore.

Watch out for NOT IN() queries!

If your legacy code involves queries that use the NOT IN() predicate, be aware that the new implementation of this predicate in the “2” series, to conform to the ISO standard, can not use an index. Statements containing this predicate will execute very slowly. In most cases, you can restore the former level of performance by replacing it with a NOT EXISTS() expression without affecting the results. Better still, if possible, re-jig the logic so that you don’t have to use a NOT expression at all. Make certain to check results.

Changes in the API

For those who write programs or interfaces using the API, make a special point to study the API section of all release notes between the old version and the new version.



This book does not describe the API.

Long-deprecated configuration parameters **OldParameterOrdering** and **CreateInternalWindow**, deprecated already for some years, were removed from Firebird 2.5 forward.

Migration Tools

The matter of tools to speed up and simplify migration tasks was already mentioned in the topic about converting dialect 1 databases to dialect 3.

Firebird users are well-served by third-party tools for many of the more tiresome migration tasks. The IBPhoenix web site (<http://ibphoenix.com/download>) tries to keep an up-to-

date list of tools and tasks, with links to download sites. In many cases, such tools are free of charge, or shareware; in others, most commercial tools developers have trial or “light” versions available at no charge.



*Make certain to take a good metadata extract of the old version of your database, while running isql or your third-party utility under the old version of the server. **Annotate it** as you proceed through the migration steps. Your notes will be worth their weight in gold if you encounter any overlooked issues during and after your database upgrade.*



The
Firebird Book
A Reference for Database Developers

SECOND EDITION

PART II



Firebird Data Types & Domains

ABOUT FIREBIRD DATA TYPES

Data type is the primary attribute that must be defined for each column in a Firebird table. It establishes and constrains the characteristics of the set of data that the column can store and the operations that can be performed on the data. It also determines how much space each data item occupies on the disk. Choosing an optimum size for data values is an important consideration for network traffic, disk economy and index size.

The Basics

Firebird supports most SQL data types. In addition, it supports dynamically-sizeable typed and untyped binary large objects (BLOBs) and multi-dimensional, homogeneous arrays of most data types.

Where to Specify Data Types

A data type is defined for data items in the following situations:

- specifying column definitions in a CREATE TABLE specification
- creating a re-usable global column template using CREATE DOMAIN
- modifying a global column template using ALTER DOMAIN
- adding a new column to a table or altering a column using ALTER TABLE
- declaring arguments and local variables in stored procedures and triggers
- declaring arguments and return values for external (user-defined) functions

Supported Data Types

- Number types (Chapter 7)

- BIGINT, INTEGER and SMALLINT
- NUMERIC and DECIMAL
- FLOAT and DOUBLE PRECISION
- Date and time types (Chapter 8)
 - DATE, TIME, and TIMESTAMP
- Character types (Chapter 9)
 - CHARACTER, VARYING CHARACTER, and NATIONAL CHARACTER
- BLOB and ARRAY types (Chapter 10)
 - BLOB, typed and untyped
 - ARRAY

Booleans

Up to and including release 2.5, Firebird does not provide a Boolean (“logical”) type. The usual practice is to define a 1-char or SMALLINT domain for generic use whenever the design calls for a Boolean.

For tips about defining Boolean domains, refer to Chapter 11.

SQL “Dialects”

Firebird supports three SQL “dialects” which have no practical use except to facilitate conversion of an ancient InterBase v.5 (or older) database to Firebird. Firebird’s “native” dialect is currently known as “dialect 3”. By default, a Firebird server creates databases in this native dialect. If your Firebird experience brings with it no cargo of existing assumptions nor any legacy databases that you want to upgrade to Firebird, you can safely “go native” and ignore all of the notes and warnings about dialect 1.

If you are an ex-InterBase user, or you have used outdated migration tools to convert another RDBMS to InterBase, then SQL dialects will be an issue for you in one way or another.

As you work your way through this book, issues that are affected by SQL dialect are annotated appropriately for your attention. However, some of the more serious effects arise from the dialectal differences between data types. For this reason, the question of dialects gets space in Chapter 5, *Migration Topics*, in the topic *Dialect 1 Databases*.

Quick check for SQL dialect

If you have a legacy database and you don’t know which dialect your database speaks, you can do a quick check with the *gstat* tool, located in the `../bin/` directory of your Firebird installation. You don’t need any special privileges for this. Go to that directory and type

for POSIX:

```
./gstat -h yourserver:/path/to/your/database
```

for Windows:

```
gstat -h yourserver:d:\path\to\your\database
```

You will see a list of useful information about your database, which *gstat* has read from the database header page. The dialect and on-disk structure (ODS) are included in the list.

- If you are using the isql interactive tool, you can get a similar summary using SHOW DATABASE.
- The SHOW SQL DIALECT command displays both the client and database dialects.



Any database that has an ODS lower than 10 will always be dialect 1.

Context Variables

Firebird makes available a number of system-maintained variable values in the context of the current client connection and its activity. These context variables are available for use in SQL, including the trigger and stored procedure language, PSQL. Some are available only in PSQL and most are available only in Dialect 3 databases.

The context variables are fully listed and described in Appendix III, *Context Variables*. Table 6.1 is a sampling of some commonly used context variables.

Table 6.1 A Sampling of Available Context Variables

Context variable	Data type	Description	Availability
CURRENT_CONNECTION	INTEGER	System ID of the connection that is making this query	Firebird 1.5 onward, DSQL and PSQL
CURRENT_TRANSACTION	INTEGER	System ID of the transaction in which this query is being requested	Firebird 1.5 onward, DSQL and PSQL
CURRENT_TIMESTAMP	TIMESTAMP	Current date and time on the host server's clock to the nearest millisecond	Prior to v.2.0, milliseconds were not returned. The UDF <i>GetExactTimestamp()</i> could be used with the older versions to return server time to nearest ten-thousandths of a second.
CURRENT_TIME	TIME	Current time on the server's clock, expressed as seconds since midnight	All versions, all SQL environments, Dialect 3 only
CURRENT_USER	VARCHAR(128)	User name that is communicating through this instance of the client library	All versions, all SQL environments

Points about Usage

Remember that these are transient values.

- CURRENT_CONNECTION and CURRENT_TRANSACTION have no meaning outside the current connection and transaction contexts respectively. Do not treat them as perpetually unique, since the Firebird engine stores these identifiers on the database header page. After a restore, their series will be re-initialized to zero.

- `CURRENT_TIMESTAMP` records the server time at the start of the operation and should not be relied upon as the sole sequencer of a batch of rows inserted or updated by a single statement.
- Even though `CURRENT_TIME` is stored as the time elapsed since midnight on the server's clock, it is a `TIME` value, not an interval of time. To achieve a time interval, use timestamps for start and finish times and subtract the start time from the finish time. The result will be a time interval in days.
- The date/time context variables are based on server time, which may be different to the internal clock time on clients.

Examples using context variables

This statement returns the server time at the moment the server serves the request of the Firebird client:

```
SELECT CURRENT_TIME AS TIME_FINISHED FROM RDB$DATABASE;
```

In this insert statement, current transaction ID, the current server timestamp and the system user name will be written to a table:

```
INSERT INTO TRANSACTIONLOG (TRANS_ID, USERNAME, DATESTAMP)
VALUES (
    CURRENT_TRANSACTION,
    CURRENT_USER,
    CURRENT_TIMESTAMP);
```

Pre-defined Date Literals

A number of date literals—single-quoted strings which Firebird SQL will accept in lieu of certain special dates—are available to both Dialect 1 and Dialect 3 databases. In Dialect 1, the strings can be used directly; in Dialect 3, they must be cast to type. Table 6.2 shows the usage in each dialect.

Table 6.2 List of predefined date literals

Date literal	Date substituted	Dialect 3 Type	Dialect 1 Type
'NOW'	Current date and time	TIMESTAMP	DATE (equivalent to Dialect 3 Timestamp)
'TODAY'	Current date	DATE (date-only type)	DATE (TIMESTAMP) with zero time part
'YESTERDAY'	Current date – 1	DATE	DATE (TIMESTAMP) with zero time part
'TOMORROW'	Current date + 1	DATE	DATE (TIMESTAMP) with zero time part



The Dialect 1 `DATE` type is equivalent to a Dialect 3 `TIMESTAMP`. In Dialect 3, the `DATE` type is date only. Dialect 1 has no equivalent type.

Examples using predefined date literals

In a Dialect 1 database, this statement returns exact server time:

```
SELECT 'NOW' AS TIME_FINISHED FROM RDB$DATABASE;
```

In a Dialect 3 database the date literal must be cast as a timestamp type:

```
SELECT CAST('NOW' AS TIMESTAMP) AS TIME_FINISHED FROM RDB$DATABASE;
```

This update statement sets a date column to server time plus one day in Dialect 1:

```
UPDATE TABLE_A
  SET UPDATE_DATE = 'TOMORROW'
 WHERE KEY_ID = 144;
```

The same operation in Dialect 3, with casting:

```
UPDATE TABLE_A
  SET UPDATE_DATE = CAST('TOMORROW' AS DATE)
 WHERE KEY_ID = 144;
```

Optional SQL-92 Delimited Identifiers

In Dialect 3 databases, Firebird supports the ANSI SQL convention for optionally delimiting identifiers. To use reserved words, case-sensitive strings, or embedded spaces in an object name, enclose the name in double quotes. It is then a delimited identifier. Delimited identifiers must always be referenced in double quotes.

For more details, follow up this topic in Chapter ??, Database object naming conventions and constraints.

For more information about naming database objects with CREATE or DECLARE statements, refer to Part Three, ***A Database and its Objects***. Refer to Appendix II for the list of keywords that are reserved for SQL.

Columns

Data in a relational database system such as Firebird are arranged logically in sets consisting of rows and columns. A column holds one piece of data with defining attributes which are identical for every row in the set. A column definition has two required attributes—an identifier (or column name) and a data type. Certain other attributes can be included in a column definition, such as a CHARACTER SET and constraints like NOT NULL and UNIQUE.

Sets that are defined for storing data are known as tables. The row structure of a table is defined by declaring a table identifier and listing the column identifiers, their data types and any other attributes needed, on a column-by-column basis.

A simple example of a table definition is:

```
CREATE TABLE SIMPLE (
  COLUMN1 INTEGER,
  COLUMN2 CHAR(3),
  COLUMN3 DATE);
```

For full details about defining tables and columns, refer to the relevant chapters in Part Three, ***A Database and its Objects***.

Domains

In Firebird you can pre-package a column definition, with a data type and a “template set” of attributes, as a **domain**. Once a domain is defined and committed, it is available for use in any table in your database, just as if it were a data type in its own right. In Firebird 2.5+, a domain can be used to declare the data type of a variable in a PSQL module, although it is not possible in older versions.

Columns based on a domain definition inherit all attributes of the domain—its data type, along with any number of optional, extra attributes, including a default value for inserts, validation constraints, character set and collation order.

Any attribute except the data type can be overridden when the domain is used to define a column in a table definition, by replacing an attribute with a different, compatible attribute or by adding an attribute. Thus, for example, it is possible to define a domain with a complex set of attributes, not including NOT NULL, which can be made nullable in some usages and NOT NULL in others.

For full details about defining, maintaining and using domains, refer to Chapter 11, [Domains](#), later in this part.

Converting Data Types

Normally, you must use compatible data types to perform arithmetic operations, or to compare data in search conditions. If you need to perform operations on mixed data types, or if your programming language uses a data type that is not supported by Firebird, then data type conversions must be performed before the database operation can proceed.

Implicit type conversion

Dialects 1 and 3 behave differently with regard to implicit type conversion. This will be an issue if you wish to convert an existing database to Dialect 3 and update its supporting applications.

- In dialect 1, for some expressions, Firebird automatically converts the data to an equivalent data type (an implicit type conversion). The CAST() function can also be used, although it is optional in most cases.
- In dialect 3, the CAST() function is required in search conditions to explicitly translate one data type into another for comparison purposes.

For example, comparing a DATE or TIMESTAMP column to '12/31/2012' in Dialect 1 causes the string literal '12/31/2012' to be converted implicitly to a DATE entity.

```
SELECT * FROM TABLE_A
WHERE START_DATE < '12/31/2012';
```

In Dialect 3:

```
SELECT * FROM TABLE_A
WHERE START_DATE < CAST('12/31/2012' AS DATE);
```

An expression mixing integers with string digits in Dialect 1 implicitly converts the string to an integer if it can. In the following expression:

```
3 + '1'
```

Dialect 1 automatically converts the character '1' to a smallint for the addition whereas Firebird dialect 3 returns an error. It requires an explicit type conversion:

```
3 + CAST('1' AS SMALLINT)
```

Both dialects will return an error on the next statement, because Firebird cannot convert the character “a” to an integer:

```
3 + 'a'
```

Both dialects will happily perform an explicit type conversion for concatenating:

```
SQL> select 1||'3' from rdb$database;
```

```
CONCATENATION
=====
13
```

Explicit type conversion: CAST()

When Firebird cannot do an implicit type conversion, you must perform an explicit type conversion using the CAST() function. Use CAST() to convert one data type to another inside a SELECT statement, typically, in the WHERE clause to compare different data types. The syntax is:

```
CAST (value | NULL AS data type)
```

You can use CAST() to compare columns with different data types in the same table, or across tables. For example, you can cast between properly formed strings and date/time types and between various number types. For detailed information about casting between specific types, refer to the chapter in this part of the guide that deals with the data types in question.

Changing column and domain definitions

In both dialects, you can change the data type of a column in tables and domains. If you are migrating a database from another RDMBS, this can be useful. Certain restrictions apply when altering the data type:

- 1 Firebird does not allow the data type of a column or domain to be altered in a way that might result in data loss. For example the number of characters in a column is not allowed to be smaller than the largest value in the column.
- 2 Converting a numeric data type to a character type requires a minimum length for the character type as listed in Table 6.3, below.

Table 6.3 Minimum character lengths for numeric conversions

Data type	Minimum length for character type
BigInt	19 (or 20 for signed numbers)
Decimal	20
Double precision	22
Float	13
Integer	10 (11 for signed numbers)
Numeric	20 (or 21 for signed numbers)
SmallInt	6

Altering data type of a column

Use the ALTER COLUMN clause of the ALTER TABLE statement with the TYPE keyword. For example:

```
ALTER TABLE table1 ALTER COLUMN field1 TYPE char(20);
```

For more information about altering columns in a table, refer to the topic [Altering Tables](#) in Chapter 15.

Altering data type of a domain

Use the TYPE clauselet of the ALTER DOMAIN statement to change the data type of a domain. For example:

```
ALTER DOMAIN MyDomain TYPE VARCHAR(40);
```

For more information about altering the attributes of a domain, refer to [Changing a Domain Definition](#) in Chapter 11.

Valid Conversions

Table 6.4 shows the data type conversions that are possible.

Table 6.4 Valid data type conversions using ALTER COLUMN and ALTER DOMAIN

Convert TO ► FROM ▼	ARRAY	BIGINT	BLOB	CHAR	DATE	DECIMAL	DOUBLE	FLOAT	INTEGER	NUMERIC	TIMESTAMP	TIME	SMALLINT	VARCHAR
Array														
BigInt				■		■	■							■
BLOB														
Char				■										■
Date				■	■									
Decimal				■		■				■				■
Double				■			■	■						■
Float				■			■	■						■
Integer		■		■		■				■				■
Numeric				■						■				■
Timestamp				■	■						■	■		
Time				■										
Smallint		■		■		■	■	■	■	■			■	■
Varchar				■										■

Keywords Used for Specifying Data Type

The keywords for specifying the data type in DDL statements is provided here for quick reference. For exact syntax, refer to the relevant chapter in this part of the guide for the data type in question and to Chapters 11, [Domains](#) and 15, [Tables](#).

```
--
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DATE | TIME | TIMESTAMP} [<array_dim>]
| {DECIMAL | NUMERIC} [ (precision [, scale])] [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
[<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE int | subtype_name] ] [SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [(seglen [, subtype])]
```

also supported, from v.2.1 onward:

```
| TYPE OF <domain-name>
```

and, from v.2.5 on:

```
| TYPE OF COLUMN <table-name.column-name>
```

For details, see the Arguments notes for [CAST\(\)](#) in Chapter 20, *Expressions and Predicates*:

Demystifying NULL

NULL can be quite a mystery for folk who have previously worked with desktop database systems that conveniently swallow NULLs by storing them as “zero values”: empty strings, 0 for numerics, false for logicals, and so on. In SQL, any data item in a nullable column—that is, a column that does not have the NOT NULL constraint—will be stored with a NULL token if no value is ever provided for it in a DML statement or through a column default.

All column definitions in Firebird default to being nullable. In triggers, procedures and procedural blocks, variables contain NULL until a value is assigned to them. Unless your database has been consciously designed to prevent NULLs from being stored, all of your data manipulation work needs to be prepared to encounter and manage them.



If you are reading this chapter as a newcomer to SQL, then some terminology in this topic might baffle you. Don't worry—you will meet them all again. For now, the important thing is to know and understand how NULL works.

NULL in expressions

NULL is not a value, so it cannot be “equal to” any value. For example, a predicate such as

```
WHERE (COL1 = NULL)
```

will return an error because the equivalence operator (“=”) is not valid for NULLs. NULL is a *state* and the correct predictor for a NULL test is *IS NULL*. The corrected predicate for the failed test above would be

```
WHERE (COL1 IS NULL)
```

You can also test for NOT NULL:

```
WHERE (COL1 IS NOT NULL)
```

Two NULLs are not equal to each other, either. When constructing expressions, be mindful of the cases when a predicate might resolve to

```
WHERE <NULL result> = <NULL result>
```

because false is always the result when two NULLs are compared.

An expression like

```
WHERE COL1 > NULL
```

will fail because an arithmetic operator is not valid for NULL.

NULL in calculations

In an expression where a column identifier “stands in” for the current value of a data item, a NULL operand in a calculation will produce NULL as the result of the calculation, e.g.

```
UPDATE TABLEA
  SET COL4 = COL4 + COL5;
```

will set COL4 to NULL if COL5 is NULL.

In aggregate expressions using operators like SUM() and AVG() and COUNT(<specific_column_name>), rows containing NULL in the targeted column are ignored for the aggregation. AVG() forms the numerator by aggregating the non-null values and the denominator by counting the rows containing non-null values.

Gotchas with True and False

Semantically, if a result is neither false nor true, it should be returned as “unknown”. However, in SQL, assertions resolve as either “true” or “false”—an assertion that does not evaluate as “true” shakes out as “false”.

The “IF” condition implicit in search predicates can trip you up when the NOT predicate is used on an embedded assertion, i.e.

```
NOT <condition evaluating to false> evaluates to TRUE
```

whereas

```
NOT <condition evaluating to null> evaluates to NULL
```

To take an example of where our assumptions can bite, consider

```
WHERE NOT (COLUMNA = COLUMNB)
```

If both COLUMNA and COLUMNB have values and they are not equal, the inner predicate evaluates to false. The assertion NOT(FALSE) returns true—the flip-side of false.

However, if either of the columns is NULL, the inner predicate evaluates to NULL, standing for the semantic meaning “unknown” (“not proven”, “unknowable”). The assertion that is finally tested is NOT(NULL) and the result returns NULL. Take note also that NOT(NULL) is not the same as IS NOT NULL—a pure binary predicate that never returns “unknown”.



The lesson in this is to be careful with SQL logic and always to test your expressions hard. Cover the null conditions and, if you can do so, avoid NOT assertions altogether in nested predicates.

Setting a value to NULL

A data item can be made NULL only in a column that is not subject to the NOT NULL constraint—refer to the topic [The NOT NULL Constraint](#) in Chapter 15, **Tables**.

In an UPDATE statement the assignment operator is "=":

```
UPDATE FOO SET COL1= NULL
      WHERE COL2 = 4;
```

In an INSERT statement, pass the keyword NULL in place of a value:

```
INSERT INTO FOO (COL1, COL2, COL3)
      VALUES (1, 2, NULL);
```

In this case, NULL overrides any default set for COL3, because the data content for it is set explicitly.

Columns defaulting to NULL

In Firebird, all columns are created nullable by default. What this means is that, unless you apply a NOT NULL constraint to a column, NULL is treated as valid content.



If you came from MS SQLServer or another system that has low compliance with standards, you might expect things to be the other way around, where all columns are constrained as NOT NULL unless explicitly defined as NULLABLE.

This characteristic provides another way to cause NULL to be stored by an INSERT statement since, if the nullable column is omitted from the input list, the engine will set it to NULL.

For example, the following statement has the same effect as the previous example, as long as no default is defined for the column COL3:

```
INSERT INTO FOO (COL1, COL2)
      VALUES (1, 2);
```

Assigning NULL to variables

In PSQL (procedural language), use the "=" symbol as the assignment operator when assigning NULL to a variable and use IS [NOT] NULL in the predicate of an IF test:

```
...
DECLARE VARIABLE foobar integer;
...
IF (COL1 IS NOT NULL) THEN
    FOOBAR = NULL;
...
```


CHAPTER

7

NUMBER TYPES

Firebird supports both fixed (exact precision) decimal and floating-point (approximate precision) number types. Fixed decimal types are the zero-scaled integer types SMALLINT, INTEGER and, in dialect 3, BIGINT, and two nearly identical scaled numeric types, NUMERIC and DECIMAL. The two floating-point types are FLOAT (single precision) and DOUBLE PRECISION.

Firebird does not support an unsigned integer type.

Numerical Limits

Table 7.1 below shows the numerical limits of each number type in Firebird:

Table 7.1 Limits of Firebird number types

Number type	Minimum	Maximum
SMALLINT	−32,768	32,767
INTEGER	−2,147,483,648	2,147,483,647
BIGINT	−2 ⁶³	2 ⁶³ −1
(for masochists)	−9223372036854775808	9223372036854775807
NUMERIC †	varies	varies
DECIMAL †	varies	varies
FLOAT		
Positive	1.175 x 10 ^{−38}	3.402 x 10 ³⁸
Negative	−3.402 x 10 ³⁸	
DOUBLE PRECISION ¹		
Positive	2.225 x 10 ^{−308}	1.797 x 10 ³⁰⁸
Negative	−1.797 x 10 ³⁰⁸	

- [†] Limits for NUMERIC and DECIMAL types vary according to storage type and scale.
The limits will always fall within the range for the internal type in which they are stored.²
- ¹ Precision describes the number of significant digits, ignoring trailing or leading zeroes, that can be stored in the data type with overflow or loss of information.
- ² Stored type is SMALLINT, INTEGER or BIGINT, according to the declared precision.

Points About Points

In SQL, the decimal point symbol is always a full-stop (a.k.a. “period”, “punkt”, “point”, etc.). Using a comma or any other character to represent a decimal point will cause a type exception.

If you are in a country where the convention is to group numerals to the left of the decimal point using full-stops, e.g., 2.100 representing “two thousand one hundred”, be aware of the risk of exponential errors if you fail to protect your applications from inputs that allow this convention.

Operations on Number Types

- Operations supported include
- Comparisons** using the standard relational operators (=, <, >, >=, <=, <> or !=).
 - String comparisons** using SQL operators such as CONTAINING, STARTING WITH, and LIKE are possible. In these operations, the numbers are treated as strings. For more information about these operators, refer to the Chapter 21, Expressions and Predicates.
 - Arithmetic operations** The standard dyadic arithmetic operators (+, - * and /) can be applied.
 - Conversions** Firebird automatically converts between fixed numeric, floating point and character types when performing operations on mixed data types. When an operation involves a comparison or arithmetical operation between numeric data and non-numeric data types, data are first converted to a numeric type, then operated on.
 - Sorts** By default, a query retrieves rows in the exact order that it finds them in the table, which is likely to be unordered. You can sort rows on integer columns using the ORDER BY clause of a SELECT statement in descending or ascending order. If numbers are stored or cast as character types, the sort order will be alphanumeric, not numeric, e.g. 1 – 10 – 11 ... 19 - 2

Integer Types

All integer types are *signed exact numerics* having a scale of zero.

Notation

In all versions, integer values are submitted in decimal notation. From v.2.5 onwards, you have the option of submitting them in either decimal or hexadecimal notation.

Numbers with 1–8 hex digits are interpreted as 32-bits, while numbers with 9–16 hex digits as interpreted as 64 bits.

For example, the following two statements are equivalent:

```
INSERT INTO ATABLE (ID, PASSPHRASE)
VALUES (1273, 'ELEPHANT IN THE ROOM');

--
INSERT INTO ATABLE (ID, PASSPHRASE)
VALUES (0x4F9, 'ELEPHANT IN THE ROOM');
```

Also, the A-F digits are case-insensitive:

```
INSERT INTO ATABLE (ID, PASSPHRASE)
VALUES (0x4f9, 'ELEPHANT IN THE ROOM');
```

Types

Firebird supports three named ranges of precision as integer data types: SMALLINT, INTEGER and BIGINT.

- SMALLINT is a signed short integer with a range from –32,768 to 32,767
- INTEGER is a signed long integer with a range from –2,147,483,648 to 2,147,483,647
- BIGINT is a signed 64-bit integer with a range from 2⁶³ to 2⁶³ - 1. Not available in Dialect 1.

v.1.0.x For Firebird 1.0.x, dialect 3, you need to declare a 64-bit integer as NUMERIC(18,0) or DECIMAL(18,0). It is always valid to use this syntax for the integer types, optionally omitting the second (scale) argument.

Scale, precision and the operations that can be performed on fixed types are discussed in more detail in the next section, **Fixed decimal (scaled) types**.

The next two statements create a domain and a column with the SMALLINT and INTEGER data types respectively:

```
CREATE DOMAIN RGB_RED_VALUE AS SMALLINT;

/**/
CREATE TABLE STUDENT_ROLL (
    STUDENT_ID INTEGER,
    ...);
```

Each of these statements creates a domain which is a 64-bit integer:

```
CREATE DOMAIN IDENTITY BIGINT CHECK (VALUE >=0); /* Firebird 1.5 and higher */
CREATE DOMAIN IDENTITY NUMERIC(18,0) CHECK (VALUE >=0);
```

SMALLINT

A two-byte integer providing compact storage for whole numbers with a limited range. For example, SMALLINT would be suitable for storing the value of colors in the RGB scale, as in the domain example above.

SMALLINT is often used to define a two-state Boolean, usually 0=False, 1=True. An example of this usage can be found in Chapter 11, **Domains**, in the topic [Defining a BOOLEAN Domain](#).

INTEGER

A four byte integer. You can store integers in BigInt columns without casting.



In Dialect 1, generators (see below) generate integers.

BIGINT, NUMERIC(18,0)

Available in Dialect 3 only, this is an 8-byte integer, useful for storing whole numbers with very low and high ranges. In Dialect 3, generators or sequences (see below) generate BigInt numbers.



In a Dialect 1 database, Firebird rejects any attempt to define a domain or column as BIGINT. It will not raise an exception if you try to define a NUMERIC(18,0) domain or column, but will silently define a DOUBLE PRECISION column instead. Because of precision problems with matching floating point numbers, be careful to avoid accidentally using NUMERIC(18,0) in a Dialect 1 database to define a column which is going to be used as a key for searches or joins.

Autoincrement or Identity type

Firebird has no autoincrement or identity type such as you find in some other database management systems. What it does have is a number-generator engine and the ability to maintain independent, named series of BIGINT numbers. Each series is known as a **generator** or **sequence**.

Generators are ideal for populating an automatically incrementing unique key or a stepping serial number column or other series. The benefit of generator values is that they are guaranteed to be unique and they operate outside transaction control. This absolutely assures the integrity of number sequences, provided the generators are not tampered with by humans!

Details about creating and working with generators (sequences) are in Chapter 12, **Database Basics**, in the topic Generators (Sequences). A technique for using them to implement and maintain primary keys and other automatically incrementing series is described in Chapter 30, in the topic Implementing Auto-Incrementing Keys.

Integer/integer division

Using the example above, the following query in dialect 3 returns the integer 0 because each operand has a scale of 0, so the sum of the scales is 0:

```
SELECT i1/i2 FROM t1
```

In dialect 1, in line with many desktop-style data management products, dividing one integer by another produces a floating-point result of DOUBLE PRECISION type:

```
SELECT 1/3 AS RESULT FROM RDB$DATABASE
```

yields .333333333333333

Although this dialect 1 rule is intuitive for language programmers and calculator users, it does not conform with the SQL-92 standard. Integer types have a scale of 0 which, for consistency, requires that the result (quotient) of any integer/integer operation conform with the scaling rules for fixed numerics and produce an integer.

Dialect 3 conforms with the standard and truncates the quotient of integer/integer division operations to integer. Hence, it can trap the unwary:

```
SELECT 1/3 AS RESULT FROM RDB$DATABASE
```

yields 0

When you need to preserve sub-integer parts in the result (quotient) of integer/integer divisions in dialect 3, be sure that you either scale one of the operands or include a “factor” in the expression which will guarantee a scaled result.

Examples

```
SELECT (1 + 0.00)/3 AS RESULT FROM RDB$DATABASE
```

yields .33

```
SELECT (5 * 1.00)/2 AS RESULT FROM RDB$DATABASE
```

yields 2.50

Fixed Decimal (Scaled) Types

Fixed decimal types allow the management of numbers which need to be expressed with a fractional portion that is exact to a specific number of decimal places, or scale. Typically, you need scaled types for money values and any other numbers that result from counting or performing arithmetic on whole units and parts of units.

Firebird provides two fixed decimal or scaled data types: NUMERIC and DECIMAL. Broadly, either scaled type is declared as TYPE(P, S) with P indicating precision (number of significant digits) and S indicating scale (number of decimal places, i.e. digits to the right of the decimal point symbol).

According to the SQL-92 standard, both NUMERIC and DECIMAL constrain the stored number to be within the declared scale. The difference between the two types is in the way the precision is constrained. Precision is exactly “as declared” for a column of type NUMERIC, whereas a DECIMAL column can accept a number whose precision is at least equal to that which was declared, up to the implementation limit.

NUMERIC and DECIMAL types, as implemented in Firebird, are identical except when precision is less than 5. Both types effectively conform with the standard DECIMAL type. NUMERIC is thus not compliant with SQL-92.

The predictability of results from multiplying and dividing fixed-point numbers favors choosing them for storing money values. However, because fixed-point types have a finite “window” in which numbers may be accommodated, they become prone to overflow/underflow exceptions near their upper and lower limits.

In countries where the unit of currency represents a small value, number limits should be considered carefully with regard to huge totals. For example, the following statement applies a tax rate (DECIMAL(5,4)) to a net profit (NUMERIC(18,2)):

```
UPDATE ATABLE
SET INCOME_AFTER_TAX = NET_PROFIT - (NET_PROFIT * TAX_RATE);
```

Let the tax rate be 0.3333

Let the net profit be 1234567890123456.78

Result:

```
ISC ERROR CODE:335544779
```

Integer overflow. The result of an integer operation caused the most significant bit of the result to carry.

Internal Storage

Internally, Firebird stores the scaled number as a `SmallInt` (16 bits), `Integer` (32 bits) or `BigInt` (64 bits) type, according to the size of precision declared. Its declared precision is stored, along with the declared scale negated to a sub-zero scale-factor, representing an exponent of 10. When the number is referred to for output or a calculation, it is produced by multiplying the stored integer by $10^{\text{scale_factor}}$.

For example, for a column defined as `NUMERIC(4,3)`, Firebird stores the number internally as a `SMALLINT`. If you insert the number 7.2345, Firebird silently rounds the rightmost digit (4) and stores a 16-bit integer 7235 and a `scale_factor` of -3. The number is retrieved as 7.235 ($7235 * 10^{-3}$).

Numeric data type

Format:

`NUMERIC(p,s)`

For example, `NUMERIC(4,2)` formally defines a number consisting of up to 4 digits, including two digits to the right of the decimal point. Thus, the numbers 89.12 and 4.321 will be stored in a `NUMERIC(4,2)` column as 89.12 and 4.32 respectively. In the second example, the final 1-3 is out of scale and is simply dropped.

It is possible to store in this column a number of greater precision than that declared. The maximum here would be 327.67, i.e. a number with a precision of 5. Because the database stores the actual number as a `SMALLINT`, numbers will not begin to cause overflow errors until the internally-stored number is more than 32,767 or less than -32,768.

Decimal data type

Format:

`DECIMAL(p,s)`

Similarly to Numeric, `DECIMAL(4,2)` formally defines a number consisting of at least 4 digits, including two digits to the right of the decimal point. However, because Firebird stores `DECIMAL` of precision 4 and below as `INTEGER`, this type could, in a `DECIMAL(4,1)` column, potentially store a number as high as 214,748,364.7 or as low as -214,748,364.8 without causing an overflow error.

Dialect 1 Databases

Exact numerics can be confusing because of the slightness of difference between the two types. If you have the misfortune to be still dealing with a dialect 1 database, the confusion is worse the dialect of the database affects the range of precision available. The following Table 7.2 may assist as a summary guide for the precision and scale you need to specify for your various exact numeric requirements.

Table 7.2 Range and storage type of Firebird `NUMERIC` and `DECIMAL` types

Precision	Type	Dialect 3	Dialect 1
1 to 4	<code>NUMERIC</code>	<code>SMALLINT</code>	<code>SMALLINT</code>
	<code>DECIMAL</code>	<code>INTEGER</code>	<code>INTEGER</code>
5 to 9	<code>NUMERIC</code> and <code>DECIMAL</code>	<code>INTEGER</code>	<code>INTEGER</code>

Precision	Type	Dialect 3	Dialect 1
10 to 18	NUMERIC and DECIMAL	BIGINT	DOUBLE PRECISION [†]

[†] Exact numerics with precision higher than 9 can be declared in a dialect 1 database without an exception occurring. However, numbers will be stored as DOUBLE PRECISION and be subject to the same precision restrictions as any floating point numbers.

Converted databases

If a dialect 1 database is restored using *ghak*, numeric fields defined with precision higher than 9 will remain implemented as DOUBLE PRECISION. Although they will still appear as they were originally defined, e.g. NUMERIC(15,2), they will continue to be stored and used in calculations as DOUBLE PRECISION.

For information about converting dialect 1 databases to dialect 3, refer to the topic [Dialect 1 Databases](#) in Chapter 5, *Migration Notes*.

Special restrictions in static SQL

The host languages of embedded applications cannot use or recognize small precision NUMERIC or DECIMAL data types with fractional portions when they are stored internally as SMALLINT or INTEGER types. To avoid this problem, in any database that is going to be accessed via embedded applications (ESQL):

- 1 Do not define NUMERIC or DECIMAL columns or domains of small precision in a dialect 1 database. Either store an integer and have your application code deal with scale; or use DOUBLE PRECISION and apply a suitable rounding algorithm for calculations.
- 2 In a dialect 3 database, define NUMERIC and DECIMAL columns or domains of any size using a precision of at least 10, to force them to be stored internally as BIGINT. Specify a scale if you want to control the precision and scale. Apply CHECK constraints if you need to control the ranges.

Behaviour of fixed types in operations

It may be helpful to know how fixed types are treated during arithmetic operations. As before, some dialectal differences do exist that may affect your choice of data type.

Division

When performing division on fixed types, dialects 1 and 3 behave differently.

In dialect 3, where both operands are of a fixed numeric type, Firebird adds together the scales of both operands to determine the scale of the result (quotient). The quotient has a precision of 18. When designing queries with division expressions, therefore, be aware that quotients will always have more precision than either of the operands and take precautions where precision could potentially overflow the maximum of 18.

Dialect 1 In dialect 1, division always produces a quotient of DOUBLE PRECISION type.

Examples

In dialect 3, the quotient of dividing a DECIMAL(12,3) by a DECIMAL(9,2) is a DECIMAL(18,5). The scales are added together:

```
SELECT 11223344.556/1234567.89 FROM RDB$DATABASE  
yields 9.09090
```

Compare the difference in the quotient when the same query is run in dialect 1. The first operand is treated as a DOUBLE PRECISION number because its precision (12) is higher than the maximum for a dialect 1 scaled type. The quotient is also a double. The result is 9.09090917308727 because of the errors inherent in floating-point types.

From the following table defined in dialect 3, division operations produce a variety of results:

```
CREATE TABLE t1 (  
    i1 INTEGER,  
    i2 INTEGER,  
    n1 NUMERIC(16,2),  
    n2 NUMERIC(16,2));  
COMMIT;  
INSERT INTO t1 VALUES (1, 3, 1.00, 3.00);  
COMMIT;
```

The following query returns the NUMERIC(18,2) value 0.33, since the sum of the scales 0 (operand 1) and 2 (operand 2) is 2:

```
SELECT i1/n2 from t1
```

The following query returns the NUMERIC(18,4) value 0.3333, since the sum of the two operand scales is 4.

```
SELECT n1/n2 FROM t1
```

Dialect 1 database with dialect 3 client

A dialect 1 database that is opened with a dialect 3 client may cause some surprises with respect to integer division. When an operation does something that causes a CHECK condition to be checked, or a stored procedure to be executed, or a trigger to fire, the processing that takes place is based on the dialect under which the check, stored procedure, or trigger was defined, not the dialect in effect when the application causes the check, stored procedure, or trigger to be executed.

For example, suppose that a dialect 1 database has a table MYCOL1 INTEGER and MYCOL2 INTEGER with a table definition includes the following CHECK condition that was defined when the database was dialect 1:

```
CHECK(MYCOL1 / MYCOL2 > 0.5)
```

Now suppose that a user starts *isql*, or another application, and sets the dialect to 3. It tries to insert a row into the converted database:

```
INSERT INTO MYTABLE (COL1, COL2) VALUES (2,3);
```

Because the CHECK constraint was defined in dialect 1, it returns a quotient of 0.6666666666666667, and the row passes the check condition.

The reverse is also true. If the same CHECK constraint were added to the dialect 1 database through a dialect 3 client, dialect 3 arithmetic is stored for the constraint. The INSERT statement above would fail because the check would return a quotient of 0, violating the constraint.



There's a moral to this: use dialect 3 databases and always connect as dialect 3. If you intend to use Firebird then upgrade any existing databases to dialect 3—then rest easy and avoid a raft of nasty surprises.

Multiplication

As with division, if both multiplier operands are exact numeric, multiplying the operands produces an exact numeric with a scale equal to the sum of the scales of the operands. For example:

```
CREATE TABLE t1 (
  n1 NUMERIC(9,2),
  n2 NUMERIC(9,3));
COMMIT;
INSERT INTO t1 VALUES (12.12, 123.123);
COMMIT;
```

The following query returns the number **1492.25076** because n1 has a scale of 2 and n2 has a scale of 3. The sum of the scales is 5.

```
SELECT n1*n2 FROM t1 as product
```

In Dialect 3, the precision of the result of multiplying a fixed numeric by a fixed numeric is 18. Precautions must be taken to ensure that potential overflows will not result from the propagation of scale in multiplication.

In Dialect 1, if the propagation of scale caused by the calculation would produce a result with precision higher than 9, the result will be DOUBLE PRECISION.

Addition and subtraction

If all operands are exact numeric, adding or subtracting the operands produces an exact numeric with a scale equal to that of the largest operand. For example:

```
CREATE TABLE t1 (
  n1 NUMERIC(9,2),
  n2 NUMERIC(9,3));
COMMIT;
INSERT INTO t1 VALUES (12.12, 123.123);
COMMIT;
SELECT n1 + n2 FROM t1;
```

The query returns **135.243**, taking the scale of the operand with the larger scale.

Similarly, the following query returns the numeric **-111.003**:

```
SELECT n1 - n2 FROM t1;
```

In dialect 3, the result of any addition or subtraction is a NUMERIC(18,3). In dialect 1, it is a NUMERIC(9,3).

Numeric input and exponents

Any numeric string in dynamic SQL (DSQL) that can be stored as a DECIMAL(18,S) is evaluated exactly, without the loss of precision that might result from intermediate storage as a DOUBLE PRECISION. The DSQL parser can be forced to recognize a numeric string as floating point by the use of scientific notation, i.e. appending the character “e” or “E” followed by an exponent, which can be zero.

For example, DSQL will recognize **16.92** as a scaled exact numeric and passes it to the engine in that form. On the other hand, it will treat **16.92E0** as a floating-point value.

Floating Point Types

Floating-point types employ a sort of “sliding window” of precision appropriate to the scale of the number. By the nature of “float” types, the placement of the decimal point is not a restriction—it is valid to store, in the same column, one value as 25.33333, and another as 25.333. The values are different and both are acceptable.

Define a floating-point column when you need to store numbers of *varying scale*. The general rule of thumb for choosing a floating-point, rather than a fixed decimal type, is “use them for values you measure, not for values you count”. If a floating-point column or variable must be used to store money, you should pay careful attention both to rounding issues and to testing the results of calculations.

Floating point numbers can be used to represent a value over a much larger range than is possible with plain or scaled integers. For example, the FLOAT type can carry values with a magnitude as large as 3.4E38 (that’s 34 followed by 37 zeros), and as small as 1.1E-38 (that’s 11 preceded by 37 zeros and a then a decimal point).

The breadth of range is achieved by a loss in exactness. A floating-point carries an approximate representation of its value that is accurate for a specific number of digits (its precision), according to the current magnitude (scale). It can not carry a value close to either extreme of its range.

The floating-point value carries more information than the stated number digits of precision. The FLOAT type, for example, is said to have a precision of 7 digits but its “assumed accurate” precision is 6 digits. The last part is an approximation providing additional information about the number, such as an indicator for rounding and some more things that are important when arithmetic is performed on the number.

For example, a FLOAT can carry the value 1000000000 (1,000,000,000, or 10⁹). The FLOAT “container” sees this value as (effectively) 100000*E4. (This is illustrative only—an authoritative exposition of floating point implementation is beyond the scope of this book and seriously beyond the author’s reach!) If you add 1 to the value of the FLOAT, the additional information carried in the seventh digit is ignored because it is not significant in terms of the number’s current magnitude and the precision available. If you add 10,000—a value that is significant to the magnitude of the number currently stored in the FLOAT—it can represent the result as 100001*E4.

Even values within the available precision of the float may not always store an exact representation. Values such as 1.93, or even 123 may be represented in storage as a value that is very close to the specific number. It is close enough that, when the floating point number is rounded for output, it will display the value expected and, when it is used in calculations, the result is a very close approximation to the expected result.

The effect is that, when you perform some calculation that should result in the value 123, it may only be a very close approximation to 123. Exact comparisons (equality, greater than, less than, and so on) between two floating-point numbers, or between a floating-point number and zero, or a floating-point number and a fixed type, thus can not be depended on to produce the expected results.

For this reason, do not consider using floating-point columns in keys or applying uniqueness constraints to them. They will not work predictably for foreign key relationships or joins.

For comparisons, test floating-point values as being BETWEEN some acceptable range rather than testing for an exact match. The same advice applies when testing for 0—

choose a range appropriate to the magnitude and signing of your data that between zero and a near-zero value, or between two suitable near-zero values.

Dialect 1 In a dialect 1 database, the need to store numeric data values having a wider range than the limits of a 32-bit integer may force the choice of a DOUBLE PRECISION type. Dialect 1 limitations also require the use of floating-point numbers for all reals if the database is going to be accessed by an embedded (ESQL) application.

Supported Float Types

Firebird provides two floating-point or approximate numeric data types, FLOAT and DOUBLE PRECISION, differing only in the limit of precision.

FLOAT

FLOAT is a 32-bit floating-point data type with a limit of approximately 7 digits of precision—assume 6 digits for reliability. A 10-digit number 25.33333312 inserted into a FLOAT column is stored as 25.33333. Range from -3.402×10^{38} to 3.402×10^{38} . The smallest positive number it can store is 1.175×10^{-38} .

DOUBLE PRECISION

DOUBLE PRECISION is a 64-bit floating-point data type with a limit of approximately 15 digits of precision. Range from -1.797×10^{308} to 1.797×10^{308} . The smallest positive number it can store is 2.225×10^{-308} .

Arithmetic mixing fixed and floating-point types

When a dyadic operation (addition, subtraction, multiplication, division) involves an exact numeric operand and a floating-point operand, the result will be a DOUBLE PRECISION type.

The next statement creates a column, PERCENT_CHANGE, using a DOUBLE PRECISION type:

```
CREATE TABLE SALARY_HISTORY (
...
    PERCENT_CHANGE DOUBLE PRECISION
    DEFAULT 0 NOT NULL
    CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
...
);
```

The following CREATE TABLE statement provides an example of how the different numeric types can be used: an INTEGER for the total number of orders, a fixed DECIMAL for the dollar value of total sales, and a FLOAT for a discount rate applied to the sale.

```
CREATE TABLE SALES (
...
    QTY_ORDERED INTEGER
    DEFAULT 1
    CHECK (QTY_ORDERED >= 1),
    TOTAL_VALUE DECIMAL (9,2)
```

```
DISCOUNT FLOAT
  DEFAULT 0
  CHECK (DISCOUNT >= 0 AND DISCOUNT <= 1)
);
```

DATE AND TIME TYPES

Firebird supports SQL standard DATE, TIME, and TIMESTAMP data types.

Dialect 1 In dialect 1, only one date type is supported, a timestamp-like implementation which, although named DATE, is not interchangeable with the dialect 3 DATE type.

Choices for Date and Time Values

Each of the three options for storing date and time values has its own, distinct usages.

DATE

DATE stores the date alone, with no time portion – “date-only”. Storage is a 32-bit signed longword. Storable dates range from January 1, 0001 to December 31, 9999. SQLTYPE is *isc_date*.

Choose DATE only if you are certain that nobody will ever want to use the date in a context narrower than a day. Examples are birthdays—as long as the time of birth is not significant—and publication dates. If a more granular record is not in the current requirements but might be needed in future, choose TIMESTAMP instead.

Dialect 1 There is no “date-only” type in dialect 1. To store a “date-only” value, pass a valid date and time literal with “00:00:00.0000” in the time portion. For more information about date and time literals, keep reading.



If you are using isql to examine dialect 1 dates, you can toggle on/off the display of the time-part of date output using the isql command SET TIME; “Off” is the default.

TIMESTAMP

The `TIMESTAMP` data type is made of two 32-bit longword portions, storing a date and a time. It is stored as two 32-bit longwords, equivalent to the `DATE` type in dialect 1. `SQLTYPE` is `isc_timestamp`.

`TIMESTAMP` is the general purpose type for recording both dates and times. It is the recommended type for usages where one of the following possibilities applies:

- either the date or the time of the record, or both, are likely to be relevant
- intervals of time might need to be calculated
- dates or times might need to be compared across different time zones

Fractions of seconds

Fractions of seconds, if stored, are in ten-thousandths of a second for all date and time types.

TIME

A column or variable of type `TIME` stores the time of day, with no date portion – “time-only”. Storage is a 32-bit unsigned longword. Storable times range `00:00` to `23:59:59.9999`. `SQLTYPE` is `isc_time`.

Dialect 1 If time of day needs to be stored, extract the hours, minutes and seconds elements from the `DATE` data and convert it to a string. A suggested technique follows later in this chapter—refer to the topic `Combining EXTRACT(..) with other functions`.

Interval of time (time elapsed)

It is a common mistake to think that a `TIME` type can store an interval of time (time elapsed). It cannot. To calculate an interval of time, subtract the later of two date or time types from the earlier. The result will be a `DOUBLE PRECISION` number expressing the interval (time elapsed) in days for `TIMESTAMP` and `DATE` arguments or in seconds for `TIME` arguments.

Use regular arithmetic operations to convert days to hours, minutes or seconds, as required.

Suppose, for example, that the columns `STARTED` and `FINISHED` are both `TIMESTAMP`. To calculate and store the time elapsed in minutes into a `DOUBLE PRECISION` column `TIME_ELAPSED`:

```
UPDATE ATABLE
SET TIME_ELAPSED = (FINISHED - STARTED) * 24 * 60
WHERE ((FINISHED IS NOT NULL) AND (STARTED IS NOT NULL));
```



Calculating intervals between two `TIME` types has traps because a `TIME` value has no “day” context. Think about a time-clock that records the start time and end time of a worker’s shift. How would you handle the situation where a worker started the night shift at 10 p.m. and worked an eight-hour shift?

Date/Time Literals

Date literals are “human-readable” strings, enclosed in single quotation marks, that a Firebird server recognises as date or date-and-time constants for `EXTRACT` and other expressions, `INSERT`, and `UPDATE` operations and in the `WHERE` clause of a `SELECT` statement.

Specifically, date literals are used when supplying date constants to

- `SELECT`, `UPDATE` and `DELETE` statements, in the search condition of a `WHERE` clause
- `INSERT` and `UPDATE` statements, to enter date and time constants
- the `FROM` argument of the `EXTRACT` function.

Recognised Date/Time Literal Formats

The formats of the strings recognized as date literals are restricted. These formats are discussed below, using placeholders for the elements of the strings. Table 8.1 provides a key to the conventions used.

Table 8.1 Elements of date literals

Element	Representing
CC	Century - first two digits of a year segment, e.g. ‘20’ for the twenty-first century
YY	Year in century. Firebird always stores the full year value if the year is entered without the ‘CC’ segment, using a “sliding window” algorithm (see below) to determine which century to store.
MM	Month, evaluating to an integer in the range 1 to 12. In some formats, two digits are required.
MMM	Month, one of [JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC]. English month names fully spelt out (correctly) are also valid.
DD	Day of the month, evaluating to an integer in the range 1 to 31. In some formats, two digits are required. An invalid day-of-month number for the given month will cause an error.
HH	Hours, evaluating to an integer in the range 00 to 23. Two digits are required when storing a time portion.
NN	Minutes, evaluating to an integer in the range 00 to 59. Two digits are required when storing a time portion.
SS	Whole seconds, evaluating to an integer in the range 00 to 59. Two digits are required when storing a time portion.
[.]nnnn	Ten-thousandths of a second in the range zero to 9999. Optional for time portions, defaults to 0000. If used, four digits are required.

The recognised formats are shown in Table 8.2:

Table 8.2 Recognised date and time literal formats

Format	Dialect 3 DATE	Dialect 3 TIMESTAMP	Dialect 1 DATE
'CCYY-MM-DD' or 'YY-MM-DD'	Stores date only	Stores date and a time portion of 00:00:00	Stores date and a time portion of 00:00:00
'MM/DD/CCYY' or 'MM/DD/YY'	As above	As above	As above
'DD.MM.CCYY' or 'DD.MM.YY'	As above	As above	As above
'DD-MMM-CCYY' or 'DD-MMM-YY'	As above	As above	As above
'DD,MMM,CCYY' or 'DD,MMM,YY'	As above	As above	As above
'DD MMM CCYY' or 'DD MMM YY'	As above	As above	As above
'DDMMMCCYY' or 'DDMMYY'	As above	As above	As above
<i>Case-insensitive English month names fully spelt out are also valid in the MMM element. Correct spelling is shown in Table 8.3, below.</i>			
'CCYY-MM-DD HH:NN:SS.nnnn' or 'YY-MM-DD HH:NN:SS.nnnn' (“.nnnn” element is optional)	Stores date only: may need to be CAST as date. Time portion is not stored	Stores date and time.	Stores date and time.
'MM/DD/CCYYHH: NN:SS.nnnn' or 'MM/DD/YY HH:NN:SS.nnnn'	As above	As above	As above
'MM/DD/CCYYHH: NN:SS.nnnn' or 'MM/DD/YY HH:NN:SS.nnnn'	As above	As above	As above
'DD.MM.CCYYHH: NN:SS.nnnn' or 'DD.MM.YY HH:NN:SS.nnnn'	As above	As above	As above
'DD-MMM-CCYY HH:NN:SS.nnnn' or 'DD-MMM-YY HH:NN:SS.nnnn'	As above	As above	As above

The `TIMESTAMP` type accepts both date and time parts in a date literal. A date literal submitted without a time part will be stored with a time part equivalent to '00:00:00'.

The `DATE` type accepts only the date part. The `TIME` data type accepts only the time part.

Firebird's “sliding century window”

Whether the year part of a DATE or TIMESTAMP literal is submitted in SQL as ‘CCYY’ or ‘YY’, Firebird always stores the full year value. It applies an algorithm in order to deduce the ‘CC’ (century) part and it always includes the century part when it retrieves date types. Client applications are responsible for displaying the year as two or four digits.

To deduce the century, Firebird uses a sliding window algorithm. Its effect is to interpret a two-digit year value as the nearest year to the current year, in a range spanning the preceding 50 years and the succeeding 50 years.

For example, if the current year were 2012, two-digit year values would be interpreted thus:

Table 8.3 Deduction of year from two-digit year if current year is 2012

taking 2012 as an example of current year

Two-digit year	becomes year	Deduced from
98	1998	$(2012 - 1998 = 14) < (2098 - 2012 = 86)$
00	2000	$(2012 - 2000 = 12) < (2100 - 2012 = 88)$
45	2045	$(2012 - 1945 = 67) > (2045 - 2012 = 33)$
50	2050	$(2012 - 1950 = 62) > (2050 - 2012 = 38)$
62	1962	$(2012 - 1962 = 50) = (2062 - 2012 = 50)^{\ddagger}$
63	1963	$(2012 - 1963 = 49) < (2063 - 2012 = 51)$

[‡] The apparent equivalence of this comparison could be misleading. However, 1962 is closer to 2012 than is 2062 because all dates between 1962 and 1963 are closer to 2012 than all dates between 2062 and 2063.

Separators in non-US dates

Nothing causes more confusion for international users than Firebird's restricting the use of the forward slash date-part separator character (“/”) to only the US ‘MM/DD/CCYY’ format. Although almost all other countries use ‘DD/MM/CCYY’, Firebird will either record the wrong date or throw an exception with date literal using the ‘DD/MM/CCYY’ convention.

For example, the date literal ‘12/01/2012’ will always be stored with meaning ‘December 1, 2012’ and ‘14/01/2012’ will cause an out-of-range exception because there is no month 14.

Note that Firebird does not honor the Windows or Linux date locale format when interpreting date literals. Its interpretation of all-number date formats is decided by the separator character. When dot (.) is used as separator, Firebird interprets it as the non-US notation DD.MM, whereas with any other separator it assumes the US MM/DD notation. Outside the US date locale, your applications should enforce or convert locale-specific DD/MM/CCYY date input to a literal that replaces the forward slash with a period (dot) as the separator. ‘DD.MM.CCYY’ is valid. Other date literal formats may be substituted.

White space in date literals

Spaces or tabs can appear between elements. A date part must be separated from a time part by at least one space.

Quoting of date literals

Date literals must be enclosed in apostrophes (single-quotes) (ASCII 39/Unicode U+0027). Only single-quotes are valid.

Month literals

Month literals with correct English spellings

Cardinal Number	Abbreviated form	Full Month Name
	<i>Case-insensitive</i>	<i>Case-insensitive</i>
01	JAN	January
02	FEB	February
03	MAR	March
04	APR	April
05	MAY	May
06	JUN	June
07	JUL	July
08	AUG	August
09	SEP	September
10	OCT	October
11	NOV	November
12	DEC	December

Examples of date literals The 25th day of the sixth month (June) in the year 2012 can be represented in all of the following ways:

'25.6.2012'	'06/25/2012'	'June 25, 2012'	'25.jun.2012'
'6,25,2012'	'25,jun,2012'	'25jun2012'	'6-25-12'
'Jun 25 12'	'25 jun 2012'	'2012 June 25'	'20120625'
'25-jun-2012'	' '2012-jun-25'	'20120625'	'25 JUN 12'
'2012-06-25'	'2012,25,06'		

Pre-defined Date Literals

Firebird supports a group of pre-defined date literals: single-quoted English words that Firebird captures or calculates and interprets in the context of an appropriate date/time type. The literals 'TODAY', 'NOW', 'TOMORROW' and 'YESTERDAY' are interpreted according to Table 8.4 below.

Table 8.4 Predefined date literals

Date Literal	Dialect 3 type	Dialect 1 type	Meaning
'NOW'	TIMESTAMP	DATE	Server date and time that was current at the start of the DML operation. 'NOW' will be cast and stored correctly in dialect 3 DATE, TIME, and TIMESTAMP fields and in dialect 1 DATE fields. In versions prior to v.2.0., the sub-second portion is always stored as '.0000'. [†]
'TODAY'	DATE	DATE stored with a time part equivalent to '00:00:00'	Server date that was current at the start of the operation. If midnight is passed during the operation, the date does not change. Equivalent to the dialect 3 CURRENT_DATE context variable. Not valid in fields of TIME type.
'TOMORROW'	DATE	DATE stored with a time part equivalent to '00:00:00'	Server date that was current at start of the operation, plus one day. If midnight is passed during the operation, the date from which the 'TOMORROW' date is calculated does not change. Not valid in fields of TIME type.
'YESTERDAY'	DATE	DATE stored with a time part equivalent to '00:00:00'	Server date that was current at start of the operation, minus one day. If midnight is passed during the operation, the date from which the 'YESTERDAY' date is calculated does not change. Not valid in fields of TIME type.

[†] All is not lost, however. You can get the server timestamp to 1/10,000th of a second by using instead the external function *GetExactTimestamp(..)* from the Firebird UDF library, *fbudf*. For more information, refer to the entry *GetExactTimestamp (fbudf)* in Appendix I, *Internal and External Functions*.

Type-casting of Date/Time Literals

Whenever date literals—whether regular or pre-defined—are used in SQL in the context of corresponding date/time type columns or variables, the SQL parser can interpret them correctly without casting. However, in a small number of cases, where there is no typed value to which the parser can map the date literal, it treats any date literal as a string.

For example, it is perfectly valid to ask a SELECT query to return a constant that is unrelated to any database column. A common “hack” in Firebird to use the system table RDB\$DATABASE for just such a query—since this table has one and only one row and thus can always be relied upon to return a scalar set—to pick up a single context value from the server. The next two examples illustrate a typical use of this hack:

```
SELECT 'NOW' FROM RDB$DATABASE;
```

Because the query returns a constant, not a column value, its data type is interpreted as a char(3), 'NOW'.

```
SELECT '2.09.2012' FROM RDB$DATABASE;
```

returns a CHAR(9), '2.09.2012'.

To have the parser correctly interpret a date literal in conditions where the parser cannot determine the data type, use the CAST(..) function:

```
SELECT CAST('NOW' AS TIMESTAMP) FROM RDB$DATABASE;
SELECT CAST('2.09.2012' AS TIMESTAMP) FROM RDB$DATABASE;
```

We take a more detailed look at casting later, in the topic Operations Using Date and Time Values.

Date and Time Context Variables

Date and time context variables CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP return date and time values capturing the server time at the moment execution of the containing SQL statement begins.

Table 8.5 Date and time context variables

Variable	Dialect 3 type	Dialect 1 type	Meaning
CURRENT_TIMESTAMP	TIMESTAMP	DATE	Current date and time to the nearest millisecond (default).
CURRENT_DATE	DATE	Not supported	Current date on the server.
CURRENT_TIME	TIME	Not supported	Current time, expressed as hours, minutes and seconds since midnight, to the nearest millisecond (default).

Specifying Sub-seconds Precision

From v.2.0 forward, the sub-second part of CURRENT_TIMESTAMP and CURRENT_TIME is returned by default as milliseconds. It may be overridden to hundredths of seconds, tenths of seconds or to the v.1.X behaviour by the inclusion of an optional argument of the form

CONTEXT_VAR(n)

where n signals the sub-seconds precision as follows:

(3) for milliseconds. It is the default and need not be specified.	(2) for hundredths of second	(1) for tenths of seconds	(0) for legacy behaviour, returning subseconds as '.0000'
--	------------------------------	---------------------------	---

Operations Using Date and Time Values

The use of an arithmetic operation to manipulate, calculate, set or condition the relationships between two dates has already been visited in the topic earlier in this chapter, Interval of time (time elapsed). The capability to subtract an earlier date, time or timestamp value from a later one is possible because of the way Firebird stores date/time types. It uses one or two integers to store timestamps, date-only dates or time of day, respectively. The units represented by these numbers are days in the date-part integer and fractions of days in the time-part integer. The date part represents the number of days since “date zero”—November 17, 1898. The time part represents the portion of one day elapsed since midnight.

Dialect 3 DATE stores only the date part. Dialect 3 TIME stores only the fractional part. TIMESTAMP and dialect 1 DATE store both parts.

Quite simply, these number structures can be operated on, using simple addition and subtraction expressions, to calculate time differences (intervals), increment or decrement dates and set date and time ranges. Table 8.6 explains which operations are valid and the results to be achieved.

Table 8.6 Arithmetic involving date/time data types

Operand 1	Operator	Operand 2	Result
DATE	+	TIME	TIMESTAMP (arithmetic concatenation)
DATE	+	Numeric value n^{\ddagger}	Advances DATE value by n whole days (ignoring any fractional part supplied for n)
TIME	+	DATE	TIMESTAMP (arithmetic concatenation)
TIME	+	Numeric value n^{\ddagger}	Advances TIME value by n seconds †
TIMESTAMP	+	Numeric value n^{\ddagger}	Advances the time part of TIMESTAMP value by fraction part of n (if present) in seconds and date part by the whole-number part of n in days
DATE	–	DATE	DECIMAL(9,0) days of interval
DATE	–	Numeric value n^{\ddagger}	Reduces DATE value by n whole days (ignoring any fractional part supplied for n)
TIME	–	TIME	DECIMAL(9,4) seconds of interval
TIME	–	Numeric value n^{\ddagger}	Reduces TIME value by n seconds †
TIMESTAMP	–	TIMESTAMP	DECIMAL(18,9) days and part-day of interval
TIMESTAMP	–	Numeric value n^{\ddagger}	Reduces the time part of TIMESTAMP value by fraction part of n (if present) in seconds and date part by the whole-number part of n in days

† If necessary, repeats (result=modulo(result, (24 * 60 * 60))) until resultant day-part is eliminated.

‡ For dialect 3 DATE type, n is an integer, representing days. For TIMESTAMP and dialect 1 DATE types, n can be a numeric representing days to the left of the decimal point and a part-day to the right. For TIME type, n is an integer representing seconds.

General rules for operations

One date/time value can be subtracted from another, provided

- both values are of the same date/time type
- the first operand is later than the second

A valid subtraction involving date/time types produces a scaled DECIMAL result in dialect 3 and a DOUBLE PRECISION result in dialect 1.

Date/time types cannot be added together. However, a date-part can be concatenated to a time-part using

- dyadic¹ addition syntax to concatenate a pair of fields or variables
- string concatenation to concatenate a date/time literal with another date/time literal or with a date/time field or variable

Multiplication and division operations involving date/time types are not valid.



¹Dyadic: in or by groups of two

Expressions as operands

The operand to advance or reduce a TIMESTAMP, TIME or DATE value can be a constant or an expression. An expression may be especially useful in your applications when you want to advance or reduce the value specifically in seconds, minutes or hours or, for example, half-days, rather than directly by days.



When using expressions in dialect 3, ensure that one of the operands is a real number to avoid the possibility of a division-by-zero error resulting from SQL-92 integer division.

The following table 8.7 provides some examples of operand expressions that could be used to resolve input values to days or part-days for date/time additions to and subtractions from date/time values in calculations.

Table 8.7 Examples of operands using expressions

Input n	Add or subtract	Alternative
in seconds	n/86400.0	n * 1.0/(60 * 60 * 24)
in minutes	n/1440.0	n * 1.0/(60 * 24)
in hours	n/24.0	Depends on the result you want. For example, if n=3 and the divisor for half-days is 2, the result will be 1, not 1.5, according to the rules for SQL integer/integer division.
in half days	n/2	



Because years, months and quarters are not constant, more complex algorithms are needed for handling them in date/time operations. It may be worth your while to seek out user-defined functions (UDFs) that you can use in operand expressions to suit these requirements.

Expressions can involve functions, as we have already noted when we touched on CAST(.). The next topics look at the use of some date/time compatible functions.

Using CAST() with Date/Time Types

We have encountered the CAST(.) function in expressions involving date literals and date parts. This topic explores the various aspects of date and time casting in more depth and breadth.

Casting between date/time types

Generally, casting from one date/time type to another is possible wherever the source date/time type provides the right kind of data to reconstruct as the destination date/time type. For example, a `TIMESTAMP` can provide the date part to cast on to a date-only `DATE` type or a time-only `TIME` type, whereas a `TIME` type cannot provides enough data to cast on to a `DATE` type. Firebird allows a `DATE` type to be cast to `TIMESTAMP` by casting on a time part of midnight; and it allows a `TIME` type to be cast to `TIMESTAMP` by concatenating it on to the context variable `CURRENT_DATE` (the server date).

Table 8.8 Dialect 3 casting between date/time types

Cast source type	AS TIMESTAMP	AS DATE	AS TIME
TIMESTAMP	n/a	YES: casts on date part, ignores time part	YES: casts on time part, ignores date part
DATE	YES: time part is set to midnight	n/a	NO
TIME	YES: date part is set to <code>CURRENT_DATE</code>	NO	n/a
DATE + TIME	YES: CAST((DATEFIELD + TIMEFIELD AS TIMESTAMP)	NO	NO

Casting from date types to CHAR(n) and VARCHAR(n)

Use the SQL CAST() function in statements to translate between date and time data types and character-based data types.

Firebird casts date/time types to formatted strings where the date (if present) is in a set format—dependent on dialect—and the time part (if present) is in the standard Firebird `HH:NN:SS.nnnn` time format. It is necessary to prepare a `CHAR` or `VARCHAR` column or variable of a suitable size to accommodate the output you want.

Both fixed length `CHAR` and variable-length `VARCHAR` types can be cast to and from date/time types. Because the size of a cast date/time string is known and predictable, `CHAR` has a slight advantage over `VARCHAR` where date and time castings are concerned: using char will save you transmitting over the wire the two bytes that are added to `varchar`s to store their length.



The “right size” depends on dialect, so care is needed here. VARCHAR may be more suitable to use in application code that may need to handle both dialects.

If the character field is too small for the output, an overflow exception will occur. Suppose you want a cast string to contain only the date part of a `TIMESTAMP`. Preparing a character container of smaller size will not work: `CAST(.)` does not truncate the output string to fit. It is necessary to perform a double cast, first casting the timestamp as `DATE` and then casting that date to the correctly-sized character type—refer to examples below.

Dialect 3

Casting DATE or TIMESTAMP outputs the date part in ISO format (CCYY-MM-DD). To get the full length of the output, allow 10 characters for DATE and 11 for TIMESTAMP (date part plus one for the blank preceding the time part). Allow 13 characters for TIME or the time part of TIMESTAMP.

For example:

```
SELECT CAST(timestamp_col as CHAR(24)) AS TstampTxt
FROM ..
```

produces a string like this:

```
2012-06-25 12:15:45.2345
```

This produces an overflow exception:

```
SELECT CAST(timestamp_col as CHAR(20)) AS TstampTxt
FROM ..
```

A double-cast will produce the right string:

```
SELECT FIRST 1 CAST (CAST (timestamp_col AS DATE) AS CHAR(10))
FROM ..
2012-06-25
```

Unfortunately, it is not possible by direct casting to return a cast date + time string without the sub-second portion. It can be done using a complex expression involving both CAST(..) and EXTRACT(..). For an example, refer below to the topic The EXTRACT() Function, below.

Dialect 1

The date part of a Dialect 1 DATE type is converted to the format DD-MMM-CCYY, not the ISO format as in Dialect 3. So, for example,

```
SELECT CAST(d1date_col as CHAR(25)) AS DateTimeTxt
...
```

produces

```
26-JUN-2012 12:15:45.2345
```

Consequently, casting dialect 1 dates requires 11 characters instead of 10 for the date part, plus one for the blank space, plus 13 for the time part, 25 in all.

More complex expressions

CAST(..) can be used in more complex expressions, in combination with other expression operators. For example:

```
select cast (10 + cast(('today') as date) as char(25)) texttime
from rdb$database;
```

or

```
select cast (10 + current_timestamp) as date) as char(25)) texttime
from rdb$database;
```

produces a text string showing a date 10 days advanced from today's date.

Casting between date/time types and other types

Any character type or expression whose content can be evaluated to a valid date literal can be cast to the appropriate date/time type.

Date and time types cannot be cast to or from SMALLINT, INTEGER, FLOAT, DOUBLE PRECISION, NUMERIC, DECIMAL or BLOB types.

Uses for casting

- Exchanging date/time data with other applications
 - Importing date/time data created elsewhere—by another database system, host language or data-capture device, for example—often involves some degree of “massaging” before it can become valid date/time data for storing in a Firebird database.
 - Most host languages do not support the DATE, TIME, and TIMESTAMP types, representing them internally as strings or structures.
 - Data capture devices usually store dates and times in a variety of string formats and styles.
 - Date/time types are often incompatible between different database hosts.

Conversion generally requires evaluating and decoding the date-element content of the source data. The second part of the process is to reconstruct the decoded elements and pass them in Firebird SQL by some means. For a host language that has no means to pass native Firebird date/time types, the use of CAST(..) in combination with valid text strings for Firebird to process as date literals can be invaluable.

In some cases, external data stored into text files in date literal formats may be the best solution. Firebird can open such files as input tables in a server-side code module—stored procedure or trigger—and use CAST(..) and other functions to process data into date/time columns in native tables. Refer to [*Using External Files as Tables*](#) in Chapter 15 for more information.

CAST(..) can equally be used to prepare internal data for export.

- In search condition expressions

Situations arise where using CAST(..) in the WHERE clause with a date/time type will solve logical problems inherent in comparing a column of one type with a column of a different type.

Suppose, for example, we want to join a customer account table, which contains a DATE column BALANCE_DATE, with a customer transaction table which has a TIMESTAMP column TRANS_DATE. We want to make a WHERE clause that returns a set containing all of the unbilled transactions for this customer that occurred on or before the BALANCE_DATE. We might try:

```
SELECT...
  WHERE CUST_TRANS.TRANSDATE <= CUSTOMER.BALANCE_DATE;
```

This criterion does not give us what we want! It will find all of the transaction rows up to midnight of the BALANCE_DATE, because it evaluates BALANCE_DATE with a time part of 00:00:00. Any transactions after midnight on that date will fail the search criterion.

What we really want is to include all of the transactions where the date part of TRANS_DATE matches BALANCE_DATE. Casting TRANS_DATE to a DATE type saves the day:

```
SELECT...
  WHERE CAST(CUST_TRANS.TRANSDATE AS DATE) <= CUSTOMER.BALANCE_DATE;
```

- In a dialect conversion

Dialect 3 provides richer date/time support than dialect 1. One task that is likely to catch your attention if you do such a conversion is to replace or enhance existing dialect 1 DATE type columns (which are equivalent to TIMESTAMP in dialect 3) by converting them to the dialect 3 DATE (date-only) or TIME types. CAST(..) makes this job a no-brainer.

For an example of one style of conversion using cast, refer to [A sample date/time type conversion task](#) at the end of this chapter.

Quick Date/Time Casts

An SQL-compliant short expression syntax—sometimes referred to as “hints”—is available for casting both the predefined date/time literals and the regular constant date/time literals. It takes the form

```
<data type> <date literal>
```

Taking the CAST example above, the short syntax would be as follows:

```
select TIMESTAMP 'NOW'
FROM RDB$DATABASE
```

With a regular date/time literal:

```
SELECT TIME '15:05:45.345' FROM RDB$DATABASE
```

This short syntax can participate in other expressions. The following example illustrates a date/time arithmetic operation on a predefined literal:

```
update mytable
set OVERDUE = 'T'
where DATE 'YESTERDAY' - DATE_DUE > 10
```

The EXTRACT() Function

EXTRACT (..) decodes fields of date/time types and returns a variety of elements. It can operate on all dialect 3 and dialect 1 date/time fields.

Syntax

```
EXTRACT (element FROM field)
```

- **element** must be a defined element that is valid for the data type of field. Not all elements are valid for all date/time types. The data type of element varies according to the element extracted. Table 10-10 below enumerates the elements available for each date/time type.
- **field** can be a column, a variable or an expression that evaluates to a date/time field.

Table 8.9 shows thetypes, the arguments and the rules on their usage with EXTRACT (..).

Table 8.9 EXTRACT(..) arguments, types and rules

Element	Data type	Limits	TIMESTAMP ¹	DATE	TIME
YEAR	SMALLINT	0-5400	valid	valid	not valid
YEARDAY	SMALLINT	0-365	valid	valid	not valid
MONTH	SMALLINT	1-12	valid	valid	not valid
WEEK ^{2, 3}	SMALLINT	1-53	valid	valid	not valid

Element	Data type	Limits	TIMESTAMP ¹	DATE	TIME
DAY	SMALLINT	1-31	valid	valid	not valid
WEEKDAY	SMALLINT	0-6 ⁴	valid	valid	not valid
HOURL	SMALLINT	1-23	valid	not valid	valid
MINUTE	SMALLINT	1-59	valid	not valid	valid
SECOND	DECIMAL(6,4)	0-59.9999	valid	not valid	valid

¹ Also dialect 1 DATE type

² Under the ISO 8601 standards, dates may not overlap weeks in adjoining years and no date can fall into a gap between weeks. Hence, the ISO year starts at the first Monday of Week 1 and ends at the Sunday before the new ISO year.

- If 1 January is on a Monday, Tuesday, Wednesday or Thursday, it is in week 01.
- If 1 January is on a Friday, Saturday or Sunday, it is in week 52 or 53 of the previous year. You can identify the years that have a “Week 53” by counting the number of Thursdays in the calendar year.

³ The WEEK element is not supported in any 1.X versions.

⁴(0 = Sunday...6 = Saturday)

Combining EXTRACT(..) with other functions

Following are two examples where EXTRACT(..) is used with CAST(..) to obtain date representations not available with either function.

To cast date + time without sub-seconds

Although it is not possible by direct casting to return a cast date + time string without the sub-second portion, it can be done using a complex expression involving both CAST(..) and EXTRACT(..).

To extract a TIME string

This technique has more meaning for dialect 1 than for dialect 3. However, it can be extrapolated to any dialect 3 date or time type if you need to store time of day as a string, such as when writing to an external file for sharing with another application.

The EXTRACT(..) function makes it possible to extract the individual elements of date and time types to SMALLINT values. The following trigger extracts the time elements from a dialect 1 DATE column named CAPTURE_DATE and converts them into a CHAR(13) mimicking the Firebird standard time literal 'HH:NN:SS.nnnn':

```
SET TERM ^;
CREATE TRIGGER BI_ATALE FOR ATABLE
ACTIVE BEFORE INSERT POSITION 1
AS
BEGIN
  IF (NEW.CAPTURE_DATE IS NOT NULL) THEN
  BEGIN
    NEW.CAPTURE_TIME =
      CAST(EXTRACT (HOUR FROM NEW.CAPTURE_DATE) AS CHAR(2))|| ':' ||
      CAST(EXTRACT (MINUTE FROM NEW.CAPTURE_DATE) AS CHAR(2))|| ':' ||
```

```

        CAST(EXTRACT (SECOND FROM NEW.CAPTURE_DATE) AS CHAR(2))|| '.0000';
    END
END ^
SET TERM ;^

```

Other Date/Time Functions

Until v.2.1, only a small set of in-built functions was available. That is not as tough as it might appear to be: Firebird travels with two large libraries of external functions (“UDFs”) and a great many more are available in third-party libraries.

From v.2.1, the majority of the functions in the Firebird external libraries were integrated into Firebird’s data manipulation language (DML), many with improvements and SQL standards conformance. The internal date/time functions and their descriptions are grouped together for convenience, under *Date and Time Functions* in Appendix I. In the same Appendix, on [page 881](#), you will find also the external date/time functions.

A sample date/time type conversion task

The CHAR(13) string stored by the trigger in the example above (*To extract a TIME string*) does not behave like a dialect 3 TIME type. However, by simple casting, it can be converted in a subsequent upgrade to dialect 3, to a dialect 3 TIME type.

First, we add a temporary new column to the table to store the converted time string:

```

ALTER TABLE ATABLE
    ADD TIME_CAPTURE TIME;
COMMIT;

```

Next, populate the temporary column by casting the dialect 1 time string:

```

UPDATE ATABLE
    SET TIME_CAPTURE = CAST(CAPTURE_TIME AS TIME)
    WHERE CAPTURE_TIME IS NOT NULL;
COMMIT;

```

The next thing we need to do is temporarily alter our trigger to remove the reference to the dialect 1 time string. This is needed to prevent dependency problems when we want to change and alter the old time string:

```

SET TERM ^;
RECREATE TRIGGER BI_ATALE FOR ATABLE
ACTIVE BEFORE INSERT POSITION 1
AS
BEGIN
    /* do nothing */
END ^
SET TERM ;^
COMMIT;

```

Now, we can drop the old CAPTURE_TIME column:

```

ALTER TABLE ATABLE DROP CAPTURE_TIME;
COMMIT;

```

Now, we just rename the temporary column to `CAPTURE_TIME`:

Create it again, this time as a `TIME` type:

```
ALTER TABLE ATABLE
  ALTER COLUMN TIME_CAPTURE CAPTURE_TIME;
COMMIT;
```

Finally, fix up the trigger so that it now writes the `CAPTURE_TIME` value as a `TIME` type:

```
SET TERM ^;
RECREATE TRIGGER BI_ATALE FOR ATABLE
ACTIVE BEFORE INSERT POSITION 1
AS
BEGIN
  IF (NEW.CAPTURE_DATE IS NOT NULL) THEN
    BEGIN
      NEW.CAPTURE_TIME = CAST(NEW.CAPTURE_DATE AS TIME);
    END
  END ^
SET TERM ;^
COMMIT;
```

All of these steps can be written as an SQL script. For details about SQL scripting, refer to the topic [*Creating and Running Scripts*](#) in Chapter 24, ***Interactive SQL Utility (isql)***.

CHARACTER TYPES

Firebird supports both fixed length and variable-length character (string) data types. They can be defined for local usage in any one of a large collection of character sets. Fixed-length character types cannot exceed 32,767 bytes in absolute length; for variable-length types, the limit is reduced by two bytes to accommodate storing a character-count with each string stored.

String Essentials

Firebird stores strings very economically, using a simple run-length encoding algorithm to compress the data, even if it is a fixed-length CHAR or NCHAR type. In case that tempts you to define very large string columns, be aware that there are compelling reasons to avoid very long strings—client memory limitations and index size limits, to name two.

V.1.0 If you are still using Firebird 1.0.x, it is also important to know that strings, whether fixed or variable length types, are decompressed and padded to full declared length before they leave the server.

The CHARACTER SET attribute of character types is important not only for compatibility with localized application interfaces but also, in some cases, for deciding the size of the column. Certain character sets use multiple bytes—typically two, three or four bytes—to store a single character. When such character sets are used, the maximum size is reduced by a factor of the byte size. Over-length input for Firebird character columns incurs an overflow exception.

Note *The CHARACTER SET attribute is optional in a declaration. If no character set is defined at column level, the character set attribute is taken to be the default character set of the database. The mechanism for determining the character set of columns and variables is discussed in more detail later in this chapter.*

String Delimiter

Firebird's string delimiter is ASCII 39/U+0027, the single-quote or apostrophe character, for example:

```
StringVar = 'This is a string.';
```

Double-quotes are not permitted at all for delimiting strings. You should remember this if you are connecting to Firebird using application code written for ancient InterBase 5 databases, which permitted double-quoted strings. Strings should also be corrected in the source code of stored procedures and triggers in these old databases if you plan to recompile them in Firebird.

Concatenation

Firebird uses the SQL standard symbol for concatenating strings: a doublet of the single-pipe character, ASCII code 124/U+01C0, known as double-pipe: “||”.



Do not confuse this pair of pipe characters with the Unicode “double-pipe”, U+01C1. They are not interchangeable.

The pipe-pair can be used to concatenate string constants, string expressions and/or column values. For example:

```
MyBiggerString = 'You are my sunshine,' || FirstName || ' my only sunshine.';
```

Character elements can be concatenated to numbers and number expressions to produce an alphanumeric string result. For example, to concatenate the character '#' to an integer:

```
NEW.TICKET_NUMBER = '#' || NEW.PK_INTEGER;
```

Concatenation and NULL

Avoid using concatenation expressions where one or more of the elements might be null. If any part of a concatenation contains a NULL, the result of the concatenation will be NULL. That is not usually the desired result.

An “empty string”, represented as two apostrophe (single-quote) characters, is not the same as NULL.

Escape Characters

As a rule, Firebird does not support escape characters as a means to embed non-printable codes or sequences in character fields. The single exception is the “doubling” of the apostrophe character (ASCII 39/U+0027) to enable the apostrophe to be included as a stored character and prevent its being interpreted as the end-delimiter of the string:

```
...
SET HOSTELRY = 'O''Flaherty''s Pub'
...
```

Non-printable characters

It is possible to store non-printable characters in strings. Use the function `AsciiChar(asciivalue)` to pass such characters in strings. Declare and use the corresponding

external function of the same name in the `ib_udf` if you are using a Firebird version older than v.2.1.

The following statement outputs a set of text fields—to an external file, for example—with a carriage return and line feed in the last one:

```
INSERT INTO EXTFILE(DATA1, DATA1, DATA3, CRLF)
VALUES ('String1', 'String2', 'String3', AsciiChar(13)||AsciiChar(10));
```

String Functions

Functions can be applied to the character types in expressions to modify string input or output in many different ways. For v.2.1 and beyond, a wide range of useful functions is implemented internally. In the older versions, many of the same (or comparable) functions are available to the Firebird engine as external functions in the dynamic libraries `ib_udf` and `fbudf`, the default location of which is the sub-directory `../UDF` beneath the root of your installation.



Prior to v.2.1, for the declaration of `AsciiChar(..)` that is used in the example above, and other external functions in the `ib_udf` library, look in the `../UDF` subdirectory beneath the root of your Firebird installation for the script named `ib_udf.sql`. You can copy and paste the declaration into your own script or database administration tool.

Case

Strings can be “upper-cased” (converted to all upper case or “capitalised”) in any version of Firebird, using the internal function `UPPER()`. The internal function `LOWER()`, to convert strings to all lower case characters, is available from the “2” series onward. In older versions, use the external function `LOWER()` in the `../UDF/fbudf` library or an equivalent function from a third-party library that works correctly with your character sets.

String length

The length of a string can be measured in different ways by counting different aspects of its existence.

For the “2” series and onward, the ISO-compliant internal functions `CHAR_LENGTH` (or `CHARACTER_LENGTH`), `OCTET_LENGTH` and `BIT_LENGTH` each returns an integer value that counts, respectively, characters, bytes (octets) or bits.

For older versions, you are less well-served by the Firebird UDF libraries. The `ib_udf` library has `STRLEN()`, which is equivalent to `CHARACTER_LENGTH()`.

Substrings

The internal function `SUBSTRING()` is available in all Firebird versions except v.1.0.x. `TRIM()`, for trimming leading and/or trailing white space, is available from the “2” series onward. Other functions (`LEFT()`, `RIGHT()`) internal functions for v.2.1+, or various external functions in the `fbudf` library, such as `SRIGHT()` and `SUBSTRLEN()` can return substrings from specified start and/or end positions in strings.

NULL Inputs

Before Firebird 2.0, the string functions in the `ib_udf` library would not accept NULL as input. From the “2” series onward, the external functions `ASCII_CHAR`, `LOWER`, `"LOWER"`, `LPAD`, `LTRIM`, `RPAD`, `RTRIM`, `SUBSTR` and `SUBSTRLEN` can accept a NULL input, if those functions are declared correctly in the database.

For example, using `ASCII_CHAR()` again:

```
DECLARE EXTERNAL FUNCTION ascii_char
INTEGER NULL
RETURNS CSTRING(1) FREE_IT
ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf';
```

Finding a suitable function

Appendix I, *Internal and External Functions*, has full details, with examples, of the string functions available, both internally and through the UDFs that are distributed in the Firebird kits.

If you cannot find what you need, you might like to check out some of the better-known third party external function libraries. The IBPhoenix web site maintains [summaries and links](#) for several of these.

Limitations with Character Types

If you are planning to store data in large character columns, it is important to be aware of factors that might have impact on your design.

Multi-byte character sets

Multi-byte character sets reduce the potential sizes of text items, especially those with variable length. In UTF8, for example, even a 256-character column will be relatively large—potentially a Kilobyte or more—both to store and to retrieve. More is said later in this chapter regarding the caution you need to observe when considering text storage for multi-byte character data.

Index restrictions

When deciding on the length, character set and collation sequence for a character column, you need to be aware that indexing character columns is limited to 25% of the database page size. Multi-byte and many of the more complex 1-byte character sets use up many more bytes than the simpler character sets. Multi-segment indexes consume extra bytes, as do collation sequences. Do the byte calculations at design-time!

v.1.X

For server versions 1.0.x and 1.5.x and databases whose on-disk structure (ODS) is less than 11, the total width of any index can not exceed 253 bytes—note bytes, not characters.



At the time of this writing, a useful on-line calculator for the index sizes, that takes into account the character set, collation and number of index keys, is free to use at the website of my long-time friend and colleague, Ivan Prenosil. Try this URL: http://www.volny.cz/iprenosil/interbase/ip_ib_indexcalculator.htm If it fails, Google Ivan's name to find a new link.

For more details, see Chapter 16, *Indexes*, and study the topics later in this chapter about character sets and collation sequences.

Client Memory Consumption

Clients programs will allocate memory to store copies of rows that they read from the database. Many interface layers allocate sufficient resources to accommodate the maximum (i.e. defined) size of a fixed- or variable-length column value, even if none of the actual data stored is that large. Buffering large numbers of rows at a time may consume a

large amount of memory and users will complain about slow screen refreshes and lost connections!

Consider, for example, the impact on the workstation if a query returns 1024 rows consisting of just one column declared as VARCHAR(1024). Even with the “leanest” character set, this column would cost at least 1Mb of client memory. For a Unicode column, multiply that cost by three (UNICODE_FSS) or four (UTF8). Some Unicode collations may even consume five, six or even more bytes per character in index keys.

Fixed-length Character Data

Firebird's fixed-length string data types are provided for storing strings whose length is consistently the same or very similar or where the format or relative positions of characters might convey semantic content. Typical uses are for items such as identification codes, telecom numbers, character-based numbering systems and for defining fields to store preformatted fixed-length strings for conversion to other data types—Firebird date literals, for example—or for exporting to other data systems that require fixed-length text records.

Leading spaces characters (ASCII character 32) in fixed-length string input are significant, whereas trailing spaces are not. When storing fixed-length strings, Firebird strips trailing space characters. The strings are retrieved with right-padding out to the full declared length.

Using fixed-length types is not recommended for data which might contain significant trailing space characters, or items whose actual lengths are expected to vary widely.

Transport across the network

Fixed-length string types are padded to their full declared width for transport to the client.

CHAR(*n*), alias CHARACTER(*n*)

CHAR(*n*), alias CHARACTER(*n*) is the base fixed-length character type. *n* represents the exact number of characters stored. CHAR and CHARACTER are completely synonymous. It will store strings in any of the supported character sets.



*If the length argument, *n*, is omitted from the declaration, a CHAR(1) is assumed. It is acceptable to declare single-character CHAR fields simply as CHAR.*

NCHAR(*n*), alias NATIONAL CHARACTER(*n*)

NCHAR(*n*), alias NATIONAL CHAR(*n*) is a specialized implementation of CHAR(*n*) that is pre-defined with ISO8859_1 as its character set attribute. Of course, it is not valid to define a character set attribute for an NCHAR column, although a collation sequence—the sequence in which the sorting of characters is arranged for searches and ordered output—can be defined for a column or domain that uses this type.

Variable-length Character Data

Firebird's variable-length string data types are provided for storing strings which may vary in length. The mandatory size argument *n* limits the number of discrete characters that may be stored in the column to a maximum of *n* characters. The size of a varchar cannot exceed 32,765 bytes because Firebird adds a two-byte size element to each varchar item.

Important *Counting bytes is not necessarily the same as counting characters. For two-byte character sets the maximum length of a VARCHAR is 16,382 characters; for UTF8 it is 8,191 characters.*

Choosing, storing and retrieving variable-length text

A variable-length character type is the workhorse for storing text because the size of the stored structure is the size of the actual data plus two bytes. All characters submitted as input to a field of a variable-length type are treated as significant, including both leading and trailing space characters (“blanks”).

Transport across the network

Variable-length string types are not padded for transport to the client.

V.1.0.x Before Firebird 1.5, retrieved variable-length text data items were padded at the server to the full, declared size before being returned to the client, a “gotcha” that might influence your choice of column size and type if you are writing applications for remote clients connecting to a 1.0.x server on a slow network.

Although variable-length types can store strings of almost 32Kb, in practice it is not recommended to use them for defining data items longer than about 250 bytes, especially if their tables will become large or will be subject to frequent SELECT queries. BLOBs of SUB_TYPE 1 (text) are usually more suitable for storing large string data. Text BLOBs are discussed in detail in the next chapter.

VARCHAR(n), alias CHARACTER VARYING(n)

VARCHAR(*n*), alias CHARACTER VARYING(*n*), is the base variable-length character type. *n* represents the maximum number of characters that can be stored in the column. It will store strings in any of the supported character sets. If no character set is defined, the character set attribute defaults to the one defined during CREATE DATABASE as DEFAULT CHARACTER SET. If there is no default character set, the column default is CHARACTER SET NONE.

NCHAR VARYING(n), alias NATIONAL CHAR VARYING(n)

also aliased as NATIONAL CHARACTER VARYING(*n*)

NCHAR VARYING(*n*), alias NATIONAL CHAR VARYING(*n*), alias NATIONAL CHARACTER VARYING(*n*), is a specialized implementation of VARCHAR(*n*) that is pre-defined with ISO8859_1 as its character set attribute. It is not valid to define a character set attribute for an NVARCHAR column, although a collation sequence—the sequence in which the sorting of characters is arranged for searches and ordered output—can be defined for a column or domain that uses this type.

Character Sets and Collation Sequences

A character set is a collection of symbols that includes at least one *character repertoire*.. A character repertoire is a set of characters used by a particular culture for its publications, written communication and—in the context of a database—for computer input and output.

For example, ISO Latin_1 is a character set that encompasses the English repertoire (A, B, C ... Z) and the French repertoire (A, Á, À, B, C, Ç, D ... Z), making it useful for systems that span both cultural communities.

The character set chosen for storing text data determines:

- the characters that can be used in CHAR, VARCHAR, and BLOB SUB_TYPE 1 (text) columns.
- the number of bytes allocated to each character
- the default collation sequence (“alphabetical order”) to be used when sorting CHAR and VARCHAR columns.

Firebird allows character sets and collations to be defined for any character field or variable declaration.



BLOBs cannot be sorted—so collation sequence is not relevant to text BLOBs from that point of view. However, because they can be upper-cased and searched case-insensitively, that aspect of a collation is likely to be relevant wherever the character set contains characters of more than 7 bits.

Character Sets

Text input from keyboards and other input devices such as bar code readers—potentially for all characters—is encoded specifically according to a certain standard *code page* that may be linked to the locale that the input device is set up for.

In one code page, the numeric code that maps to a certain character image may be quite different to its mapping in another. Broadly, a character set reflects a certain code page or group of related code pages. Some character sets work with more than one code page; in some cases, a code page will work with more than one character set. Different languages may share one overall character set but map different upper-case/lower-case pairs, currency symbols, and so on.

The server needs to know what character set is to be stored, in order to compute the storage space and to assess the collation characteristics for proper ordering, comparison, upper-casing, and so on. Other than those requirements, it is unconcerned about the content of text input.

Default Character Set

The keyword triplet DEFAULT CHARACTER SET can be used as a parameter on the CREATE DATABASE statement. In Firebird 2.5+, you can use the optional COLLATION keyword to extend the DEFAULT CHARACTER SET parameter and specify a *default collation order* to be used with that character set.

If no default character set is defined for the database, the default is CHARACTER SET NONE.

Character Set Overrides

The default character set and collation are overridden by any CHARACTER SET or COLLATE clause, respectively, in column, domain or index definitions. Hence, having set the global default for the database, you can override it further down the food chain, if required. You can include a character set attribute when you create a domain. You can override either the database default or a domain setting in an individual column definition.



Overriding the database's default character set on a column or domain definition propagates the change only to rows added subsequently. Existing data retains the characteristics that were formerly stored. Great care should be taken to avoid having columns containing data that might be incompatible from one row to another.

Character set NONE

Character set NONE will accept any characters without complaining. It does not “know” about multi-byte characters and it does not recognise any single-byte characters other than the 7-bit unaccented “US ASCII” ones that are common to all character sets. Problems start when—in a non-US-English or mixed-language environment—you find that you are getting transliteration errors when you SELECT your text data. What comes out isn’t always what goes in!



If your database is for use in an English-only language environment, you can be tempted to ignore character sets and just let Firebird go ahead and use NONE for everything. Don't be tempted unless you are certain that no text data will ever contain data that depends on interpretation of characters of eight bits or more.

Client Character Set

What really matters with regard to character set is the interaction between server and client. The Firebird client library must be passed a character set attribute as part of the structure of a connection request.

The client application specifies its character set before it connects to the database. Connecting applications must pass the character set of the database to the API via the database parameter block (DPB) in the parameter *isc_dpb_lc_ctype*.

If the engine detects a difference between the client's character set setting and the character set of the target storage destination, it performs “transliteration”—automatically substituting the character codes according to the mapping it assumes from the information about the two character sets. It assumes that the incoming codes are correct for the client's code page.

That makes it possible to store text in different target columns having character sets that are not the same as the default character set of the database.

If the client and target character sets are the same, the engine assumes that the input it is receiving really is encoded for the defined character set and stores the input unmodified. Troubles ensue if the input was not what the client said it was. When the data are selected, searched or restored after a backup, they cause transliteration errors.

Database connection classes for Delphi, Java, et al., generally surface the *isc_dpb_lc_ctype* parameter as an attribute or property of the class that implements the API function call *isc_attach_database()*. An ESQL application—such as the *isql* utility—must execute a SET NAMES statement immediately before the CONNECT statement.

For example, the following ISQL command specifies that *isql* is using the ISO8859_1 character set. The next command connects to the authors.fdb database of our earlier example:

```
SET NAMES ISO8859_1;
CONNECT 'lserver:/data/authors.fdb' USER 'ALICE' PASSWORD 'XINEOHP';
```



GUI admin tools usually provide an input field of some sort for selecting the client character set. Host-language drivers and components often use the symbol lc_ctype as the name of a property or method for setting the client character set.

If you need to cater for a non-English language, spend some time studying the character sets available and choose the one which is most appropriate for most of your text input storage and output requirements. Remember to include that character set in the database attributes when you create the database.

- Refer to the relevant topic under *Database Attributes* in Chapter 14 for the syntax.

- For a list of character sets recognized by Firebird, refer to Appendix VI, [*Character Sets and Collations*](#).

Firebird character sets

Firebird supports an increasingly broad variety of international character sets, including a number of two-byte sets and the (potentially) four-byte UTF8 subset of Unicode.

v.1.X In the old versions 1.5.x and 1.0.x the only Unicode support is in the form of the three-byte UNICODE_FSS character set, an outdated version of UTF8 that accepts malformed strings and does not enforce maximum string length correctly. A character set named “UTF8” is available in v.1.5 but it is merely an alias to UNICODE_FSS.

For many character sets, choices of collation sequence are available.

Naming of character sets

Most Firebird character sets are defined by standards and their names closely reflect the standards that define them. For example Microsoft defines Windows 1252 and Firebird implements it as WIN1252. Character set ISO8859_1 is “the set of characters defined by ISO Standard 8859-1, encoded by the values defined in ISO standard 8859-1, having each value represented by a single 8-bit byte.”

Aliases

Character-set alias names support differences in the naming standards between different platforms. For example, if you find yourself on an operating system that uses the identifier WIN_1252 for the WIN1252 character set, you can use an alias that is defined in the system table RDB\$TYPES (see [*Adding an Alias for a Character Set*](#), below).

Storage of character sets and aliases

A catalogue of character sets is “hard-wired” in a database at creation time, when the system table RDB\$CHARACTER_SETS is built and populated. For a listing by character set name, including the name of the default collation sequence in each case, execute this query:

```
SELECT
    RDB$CHARACTER_SET_NAME,
    RDB$DEFAULT_COLLATE_NAME,
    RDB$BYTES_PER_CHARACTER
FROM RDB$CHARACTER_SETS
ORDER BY 1 ;
```

Aliases are added as required to RDB\$TYPES, another system table which stores enumerated sets of various kinds used by the database engine. To see all of the aliases which are set up at database-creation time, run this query, filtering RDB\$TYPES to see just the enumerated set of character set names:

```
SELECT
    C.RDB$CHARACTER_SET_NAME,
    T.RDB$TYPE_NAME
FROM RDB$TYPES T
JOIN RDB$CHARACTER_SETS C
ON C.RDB$CHARACTER_SET_ID = T.RDB$TYPE
```

```
WHERE T.RDB$FIELD_NAME = 'RDB$CHARACTER_SET_NAME'
ORDER BY 1 ;
```

If an alias you want is not present, you can add one. Later in this chapter, the topic Adding character sets shows how to do it.



In order to use any character set other than NONE, ASCII, OCTETS and UNICODE_FSS, it is necessary to have the fbintl library present in the /intl directory beneath the root directory. (For Windows embedded server, the \intl directory should be a sub-folder of the folder where the fbembed.dll is located).

Well-formedness

Some character sets (especially multi-byte ones) do not accept “just any string”. In the “2” series and beyond, the engine verifies that strings are well-formed when assigning from NONE/OCTETS and when processing statement strings and text parameters sent by a client application.

Versions 1.X *In the 1.X versions there are no well-formedness checks.*

Storage restrictions

It is really important to understand how your choice of character set affects the storage boundaries for the data you plan to accommodate. In the case of CHAR and VARCHAR columns, the maximum amount of storage in any column to 32,767 bytes and 32,765 bytes, respectively. Yes, BYTES, not characters.

In versions prior to Firebird 2, the engine does not verify the logical length of multi-byte character set strings. Hence, for example, a UNICODE_FSS field takes three times as many characters as the declared field size, three being the maximum length of one UNICODE_FSS character. The three-byte rule applies, even if the actual character is only one or two bytes. That rule is retained for the pre-existing multi-byte character sets to maintain compatibility.

Character sets added since Firebird 2, UTF8 for example, do not inherit that limitation. Under the newer rules, the byte length of characters is counted on the actual data.

The actual number of characters that can be stored might be severely limited by a number of factors:

Non-indexed columns using the default collation sequence can store (number of characters) * (number of bytes per character) up to the byte limit for the data type. For example, a VARCHAR(32765) in character set ISO_8859_1 can store up to 32,765 characters, while in character set UTF8 (which uses up to four bytes per character) the upper limit may be as little as 8,191 characters.

If a column is to be indexed and/or modified with a COLLATE clause, a significant number of “spare” bytes must be allowed. Even the least demanding index—on a single VARCHAR column using a single-byte character set and the default collation sequence—is limited to 25 per cent of the database page size. In the old 1.X versions, the limit is 252 bytes.

For multi-byte character set columns, the character limit for an index is less than (252/(number of bytes per character)). Multi-column indexes consume more bytes than do single-column indexes; those using a non-default collation sequence consume still more. For more detail about these effects, refer to the topic Collation sequence and index size later in this chapter.

Tip *When designing columns, always consider likely needs with regard to character set, indexing and key usage. Keep a special “scratch-pad” table in your development database just for testing the limits of indexes and keys.*

The storage size for BLOB columns, which are non-indexable, is not formally restricted by the limits on string sizes.

Field-level character set overrides

A character set attribute can be added to the individual definition of a domain, table column or PSQL variable of type CHAR, VARCHAR or BLOB SUB_TYPE 1 to override the default character set of the database.

For example, the following script fragment creates a database with a default character set of ISO8859_1 and a table containing different language versions of similar data in separate columns:

```
CREATE DATABASE '/data/authors.fdb' DEFAULT CHARACTER SET ISO8859_1;
...
CREATE TABLE COUNTRY_INTL(
  CNTRYCODE INTEGER NOT NULL,
  NOM_FR VARCHAR(30) NOT NULL, /* uses default charset */
  NOM_EN VARCHAR(30), /* uses default charset */
  NOM_RU VARCHAR(30) CHARACTER SET CYRL,
  NOM_JP VARCHAR(30) CHARACTER SET SJIS_0208
);
```

Another fragment from the same script creates a domain for storing BLOB data in the Cyrillic character set:

```
CREATE DOMAIN MEMO_RU AS BLOB SUB_TYPE 1 CHARACTER SET CYRL;
```

Later in the script, we define a table that stores some text in Cyrillic:

```
CREATE TABLE NOTES_RU (
  DOC_ID BIGINT NOT NULL,
  NOTES MEMO_RU
);
```

Creating domains is explained in Chapter 11, *Domains*. Complete syntax for CREATE TABLE is in Chapter 15, *Tables*.

Statement-level character set overrides

The character set for text values in a statement is interpreted according to the connection’s character set at run-time (not according to the character set defined for the column when it was created) unless you specify a character set marker (or “introducer”) to indicate a different character set.

Character set marker—INTRODUCER

A character set marker—also known as an INTRODUCER—consists of the character set name prefixed by an underscore character. It is required to “introduce” an input string when the client application is connected to the database using a character set that is different to that of the destination column in the database.

Position the marker directly to the left of the text value being marked. For example, the introducer for input to a UNICODE_FSS field is `_UNICODE_FSS`:

```
INSERT INTO CONTRACTOR(Contractor_ID, Contractor_Name)
```

```
values(1234, _UNICODE_FSS'Smith, John Joseph');
```

For clarity, you may leave white space between the introducer and the string without affecting the way the input is parsed.



Use the name of the character set for the introducer, not an alias. Thus, for example, use `_ISO8859_1`, not any of `(_ANSI, _ISO88591, _LATIN1)`, which will just return character set errors.

String literals

A string literal in a test or search condition, for example, in a WHERE clause, is interpreted according to the character set of the client's connection at the time the condition is tested. An introducer will be required if the database column being searched has a character set which is different to that of the client connection:

```
... WHERE name = _ISO8859_1 'joe';
```



When you are developing with mixed character sets, it is good practice to use introducers as a matter of form, especially if your application will cross the boundaries of multiple databases and/or it will be deployed internationally.

sqlsubtype of an XSQLVAR

An XSQLVAR is a structure that applications pass across the interface when describing columns or parameters supplied in statements. Usually, the *sqlsubtype* member of an XSQLVAR pertaining to a column contains the identifying number of the column's character set.

In Firebird 2 and higher, when the character set of a CHAR or VARCHAR column is anything but NONE or OCTETS and the attachment character set is other than NONE, the *sqlsubtype* contains the identifier of the client character set instead.



The scope of this book does not reach the API. It is assumed that you will be using an existing API wrapper or driver for your application development. The change is mentioned here because it could present a compatibility issue with legacy applications that have been built using an API wrapper for an older Firebird, or for InterBase.

Special Character Sets

The general rule for character sets is that every byte (or pair or trio, in the case of multi-byte sets) is specifically defined by the standard it implements. There are four special exceptions—NONE, OCTETS, ASCII and UNICODE_FSS. The following table explains the special qualities of these three sets.

Table 9.1 Special character sets

Name	Qualities
NONE	Each byte is part of a string but no assumption is made about which character set it belongs to. Client-side or user-defined server code is responsible for character fidelity.
OCTETS BINARY	Bytes which will not be interpreted as characters. Useful for storing binary data, GUID strings, hexadecimal numbers and for correcting string data that is causing transliteration errors. Aliased as BINARY from the “2” series onward.

Name	Qualities
ASCII	Values 0..127 are defined by ASCII; values outside that range are not characters, but are preserved. Firebird is fairly liberal about transliterating bytes in the 0..127 range of ASCII characters.
UNICODE_FSS	A limited UTF8 implementation that was superseded by a more proper UTF8 implementation from the “2” series onward. In v.1.X it is aliased as UTF8. Firebird still uses this character set to store metadata text. No collation sequence other than the default binary one is available

Two special character sets, NONE and OCTETS, can be used in declarations. However, OCTETS cannot be used as a connection character set. The two sets are similar, except that the space character of NONE is ASCII 0x20, whereas the space character of OCTETS is 0x00.

Conversion Rules

NONE and OCTETS follow different conversion rules to those that apply to other character sets. With other character sets, conversion is performed as

`CHARACTER_SET1->UNICODE->CHARACTER_SET2,`

With NONE and OCTETS, the bytes are just copied verbatim:

`NONE/OCTETS->CHARACTER_SET2` and `CHARACTER_SET1->NONE/OCTETS`

Binary Strings

You can use either NONE or OCTETS to store binary strings, such as GUIDs, hexadecimal numbers or bit arrays. Both are locale-neutral. OCTETS is recommended over NONE because of its special treatment of trailing blanks in equality comparisons: they are stored as the NUL character (hex 0), instead of the SPACE character (hex 20), and will not be ignored.



In isql, from the “2” series onward, columns in OCTETS are displayed in hexadecimal format.

Hexadecimal input

From v.2.5 onward, you have the option to enter “binary string” literals in hexadecimal notation, i.e., as pairs of hex digits (0..9, a..f, A..F), each pair representing one byte. The input syntax has the pattern:

`{X|x}'<hex-pairs>'`

The letter “X” signals that the ensuing strings are to be interpreted as hex pairs. It can be lower or upper case. The string of hex pairs itself must not contain spaces and will cause an exception if the digit count is an odd number.

For example, to input a hex string and return a string in character set NONE:

`select x'44756c63696d6572' from rdb$database`

—returns the “binary string” ‘44756c63696d6572’

If we want the hex string returned as a character string, we can apply the “introducer” syntax to tell the engine to interpret the hex string according to a specific character set, e.g.,

`select _UTF8 x'44756c63696d6572' from rdb$database`

—returns ‘Dulcimer’

```
The next hex input contains a Norwegian character:
select x'76696e646dc3a5bc6572' from rdb$database
—returns the “binary string” ‘76696e646dc3a5bc6572’
select _utf8 x'76696e646dc3a5bc6572' from rdb$database
—returns ‘vindmåler’
```

Notice how the hex string contains an “extra” byte: it has 10 hex pairs for nine characters. All of the characters in ‘vindmåler’ are single-byte except the sixth (å), which has two. This feature will have its uses where you might want to store data from multiple character sets in the same column. You could define the column with character set OCTETS and input the data as hex strings.

ISO8859_1 (LATIN_1) and WIN1252

The Firebird ISO8859_1 character set is often specified to support European languages. ISO8859_1, also known as LATIN_1, is a proper subset of WIN1252: Microsoft added characters in positions that ISO specifically defines as Not a character (not “undefined”, but specifically “not a character”). Firebird supports both WIN1252 and ISO8859_1. You can always transliterate ISO8859_1 to WIN1252, but transliterating WIN1252 to ISO8859_1 can result in errors.

Character sets for Microsoft Windows

Five character sets support Windows client applications, such as Paradox for Windows. These character sets are WIN1250, WIN1251, WIN1252, WIN1253, and WIN1254. The names of collation sequences for these character sets that are specific to Paradox for Windows begin “PXW” and correspond to the Paradox/DBase language drivers supplied in the now-obsolete BDE (Borland Database Engine).

A “Gotcha” *The PXW collation sequences really implement collations from Paradox and DBase, including ALL BUGS. One exception, PXW_CSY was fixed in Firebird 1.0. Thus, legacy InterBase® databases that use it, for example in indexes, are not binary-compatible with Firebird.*

For a list of the international character sets and collation sequences that Firebird supports “out-of-the-box” at the time of this writing, see Appendix VI.

Transliteration

Converting characters from one Firebird character set to another—for example, converting from DOS437 to ISO8859_1—is transliteration. Firebird’s transliterations preserve character fidelity: By design, they will not substitute any kind of “placeholder” character for an input character that is not represented in the output character set. The purpose of this restriction is to guarantee that it will be possible to transliterate the same passage of text from one character set to another, in either direction, without losing any characters in the process.

Transliteration errors

Firebird reports a transliteration error if a character in the source set does not have an exact representation in the destination set (error code 335544565):
Cannot transliterate character between character sets
An example of where a transliteration error may occur is when an application passes input of some unspecified character set into a column defined with NONE and later tries to

select that data for input to another column that has been defined with a different character set. Even though you thought it should work because the character images looked as though they belonged to the character set of the destination column, the entire transliteration fails because a character is encountered which is not represented in the destination character set.

Fixing transliteration errors

How can you deal with a bunch of character data that you have stored using the wrong character set? One trick is to use character set OCTETS as a staging post between the wrong and the right encoding. Because OCTETS is a special character set that blindly stores only what you poke into it—without transliteration—it is ideal for making the character codes neutral with respect to code page.

For example, suppose your problem table has a column COL_ORIGINAL that you accidentally created in character set NONE, when you meant it to be CHARACTER SET ISO8859_2. You've been loading this column with Hungarian data but, every time you try to select from it, you get that wretched transliteration error!

Here's what you can do:

```
ALTER TABLE TABLEA
  ADD COL_ISO8859_2 VARCHAR(30) CHARACTER SET ISO8859_2;
COMMIT;
UPDATE TABLEA
  SET COL_ISO8859_2 = CAST(COL_ORIGINAL AS CHAR(30) CHARACTER SET OCTETS);
```

Now you have a temporary column designed to store Hungarian text—and it is storing all of your “lost” text from the unusable COL_ORIGINAL.

It would be wise to view the data in column COL_ISO8859_2 now! Of course, we assume that you are on a computer that can display Hungarian characters. If it looks good, you can proceed to drop COL_ORIGINAL:

```
ALTER TABLE TABLEA
  DROP COL_ORIGINAL;
COMMIT;
```

Now, all that is left to do is to rename the temporary column to the name of the dropped column:

```
ALTER TABLE TABLEA
  ALTER COLUMN COL_ISO8859_2 TO COL_ORIGINAL;
COMMIT;
```

Collation Sequence

Beyond character sets, different countries, languages, or even cultural groups using the same character mappings, use different sequences to determine “alphabetical order” for sorting and comparisons. Hence, for most character sets, Firebird provides a variety of collation sequences. Some collation sequences also take in upper/lower case pairings to resolve case-insensitive orderings. The COLLATE clause is used in several contexts where collation sequence is important.

Each character set has a default collation sequence that specifies how its symbols are sorted and ordered. Collation sequence determines the rules of precedence that Firebird uses to sort, compare, and transliterate character data.

Because each character set has its own subset of possible collation sequences, the character set that you choose when you define the column limits your choice. You must choose a collation sequence that is supported for the column's character set.

Collation sequence for a column is specified when the column is created or altered. When set at the column level, it overrides any collation sequence set at the domain level.

Default collation sequence

Each character set in the catalogue comes with a default collation with the same name as the character set. The sort order for this default collation is binary, the sort sequence reflecting the binary codes of the characters. In versions older than v. 2.5, it is not possible to specify a default collation sequence at database level.

As mentioned already, when creating a database of on-disk structure (ODS) 11. 2 or higher, you can use the optional COLLATION keyword to extend the DEFAULT CHARACTER SET parameter and specify a default collation sequence to be used with that character set.

Upper-casing in the default collations

The default collations of character sets in Firebird versions prior to the “2” series provided lower/upper case mappings only for the 7-bit characters—that group of characters traditionally referred to as “ASCII characters”. In the 1.X versions, the binary collations cannot handle mappings for accented characters at all.

Adriano dos Santos Fernandes, who implemented the international interface for the “2” series releases, provided an illustrative example of how the differences in upper-casing between old and new in the binary collations manifest themselves.

```
isql -q -ch dos850
SQL> create database 'test.fdb';
SQL> create table t (c char(1) character set dos850);
SQL> insert into t values ('a');
SQL> insert into t values ('e');
SQL> insert into t values ('á');
SQL> insert into t values ('é');

SQL> select c, upper(c) from t;
```

This is the result in versions 1.X:

C	UPPER
=====	=====
a	A
e	E
á	á
é	é

In Firebird 2.0 the result is:

C	UPPER
=====	=====
a	A
e	E
á	Á
é	É

Listing available collation sequences

The following query yields a list of character sets with the available collation sequences:

```
SELECT
  C.RDB$CHARACTER_SET_NAME,
  CO.RDB$COLLATION_NAME,
  CO.RDB$COLLATION_ID,
  CO.RDB$CHARACTER_SET_ID,
  CO.RDB$COLLATION_ID * 256 + CO.RDB$CHARACTER_SET_ID AS TEXTTYPEID
FROM RDB$COLLATIONS CO
JOIN RDB$CHARACTER_SETS C
  ON CO.RDB$CHARACTER_SET_ID = C.RDB$CHARACTER_SET_ID;
```

Naming of collation sequences

Many Firebird collation names use the naming convention **XX_YY**, where **XX** is a two-letter language code, and **YY** is a two-letter country code. For example, **DE_DE** is the collation name for German as used in Germany; **FR_FR** is for French as used in France; and **FR_CA** is for French as used in Canada.

Where a character set offers a choice of collations, the one with the name matching the character set is the default collation sequence, which implements binary collation for the character set. Binary collation sorts a character set by the numeric codes used to represent the characters. Some character sets support alternative collation sequences using different rules for determining precedence.

This topic explains how to specify collation sequence for character sets in domains and table columns, in string comparisons, and in **ORDER BY** and **GROUP BY** clauses.

Collation sequence for a column

When a **CHAR** or **VARCHAR** column is created for a table, with either **CREATE TABLE** or **ALTER TABLE**, the collation sequence for the column can be specified using the **COLLATE** clause. **COLLATE** is especially useful for multi-language character sets such as **ISO8859_1** or **DOS437** that support many different collation sequences.

For example, the following dynamic **ALTER TABLE** statement adds a new column to a table, and specifies both a character set and a collation sequence:

```
ALTER TABLE 'EMP_CANADIEN'
  ADD ADDRESS VARCHAR(40)
    CHARACTER SET ISO8859_1 NOT NULL COLLATE FR_CA;
```

Refer to Chapter 15 for complete syntax for *Altering Columns in a Table* (**ALTER TABLE**, et al).

Collation sequence for string comparisons

It can be necessary to specify a collation sequence when **CHAR** or **VARCHAR** values are compared in a **WHERE** clause, if the values being compared use different collation sequences and it matters to the result.

To specify the collation sequence to use for a value during a comparison, include a **COLLATE** clause after the value. For example, the following **WHERE** clause fragment forces the column value to the left of the equivalence operator to be compared with the input parameter using a specific collation sequence:

...

```
WHERE SURNAME COLLATE FR_CA >= :surname;
```

In this case, without matching collation sequences, the candidates for “greater than” might be different for each collation sequence.

Collation sequence in sorting criteria

When CHAR or VARCHAR columns are ordered in a SELECT statement, it can be necessary to specify a collation order for the ordering, especially if columns used for ordering use different collation sequences.

To specify the collation sequence to use for ordering a column in the ORDER BY clause, include a COLLATE clause after the column name. For example, in the following ORDER BY clause, the collation sequences for two columns are specified:

```
...
ORDER BY SURNAME COLLATE FR_CA, FIRST_NAME COLLATE FR_CA;
```

For the complete syntax of the ORDER BY clause, see Chapter 22, *Ordered and Aggregated Sets*.

Collation sequence in a GROUP BY clause

When CHAR or VARCHAR columns are grouped in a SELECT statement, it can be necessary to specify a collation order for the grouping, especially if columns used for grouping use different collation sequences.

To specify the collation sequence to use for grouping columns in the GROUP BY clause, include a COLLATE clause after the column name. For example, in the following GROUP BY clause, the collation sequences for multiple columns are specified:

```
...
GROUP BY ADDR_3 COLLATE CA_CA, SURNAME COLLATE FR_CA, FIRST_NAME COLLATE FR_CA;
```

For the complete syntax of the GROUP BY clause, see Chapter 22, *Ordered and Aggregated Sets*.

Collation sequence and index size

If you specify a non-binary collation (one other than the default collation) for a character set, the index key can become larger than the stored string if the collation includes precedence rules of second-, third-, or fourth-order.

Non-binary collations for ISO8859_1, for example, use full dictionary sorts, with spaces and punctuation of fourth-order importance:

- First order: A is different from B
- Second order: A is different from À
- Third order: A is different from a
- Fourth order: the type of punctuation symbol (hyphen, space, apostrophe) is important

Example
Greenfly
Green fly
Green-fly
Greensleeves
Green sleeves
Green spot

If spaces and punctuation marks are treated instead as a first-order difference, the same list would be sorted as follows:

```
Greenfly
Greensleeves
Green fly
Green sleeves
Green spot
Green-fly
```

How non-binary collations can limit index size

When an index is created, it uses the collation sequence defined for each text segment in the index. Using ISO8859_1, a single-byte character set, with the default collation, the index structure in a database with a page size of 4Kb can hold about 1020 (252 in Firebird 1.X) characters (fewer, if it is a multi-segment index). However, if you choose a non-binary collation for ISO8859_1, the index structure might accommodate only 340 characters (84 in v.1.X), despite the fact that the characters in the column being indexed occupy only one byte each.



Some ISO8859_1 collations, DE_DE for example, require an average of three bytes per character for an indexed column.

Adding an Alias for a Character Set

If you need to add a custom alias for a character set that is available on your server, all that is required is to add a member to the Firebird enumerated type identified as 'RDB\$CHARACTER_SET_NAME' in the table of symbolic constants, RDB\$TYPES.

The following stored procedure takes as arguments the name of the character set you want to alias, the alias name you want to use and an optional user description.

```
SET TERM ^;
CREATE PROCEDURE LOAD_CHAR_ALIAS (
  CHARSET CHAR(31) CHARACTER SET UNICODE_FSS,
  ALIAS_NAME CHAR(31) CHARACTER SET UNICODE_FSS,
  USER_DESCRIPTION VARCHAR(200) CHARACTER SET UNICODE_FSS)
AS
DECLARE VARIABLE TYPE_ID SMALLINT;
BEGIN
  TYPE_ID = 0;
  SELECT RDB$CHARACTER_SET_ID FROM RDB$CHARACTER_SETS
  WHERE RDB$CHARACTER_SET_NAME = UPPER(:CHARSET)
  INTO :TYPE_ID;
  IF (TYPE_ID > 0 AND TYPE_ID IS NOT NULL) THEN
    INSERT INTO RDB$TYPES (
      RDB$FIELD_NAME,
      RDB$TYPE,
      RDB$TYPE_NAME,
      RDB$DESCRIPTION,
      RDB$SYSTEM_FLAG)
```

```
VALUES (
  'RDB$CHARACTER_SET_NAME',
  :TYPE_ID,
  UPPER(:ALIAS_NAME),
  :USER_DESCRIPTION,
  0) ;

END ^
COMMIT ^
SET TERM ;^
```

Now, to load your new alias—let's say it is to be 'x-sjis' as an alias for the character set SJIS_0208—execute the procedure as follows:

```
EXECUTE PROCEDURE LOAD_CHAR_ALIAS
  ('SJIS_0208', 'x-sjis', 'This is a description');
COMMIT;
```



Although metadata are stored in UNICODE_FSS, the system identifiers are constrained to permit only the 7-bit alphanumeric characters.

Custom character sets and collations

From the “2” series forward, Firebird comes with plug-in support for the open source “International Components for Unicode” (ICU) libraries.

New character sets and collations are implemented through dynamic libraries and installed in the server with a manifest file, which acts like an index or catalogue of available character sets and their implementations. Not all implemented character sets and collations need to be listed in the manifest file but only those listed are available. If duplications are present in the manifest, only the first-occurring entry is loaded. An error is reported in `firebird.log` if a subsequent duplicate entry is detected but no exception occurs.



The manifest file should be put in the \$rootdir/intl with a “.conf” extension. For an example, see `fbintl.conf` in the /intl folder of your Firebird server installation.

Dynamic Libraries

For Windows, the dynamic libraries for the plug-in language support are located in the \bin directory of your Firebird installation, with names starting with “icu”, e.g., Firebird 2.5 ships with `icudt130.dll`, `icuin30.dll` and `icuuc30.dll`.

For Linux, the shared objects and symlinks are installed in the /lib subdirectory with names starting with “libc”.

You must deploy the ICU libraries with the “2” series and higher servers. This applies to the Windows Embedded Server model as well, in which the libraries are deployed in the emulated Firebird <root> directory.

Adding More Character Sets to a Database

For installing additional character sets and collations into a database, the character sets and collations should be registered in the database's system tables (`RDB$CHARACTER_SETS` and `RDB$COLLATIONS`). Don't try to update the metadata tables manually: the file `/misc/intl.sql`, in your Firebird 2.X installation, is a script of stored procedures for registering and unregistering them.

Example The following is an example of a section from `fbintl.conf`. The symbol `$(this)` is used to indicate the same directory as the manifest file. The library extension should be omitted.

```
<intl_module fbintl>
    filename $(this)/fbintl
</intl_module>
<charset ISO8859_1>
    intl_module fbintl
    collation ISO8859_1
    collation DA_DA
    collation DE_DE
    collation EN_UK
    collation EN_US
    collation ES_ES
    collation PT_BR
    collation PT_PT
</charset>
<charset WIN1250>
    intl_module fbintl
    collation WIN1250
    collation PXW_CSY
    collation PXW_HUN
    collation PXW_HUNDC
</charset>
```

Using ICU Character Sets

All non-wide and ASCII-based character sets present in ICU can be used by Firebird 2.1 and higher versions. To reduce the size of the distribution kit, Firebird's ICU distribution is customised to include only essential character sets and any for which there was a specific feature request.

If the character set you need is not included, you can replace the ICU libraries with another complete module, provided it is installed in your operating system.



A good place to start looking for the libraries is <http://site.icu-project.org/download>.

Registering an ICU Character Set Module

To use an alternative character set module, you need to register it in two places:

- 1 in the server's language configuration file, `intl/fbintl.conf`
- 2 in each database that is going to use it

You should prepare and register the module in the server before you register the character set[s] in the database.

Registering a Character Set on the Server

Using a text editor, register the module in `intl/fbintl.conf`, as follows.-

```
<charset          NAME>
    intl_module    fbintl
    collation      NAME [REAL-NAME]
```

</charset>

For example, to register a new character set and two of its collations, add the following to `fbintl.conf`:

```
<charset          GB>
  intl_module      fbintl
  collation        GB
  collation        GB18030
</charset>
```

Registering a Character Set in a Database

It takes two steps to make the character set and your required collation available in a database:

- 1 Run the procedure `sp_register_character_set`, the source for which can be found in `misc/intl.sql` beneath your Firebird 2.1 root, AND—
- 2 Use a `CREATE COLLATION` statement to register each collation

Step 1 Using the Stored Procedure

The stored procedure takes two arguments: a string that is the character set's identifier as declared in the configuration file and a smallint that is the maximum number of bytes a single character can occupy in the encoding. For the example (above) that we registered on the server:

```
execute procedure sp_register_character_set ('GB', 4);
```

Step 2 Registering the Collations

For the purpose of our example, the syntax for registering the two collations in a database is very simple:

```
CREATE COLLATION GB FOR GB;
CREATE COLLATION GB18030 FOR GB;
```

More complex directives can be used in `CREATE COLLATION`, as we discover next.

The CREATE COLLATION Statement

If you are using a server version 2.1 or higher and the ODS of your database is 11.1 or higher, the new `CREATE COLLATION` statement provides a way for you to register your collation to your database through dynamic SQL. The syntax pattern is:

```
CREATE COLLATION <name>
  FOR <charset>
  [ FROM <base> | FROM EXTERNAL ('<name>') ]
  [ NO PAD | PAD SPACE ]
  [ CASE SENSITIVE | CASE INSENSITIVE ]
  [ ACCENT SENSITIVE | ACCENT INSENSITIVE ]
  [ '<specific-attributes>' ]
```

Attributes are presented as a string that consists of a list of specific attributes separated by semi-colons. The attributes are case-sensitive.

The new collation should be declared in a `.conf` file in the `<root>/intl` directory before you prepare and execute the `CREATE COLLATION` statement.

Examples `CREATE COLLATION UNICODE_ENUS_CI`
 `FOR UTF8`

```

FROM UNICODE
CASE INSENSITIVE
'LOCALE=en_US';
--
CREATE COLLATION MY_COLLATION
FOR WIN1252
PAD SPACE;

```

More examples follow in the next section, illustrating use of specific attributes.

Conventions and Attributes

Attributes are stored at database creation time and only into databases having an on-disk structure of 11.1 or higher. Hence, these files can be utilised if you are creating a new database under Firebird 2.1+ or if you are restoring an older database in a server version that is newer than Firebird 2.0.

It is strongly recommended that you follow the naming convention as modelled in `fbintl.conf`, using the character set name followed by the collation name, with the two name parts joined by an underscore character, viz.:

`<character set>_<collation>`

Examples

```

CREATE COLLATION WIN1252_UNICODE
FOR WIN1252;
--
CREATE COLLATION WIN1252_UNICODE_CI
FOR WIN1252
FROM WIN1252_UNICODE
CASE INSENSITIVE;

```

Specific Attributes for Collations

Some attributes may not be applicable to some collations. If that is the case, no error is reported.

DISABLE-COMPRESSIONS

Prevents compressions (otherwise referred to as “contractions”) from changing the order of a group of characters. It is not valid for collations of multi-byte character sets.

Format: `DISABLE-COMPRESSIONS={0 | 1}`

Example `DISABLE-COMPRESSIONS=1`

DISABLE-EXPANSIONS

Prevents expansions from changing the order of a character to sort as a group of characters. It is not valid for collations of multi-byte character sets.

Format: `DISABLE-EXPANSIONS={0 | 1}`

Example `DISABLE-EXPANSIONS=1`

ICU-VERSION

Specifies the version of the ICU library to be used for UNICODE and UNICODE_CI.

Valid values are the ones defined in the configuration file (`intl/fbintl.conf`) in the entry `intl_module/icu_versions`.

Format: ICU-VERSION={default | major.minor}

Example ICU-VERSION=3.0

LOCALE

Specifies the collation locale for UNICODE and UNICODE_CI.



To use this attribute you need the complete version of the ICU libraries.

Format: LOCALE=xx_XX

Example LOCALE=en_US

MULTI-LEVEL

Registers that the collation uses more than one level for ordering purposes. It is not valid for collations of multi-byte character sets.

Format: MULTI-LEVEL={0 | 1}

Example MULTI-LEVEL=1

SPECIALS-FIRST

Specifies that special characters (spaces, symbols, etc) precede alphanumeric characters. It is not valid for collations of multi-byte character sets.

Format: SPECIALS-FIRST={0 | 1}

Example SPECIALS-FIRST=1

NUMERIC-SORT

For UNICODE collations only, available from v.2.5 onward, specifies the sorting order for numeral characters.

Format & usage

Format: NUMERIC-SORT={0 | 1}

Examples The default, 0, sorts numerals in alphabetical order, e.g.

1
10
100
2
20
...

1 sorts numerals in numerical order, e.g:

1
2
10
20
100

create collation sort_numeral for UTF8
from UNICODE 'NUMERIC-SORT=1';

Checking attributes

In *isql*, use `SHOW COLLATION <characterset>_<collation>` to display the attributes.

Unregistering a collation

To unregister an unwanted collation, use the DDL statement

```
DROP COLLATION <collation-name>
```

Rolling Your Own

It is possible to write your own character sets and collations and have the Firebird engine load them from a shared library, which should be named `fbintl2` in order to be recognised and linked.

It is also possible to implement custom character sets or collations using user-defined functions (UDFs) to transliterate input. The Firebird engine automatically uses functions that have names specially formatted to be recognized as character sets and collations. The name “`USER_CHARSET_nnn`” indicates a character set, while “`USER_TRANSLATE_nnn_nnn`” and “`USER_TEXTTYPE_nnn`” indicate character set + collation sequence. (*nnn* represents 3-digit numbers, usually in the range 128 to 254).

It is an advanced topic, beyond the scope of this book. The developer of the original *fbintl2* plug-in for custom character sets, David Brookestone Schnepfer, makes a do-it-yourself kit freely available, containing sample C code, mappings and instructions. These days, it is archived at several locations around the Web, including http://www.ibphoenix.com/resources/documents/attic/doc_39. Because the kit includes lucid instructions for creating character sets, it is also a useful reference if you plan to use the UDF approach to implement a custom character set.

Metadata Text Conversion

Firebird versions 2.0.x had two problems related to character sets and metadata extraction. You will know you have at least one of these problems if one of the first things you see on connecting to your newly upgraded database is a “Malformed string” error.

Problem 1: “raw” text in metadata

When creating or altering objects, text associated with metadata was not transliterated from the client character set to the system (`UNICODE_FSS`) character set of these BLOB columns. Instead, raw bytes were stored there. The database will have to be repaired in order to read the metadata correctly.

The types of text affected were PSQL sources, descriptions, text associated with constraints and defaults, and so on.

The problem can still occur if `CREATE` or `ALTER` operations are performed with the connection character set as `NONE` or `UNICODE_FSS` and you are using non-`UNICODE_FSS` data or if you process scripts containing strings that were written in an external editor and stored in ASCII or ANSI encoding.

Problem 2: malformed strings returned from text BLOBs

In reads from text BLOBs, transliteration from the BLOB character set to the client character set was not being performed.

Solutions

If your metadata and/or user BLOB text was created with encoding that was consistently wrong, it can be repaired using scripts distributed in the v.2.1 and 2.5 kits. The repair script for metadata is installed beneath your Firebird root directory in

```
/misc/upgrade/metadata/metadata_charset_create.sql
```

If you have user text BLOBs that are affected by Problem 2, you can use the same script as a model for writing another script to repair them. For more details, refer to the section Metadata Repair in Chapter 5, **Migration Issues**.

For v.2.5, the conversion for both problems was simplified with the addition of two new restore switches in the *gbak* utility. For details, refer to the topic The gbak method in the same section of Chapter 5.

CHAPTER

10

BLOBS AND ARRAYS

BLOB types—Binary Large Objects—are complex structures used for storing discrete data objects of variable size, which may be very large. They are “complex” in the sense that Firebird stores these types in two parts: a kind of hyperlink—referred to as a BLOB_ID—is stored with the owning row, while the actual data are stored apart from the row, often on one or many different database pages, keyed to the BLOB_ID.

Almost any sort of storable data can be stored in a BLOB: bit-mapped graphics images, vector drawings, sound files, video segments, chapter or book-length documents, or any other kind of multimedia information. Because a blob can hold different kinds of information, it requires special processing at the client for reading and writing.

BLOB types can, where practicable, store data files generated by other applications such as word processors, CAD software or XML editors. The gains can be the benefit of transaction control for dynamic data, protection from external interference, control of versions and the ability to access externally-created data through the medium of SQL statements.



BLOB types can not be indexed.

Array Types, discussed separately after BLOBs, are implemented in Firebird as a specially-formatted, system-managed BLOB subtype.

BLOBs and Subtypes

A BLOB *subtype* is a positive or negative integer that identifies the nature of the data contained in the column. Besides the two subtypes predefined for general use, Firebird has a number of subtypes that it uses internally. All of these internal subtypes have positive numbers.



Subtypes with positive numbers greater than 1 have specific purposes for storing and converting metadata. They should never be declared for user data. T

Supported User Subtypes

Out-of-the-box, Firebird makes two pre-defined BLOB types available for use in user tables, distinguished by the `sub_type` attribute (SQL keyword `SUB_TYPE`):

Table 10.1 Pre-defined BLOB subtypes

Definition	SQL Alias	Purpose
BLOB SUB_TYPE 0	BINARY	Generic BLOB type for storing any sort of data, including text. Commonly referred to as “an untyped binary BLOB”. Firebird is unaware of the contents. This is the default subtype.
BLOB SUB_TYPE 1	TEXT	More specialized sub-type for storing plain text. Equivalent to CLOB and MEMO types implemented by some other DBMSs. Recommended for use with application interfaces such as RAD components or search engines that provide special treatment for such types.

Custom Subtypes

Negative subtype numbers—from -1 to -32,768—are reserved for assigning to custom subtypes. Custom subtypes with negative numbers can be added to distinguish, identify and document special types of data objects that you want to store in BLOBs, such as HTML, XML or word-processor documents, JPEG, PNG images, etc.—the choice is up to you.



Firebird is unaware of what is stored in any BLOB and thus cannot perform any kind of validation on the streams that are written to BLOBs of a custom subtype.

BLOB Filters

Blob sub-typing also allows specific conversion from one subtype to another. Firebird has extensible support for automatic conversion between a given pair of blob sub-types in the form of *blob filters*. Blob filters are special kind of external function with a single purpose: to take a BLOB object in one format and convert it to a BLOB object in another format.

It is possible to create a blob filter that converts between a custom (negative) subtype and a predefined subtype, commonly the TEXT one.

A BLOB filter is like an external function (UDF) in most respects. The object code for blob filters is placed in shared object libraries. The filter, which is invoked dynamically when required, is recognized at database (not server) level by way of a declaration in metadata, of the form:

```
DECLARE FILTER <filter-name>
  INPUT_TYPE <sub-type> /* identifies type of object to be converted */
  OUTPUT_TYPE <sub-type> /* identifies type of object to be created */
  ENTRY_POINT '<entry-point-name>' /* name of exported function */
  MODULE_NAME '<external-library-name>'; /* name of BLOB filter library */
```

The writing and usage of BLOB filters are beyond the scope of this book. More information on the topic can be found by searching the Firebird knowledgebases.

Declaration Syntax

A BLOB is declared as a regular column for a table in a CREATE TABLE or ALTER TABLE statement. The following example defines two BLOB columns: BLOB1 with subtype 0 (the default), BLOB2 with Firebird subtype 1 (TEXT):

```
CREATE TABLE TABLE2 (
    .., /* other columns */
    BLOB1 BLOB, /* SUB_TYPE 0 */
    BLOB2 BLOB SUB_TYPE 1,
    ..);
```

The next defines a domain which is a text BLOB to store text in character set ISO_8859_1:

```
CREATE DOMAIN MEMO
    BLOB SUB_TYPE TEXT /*BLOB SUB_TYPE 1 */
    CHARACTER SET ISO_8859_1;
```

This SQL snippet shows how a local BLOB variable is declared in a PSQL module:

```
CREATE PROCEDURE ...
...
DECLARE VARIABLE AMEMO BLOB SUB_TYPE 1;
...
```

BLOB Segments

BLOB data are stored in a different format to regular column data and, usually but not always, apart from them. They are stored in blocks known as *segments* in one or more database pages, in a distinct row version that is unlike the format of a row of regular data. Segments are discrete chunks of unformatted data that are usually streamed by the application and passed to the API to be packaged for transmission across the network, one chunk at a time, in contiguous order.

In the regular row structure of the parent row, the BLOB is linked through a BLOB ID that is stored with the regular row data. A BLOB ID is a unique hexadecimal value that provides a cross-reference between a BLOB and the table it belongs to. On arrival at the server, segments are laid down in the same order as they are received, although not necessarily in chunks of the same size in which they were transported.

Where possible, the BLOB row versions are stored on the same page as parent row. However, large BLOBs can extend across many pages and this initial “blob row” may not contain actual data but an array of pointers to a data pages that are specialised as blob pages.



It is sometimes the case that the actual BLOB data will fit on the same data page as the record that owns the BLOB. Thus, it is not uncommon for Firebird to store small BLOBs on the same page as the record data.

Segment Size

When a BLOB column is defined in a table, the definition can optionally include the expected size of segments that are written to the column. The default—80 bytes—is really

quite arbitrary. Mythology says it was chosen because it was exactly the length of one line on a text terminal display!

The segment size setting does not affect Firebird’s performance in processing BLOBs on the server: the server does not use it at all. For DSQL applications—which is what most people write—you can simply ignore it or, if relevant, set it to some size that suits some local storage buffer in which your application stores BLOB data.

For DML operations—SELECT, INSERT and UPDATE—the length of the segment is specified in an API structure when it is written and can be any size, up to a maximum 65,535 bytes. Reusable classes for development environments such as FreePascal/Delphi, C++ and Java usually take care of BLOB segmentation in their internal functions and procedures if they have uses for it. If you are programming directly to the API, you might need to develop your own routines for constructing segments.

ESQL Applications

In databases for use with ESQL apps written for pre-processing by the *gpre* pre-processor—today a highly esoteric genre for Firebird development!—the segment size must be declared to indicate the maximum number of bytes that an application is expected to write to any segment in the column. Normally, an ESQL application should not attempt to write segments larger than the segment length defined in the table; doing so overflows the internal segment buffer, corrupting memory in the process. It may be advantageous to specify a relatively large segment, to reduce the number of calls to retrieve BLOB data.

The following statement creates two BLOB columns, BLOB1, with a default segment size of 80, and BLOB2, with a specified segment length of 1024:

```
CREATE TABLE TABLE2 (  
    BLOB1 BLOB SUB_TYPE 0,  
    BLOB2 BLOB SEGMENT SUB_TYPE TEXT SEGMENT SIZE 1024);
```

In this ESQL code fragment, an application inserts a BLOB segment. The segment length is specified in a host variable, `segment_length`:

```
INSERT CURSOR BCINS VALUES (:write_segment_buffer :segment_length);
```

When to Use BLOB Types

The BLOB is preferred to character types for storing text data of infinitely variable length. Because it is transported in “mindless chunks”, it is not subject to the 32 Kb length limit of strings, as long as the client application implements the appropriate techniques to pass it in the format required by the server for segmenting it.

Because the BLOB is likely to be stored apart from regular row data, it is not fetched automatically when row data are selected. Rather, the client requests the data on demand, by way of the `BLOB_ID`. Consequently, there is big “win” in the time taken to begin fetching rows from a SELECT, compared to the traffic involved when character types are used for storing large text items. On the other hand, some developers may consider it a disadvantage to have to implement “fetch-on-demand”.

When considering whether to use BLOB for non-text data, other issues arise. The convenience of being able to store images, sound files and compiled documents has to be balanced against the overhead it adds to backups. It may be an unreasonable design objective to store large numbers of huge objects that are never going to change.

Security

The idea that large binary and text objects are more secure when stored in BLOBs than when stored in filesystem files is largely an illusion. Certainly, they are more difficult to access from end-user tools. However, database privileges currently do not apply to BLOB and array types beyond the context of the tables to which they are linked by indirection. It is not absurd to suppose that a malicious hacker who gained access to open the database file could write application code that scanned the file for BLOB_IDs and read the data directly from storage, as external BLOB functions do.

Operations on BLOBs

A BLOB is never updated. Every update which “changes” a blob causes a new BLOB to be constructed, complete with a new BLOB_ID. The original BLOB becomes obsolete once the update is committed.

A BLOB column can be tested for NULL/NOT NULL and, from the “2” series forward, one text BLOB can be compared with another for equality. A BLOB cannot be compared with a string.

V.I.X No internal function exists to compare one BLOB with another in the older versions. Several BLOB UDFs are available from community download sites, including some that compare two BLOBs for equality.

String input to BLOB columns

As a broad rule, the writing of the contents of a BLOB is manipulated by client applications. Firebird does not update the contents of a BLOB: it is up to the client application to provide new contents for assignment to a BLOB when a statement updates a row and to package it using the specialised structures that are declared in the API for the purpose.

However, when accepting data for input to text BLOB columns by way of an INSERT or UPDATE operation, Firebird can take a string as input and transform it into a BLOB. For example:

```
INSERT INTO ATABLE (PK, ABLOB)
VALUES (99, 'This is some text.');
```

Note that passing a string to a stored procedure for an input argument that was defined as BLOB will cause an exception. The following, for example, will fail:

```
CREATE PROCEDURE DEMO
  (INPUTARG BLOB SUB_TYPE 1)
AS
BEGIN
  ...
END ^
COMMIT ^
EXECUTE PROCEDURE DEMO('Show us what you can do with this!') ^
```

Instead, either

- define the input argument as a VARCHAR and have your procedure submit the string to the INSERT or UPDATE statement itself; or
- have your client program take care of converting the string to a text BLOB. This will be the preferred solution if the length of the string is unknown.

Character set conversion

Conversion from one character set to another, when assigning either a string to a BLOB or the contents of a BLOB to another BLOB, is supported from the “2” series onward.

Manipulating output

For retrieving output from text BLOBs, the situation has improved with successive versions. For the “2” series and beyond, the manipulations at the server side that are available for strings have been progressively extended to work with text BLOBs, too.

Collations

Whilst Firebird always provided the means to specify the character set for a text BLOB, older versions cannot apply a collation sequence to the output. That changed with the “2” series, when something like the following became a valid way to retrieve BLOB text, in this case from a UTF8 BLOB:

```
select a_blob_column from table1
  where a_blob_column COLLATE UNICODE STARTING WITH 'Étu';
```

String functions

When the SUBSTRING() internal function was implemented in Firebird 1.5, it was also implemented for text BLOBs. Up to and including v.2.0, the substring returned is a string and is thus limited to the maximum size, in bytes, of the character type that receives the result.

In Firebird versions 2.1 and higher, most of the internal functions that apply to strings will also work with text BLOBs. Those that return strings as their result when operating on string input will return a text BLOB when operating on text BLOB input.



These enhancements mean that the SUBSTRING() function returns a BLOB, whereas it returns a string in versions 2.0 and 1.5. Legacy client and PSQL code written for the older server versions will present a migration issue for developers.

Concatenation

From v.2.1 onward, BLOBs can be concatenated together or to strings, producing a BLOB as the result. In versions 2.0.x and below, attempting this operation will result in a conversion error.

Upper-casing

From v.2.1 onward, BLOBs can be upper-cased in search expressions and query output, using the internal function UPPER(). In versions 2.0.x and below, attempting this operation will result in a conversion error.

External functions

Very few external functions exist for operating on BLOBs. The fbudf library includes just one: the function STRING2BLOB(), which takes a string or string expression as input and returns a BLOB of subtype TEXT. Its purpose was largely superseded when it became possible to pass a string argument to insert into or update text BLOB columns or variables.

Nevertheless, with careful attention to the byte-length limits for strings, it might have its uses in expressions that concatenate strings for storage into a BLOB column or variable.

Finding a suitable function

Appendix I, *Internal and External Functions*, has full details, with examples, of the functions available for BLOB inputs, both internally and through the UDFs that are distributed in the Firebird kits.

Some third-party UDF libraries have external functions that operate on BLOBs. You might like to check out some of the better-known third party external function libraries. The IBPhoenix web site maintains [summaries and links](#) for several of these.

Array Types

Firebird allows you to create homogeneous arrays of most data types. Using an array can enable multiple data items to be stored as discrete, multi-dimensional elements in a single column. At certain levels, Firebird can perform operations on an entire array, effectively treating it as a single element, or it can operate on an array slice, a subset of array elements. An array slice can consist of a single element, or a set of many contiguous elements.

ARRAY Types and SQL

Arrays are not a “natural” feature for a relational database. The relational model’s approach to packaging multi-dimensional data is to store it in rows of columns and retrieve it dynamically as sets. It has been well argued that arrays belong in client-side dynamic structures and have no place in relational databases. It has certainly remained an area of minimal attention and low demand during Firebird’s life to date.

Because Firebird does not implement any dynamic SQL syntax for operating on ARRAY types, performing DML and searches on them from a dynamic SQL (DSQL) interface is not simple. The Firebird API surfaces structures and functions to enable dynamic applications to work with them directly. Some RAD data access components—for example, IBOjects, a popular suite of components for use with Object Pascal (Delphi) and FreePascal language environments—provide classes encapsulating this area of API functionality.

Arrays are stored in a specialised BLOB structure. Multi-dimensional arrays are stored in row-major order.¹

Embedded SQL (ESQL)—which does not use the API structures and function calls—supports several static SQL syntaxes for operating on ARRAY types and integrating them with arrays declared in the host language. Like ESQL, arrays have remained an area of mild esoteric interest without engendering any remarkable interest in their existence.

For both dynamic and static applications, it is feasible—albeit not very practicable—to read array data into a stored procedure and return values that the client application can use. An example appears later in the topic entitled [Limited DSQL Access](#).

1. A flag exists in the API array descriptor that is said to enable m-d arrays to be stored in column-major order. It has never been properly documented and it is unknown whether it ever worked.

When to Use an Array Type

Using an array may be appropriate when:

- The data items naturally form a set of the same data type.
- The entire set of data items in a single database column must be represented and controlled as a unit, as opposed to storing each item in a separate column.
- Each item must also be identified and accessed individually.
- There is no requirement to access the values individually in triggers or stored procedures, or you have external functions that enable such access

Eligible Element Types

An array can contain elements of any Firebird data type except BLOB. Arrays of ARRAY are not supported. All of the elements of a particular array are of the same data type.

Defining Arrays

An array can be defined as a domain (using CREATE DOMAIN) or as a column, in a CREATE TABLE or ALTER TABLE ADD COLUMN statement. Defining an array domain or column is similar to defining any other, with the additional inclusion of the array dimensions. Array dimensions are enclosed in square brackets following data type specification.

Example The following statement defines both a regular character column, and a one-dimensional character array column containing eight elements:

```
CREATE TABLE ATABLE (  
    ID BIGINT,  
    ARR_CHAR(14)[8] CHARACTER SET OCTETS  
); /* stores 1 row * 8 elements */
```

Multi-dimensional Arrays

Firebird supports multi-dimensional arrays, arrays with 1 to 16 dimensions. For example, the following statement defines three INTEGER array columns with two, three, and four dimensions respectively:

```
CREATE TABLE BTABLE (  
    /* stores 4 rows X 5 elements = 20 elements */  
    ARR_INT2 INTEGER[4,5],  
    /* 6 layers, each 4 rows X 5 elements = 120 elements */  
    ARR_INT3 INTEGER[4,5,6],  
    /* 7 storeys, each 6 layers of 4 rows * 5 elements = 840 elements */  
    ARR_INT6 INTEGER[4,5,6,7]  
);
```



Firebird stores multi-dimensional arrays in row-major order. Some host languages, such as FORTRAN, expect arrays to be in column-major order. In these cases, care must be taken to translate element ordering correctly between Firebird and the host language.

Specifying Subscript Ranges for Dimensions

Firebird's array dimensions have a specific range of upper and lower boundaries, called *subscripts*. Dimensions are 1-based by default, i.e. the first element of the dimension array of *n* elements has subscript 1, the second element, 2, and the last element, *n*. For example, the following statement creates a table with a column that is an array of four integers:

```
CREATE TABLE TABLEC (
  ARR_INT INTEGER[4]
);
```

The subscripts for this array are 1, 2, 3, and 4.

Custom (explicit) subscript boundaries

A custom set of upper and lower boundaries can be defined explicitly for each array dimension when an array column is created. For example, C and Pascal programmers, familiar with zero-based arrays, might want to create array columns with a lower boundary of zero to map transparently to array structures in application code.

Both the lower and upper boundaries of the dimension are required when defining custom boundaries, using the following syntax pattern:

```
[lower:upper]
```

The following example creates a table with a single-dimension, zero-based array column:

```
CREATE TABLE TABLED (
  ARR_INT INTEGER[0:3]
); /* subscripts are 0, 1, 2, and 3. */
```

Each dimension's set of subscripts is separated from the next with commas. For example, the following statement creates a table with a two-dimensional array column where both dimensions are zero-based:

```
CREATE TABLE TABLEE (
  ARR_INT INTEGER[0:3, 0:3]
);
```

Storage of ARRAY Columns

As with other types implemented as BLOB, Firebird stores an array ID with the non-BLOB column data of the database table, that points to the page(s) containing the actual data.

Updates

As with other BLOB types, the Firebird engine cannot course through the data seeking successive individual target elements for conditional or selective update. However, in a single DML operation, it is possible to isolate one element or a set of contiguous elements—known as a *slice*—and target that slice for update.

Inserts

INSERT cannot operate on a slice. When a row is inserted in a table containing ARRAY columns, it is necessary to construct and populate the array entirely, before passing it in the INSERT statement.

Accessing Array Data

Some application interfaces do provide encapsulation of the API functions and descriptors and limited read access is possible from stored procedures.

The array descriptor

The API exposes the array descriptor structure for describing to the server the array or array slice to be read from or written to the database. It is presented to programmers in the C language API header file, `ibase.h`, located in the `../include` sub-directory of your Firebird installation.

Documentation

- The **InterBase® 6 API Guide** (ApiGuide.pdf), published by Borland Software Corp, provides detailed instructions for manipulating arrays through the API structures.
- More information about using arrays in embedded applications can be obtained from the **Embedded SQL Guide** (EmbedSql.pdf), a companion volume in the Borland set.

These documents can be downloaded from the archives at the IBPhoenix web site, at the URL <http://www.ibphoenix.com/resources/documents/attic>.

Limited DSQL Access

The following example is a simple demonstration of how a dynamic SQL application might get limited read access to an array slice through a stored procedure:

```
create procedure getcharslice(
    low_elem smallint, high_elem smallint)
returns (id integer, list varchar(50))
as
declare variable i smallint;
declare variable string varchar(10);
begin
    for select a1.ID from ARRAYS a1 into :id do
        begin
            i= low_elem;
            list = '';
            while (i <= high_elem) do
                begin
                    select a2.CHARARRAY[:i] from arrays a2
                    where a2.ID = :id
                    into :string;
                    list = list||string;
                    if (i < high_elem) then
                        list = list||',';
                    i = i + 1;
                end
            suspend;
        end
    end
```

end

CHAPTER

11

DOMAINS

Domains in Firebird are akin to the concept of “user-defined data types”. Although it is not possible to create a new data type, with a domain you can package a set of attributes with those of an existing data type, give it an identifier and, thereafter, use it in place of the data type parameter to define columns for any table.

Domain definitions are global to the database—all columns in any table which are defined with a particular domain will have completely identical attributes unless modified in the table definition by overrides.

Column-level overrides to domain attributes are discussed later in this chapter.

Columns based on a domain definition inherit all attributes of the domain, which can be:

- data type (required)
- a default value for inserts
- NULL status
- CHECK constraints
- character set (for character and text BLOB columns only)
- collation sequence (for character columns only)

You cannot apply referential integrity constraints to a domain.

Benefits of Using Domains

The benefits for encapsulating a data definition are obvious. For a simple but common example, suppose your design calls for a number of small tables where you want to store the text descriptions of enumerated sets—“type” tables—account types, product types, subscriptions types, etc. You have decided that each member of each of these sets will be keyed on a three-character upper-case identifier which points to a character field having a maximum of 25 characters.

All that is required is to create two domains:

- the domain for the pointer will be a CHAR(3) with two attributes added—a NOT NULL constraint because you are going to use it for primary and lookup keys and a CHECK constraint to enforce upper-case. For example,

```
CREATE DOMAIN Type_Key AS CHAR(3) NOT NULL
CHECK (VALUE = UPPER(VALUE));
```

- the description domain will be a VARCHAR(25). You want it to be non-nullable because the tables you want to use it in are control tables:

```
CREATE DOMAIN Type_Description AS VARCHAR(25) NOT NULL
```

Once you have these domains defined, all of your type-lookup tables can have similar definitions; and all tables which store lookup keys to these tables will use the matching domain for the key column:

```
CREATE TABLE TRANSAC_TYPE (
    TRANSAC_TYPE TYPE_KEY,
    DESCRIPTION TYPE_DESCRIPTION
);
CREATE TABLE CURRENCY (
    CURRENCY_CODE TYPE_KEY,
    DESCRIPTION TYPE_DESCRIPTION,
    EXCHANGE_FACTOR DECIMAL(6,4)
);
```

Creating a Domain

The data definition language (DDL) syntax pattern for creating a domain is:

```
CREATE DOMAIN <domain-identifier> [AS] <data-type>
[DEFAULT literal | NULL | USER]
[NOT NULL] [CHECK (<dom-search-condition>)]
[CHARACTER SET {<character-set> | NONE}]
[COLLATE collation];
```

Domain identifier

When you create a domain in the database, you must specify an identifier for the domain which is globally unique in the database. Developers often use a special prefix or subscript in domain identifiers, to facilitate self-documentation—the characters “D_” are suggested. For example,

```
CREATE DOMAIN D_TYPE_IDENTIFIER...
CREATE DOMAIN D_TYPE_DESCRIPTION...
```

Data type for the domain

The data type is the only other required attribute that must be set for the domain—all other attributes are optional. It specifies the SQL data type that will apply to column defined using the domain. Any Firebird native data type can be used.

The following statement creates a domain that defines an array of CHARACTER type:

```
CREATE DOMAIN DEPTARRAY AS CHAR(31) [4:5];
```

The next statement creates a BLOB domain with a text subtype that has an assigned character set: overriding the default character set of the database. It effectively creates a specialized memo type for storing Japanese text:

```
CREATE DOMAIN DESCRIPT_JP AS BLOB SUB_TYPE TEXT
    CHARACTER SET SJIS;
```

The DEFAULT attribute

A domain can define a default value that the server will use when inserting a new row if the INSERT statement does not include the column in its specification list. Defaults can save time and error during data entry. For example, a possible default for a DATE column could be today's date, or to write the CURRENT_USER context variable into a UserName column.

Values for DEFAULT can be:

a constant (literal): The default value is a user-specified string, numeric value, or date value—often used for placing a “zero value” into a non-nullable column

- CURRENT_TIMESTAMP, CURRENT_DATE, CURRENT_TIME or a Firebird predefined date literal—see Chapter 8, *[Date and Time Types](#)*
- USER, CURRENT_USER or CURRENT_ROLE (if roles are applicable)
- CURRENT_CONNECTION, CURRENT_TRANSACTION

It is possible to specify NULL as a default, but it is redundant, since nullable columns are initialized as NULL by default anyway. Furthermore, an explicit NULL default can cause conflicts when a column using the domain needs to be defined with a NOT NULL constraint (see below).

The following statement creates a domain that must have a positive value greater than 1,000. If the INSERT statement does not present a VALUE, the column will be assigned the default value of 9,999:

```
CREATE DOMAIN CUSTNO AS INTEGER
    DEFAULT 9999
    CHECK (VALUE > 1000);
```



If your operating system supports the use of multi-byte characters in user names, or you have used a multi-byte character set when defining roles, then any column into which these defaults will be stored must be defined using a compatible character set.

When defaults won't work

It is a common mistake to assume that a default value will be used whenever Firebird receives NULL in a defaulted column. When relying on defaults, it must be understood that a default will be applied

- only upon insertion of a new row AND
- only if the INSERT statement does not include the defaulted column in its column list

If your application includes the defaulted column in the INSERT statement and sends NULL in the values list, then NULL will be stored—or cause an exception in a non-nullable column—regardless of any default defined.

The NOT NULL attribute

Include this attribute in the domain if you want to force all columns created with this domain to contain a value.

NULL—which is not a *value*, but a *state*—will always be disallowed on any column bearing the NOT NULL attribute.



You cannot override the NOT NULL attribute on a domain but you can override a nullable domain definition when using it for a column definition. Consider the benefit of not including it in the domain's attributes, thereby leaving it as an option to add the attribute when columns are defined.

CHECK data conditions

The CHECK constraint provides wide scope for providing domain attributes that restrict the content of data which can be stored in columns using the domain. The CHECK constraint sets a search condition (dom-search-condition) that must be true before data can be accepted into these columns.

Syntax patterns for CHECK constraints

```
<dom-search-condition> =
    VALUE <operator> <val>
    | VALUE [NOT] BETWEEN <val> AND <val>
    | VALUE [NOT] LIKE <val> [ESCAPE <val>]
    | VALUE [NOT] IN (<val> [, <val> ...])
    | VALUE IS [NOT] NULL
    | VALUE [NOT] CONTAINING <val>
    | VALUE [NOT] STARTING [WITH] <val>
    | (<dom-search-condition>)
    | NOT <dom-search-condition>
    | <dom-search-condition> OR <dom-search-condition>
    | <dom-search-condition> AND <dom-search-condition>
<operator> = {=|<|>|<=|>=|!<|!>|<>|!=}
```

The VALUE keyword

VALUE is a placeholder for any constant or variable value or expression which would be submitted through SQL for storing in a column defined using the domain. The CHECK constraint causes VALUE to be validated against the restrictions defined in the conditions. If validation fails, an exception is raised.

If NULL is allowed in lieu of a value, it must be accommodated in the CHECK constraint. For example:

```
CHECK ((VALUE IS NULL) OR (VALUE > 1000));
```

The next statement creates a domain that disallows any input value of 1000 or less:

```
CREATE DOMAIN CUSTNO AS INTEGER
CHECK (VALUE > 1000);
```

The next statement restricts VALUE to being one of four specific values:

```
CREATE DOMAIN PRODTYPE AS VARCHAR(8) NOT NULL
CHECK (VALUE IN ('software', 'hardware', 'other', 'N/A'));
```

A validation condition can be made to search for a specific pattern in a string input. For example, the next validation check enforces a rule that requires a bracketed area code to precede telecom numbers, e.g. (02)43889474:

```
CREATE DOMAIN TEL_NUMBER AS VARCHAR(18)
CHECK (VALUE LIKE '(0%)%');
```


Multiple CHECK conditions

A domain can have only one CHECK clause but multiple conditions can be ANDed or ORed within this single clause. Care is needed with bracketing the condition expressions to avoid getting logical exceptions when the DDL statement is prepared.

For example, this statement fails:

```
create domain rhubarb as varchar(20)
  check (value is not null) and (value starting with 'J');
```

It excepts with a “oken unknown” error upon the word “and”. The corrected statement encloses the entire list of conditions within outer brackets and succeeds:

```
create domain rhubarb as varchar(20)
  check ((value is not null) and (value starting with 'J'));
```



The example above checks that the incoming value is not null. This is fine, but using the NOT NULL constraint directly is more powerful for application interfaces. The API can inform the client application at Prepare time of a NOT NULL constraint, whereas CHECK validation triggers do not fire until a DML request is actually posted to the server.

A domain's CHECK constraint cannot be overridden by one declared during column definition. However, a column can extend its use of the domain's CHECK constraint by adding its own CHECK conditions.

Dependencies in CHECK constraints

In tables, CHECK constraints can be defined legally with expressions referring to other columns in the same table or, less desirably, referring to columns in other database objects (tables, stored procedures).

Domains, of course, cannot refer to other domains. It is possible, although almost always unwise, to define a domain that refers to a column in an existing table. For example:

```
create domain rather_silly as char(3)
  check (value in (select substring(registration from 1 for 3)
    from aircraft));
```

Conceptually, it's not such a wild concept to use select expression in domains. Firebird allows it but it really does not follow through and implement it as an integrated design. It is an accidental by-product, not a feature.

As a database design approach, it integrates poorly with the referential integrity features that are purposefully implemented. Foreign key relationships enforce existence rules in all areas, whereas the scope of a CHECK constraint is limited to data entry.

CHECK constraints with inter-table dependencies would be disabled on restoring the database from a backup. They would silently fail to compile because the dependent tables had not yet been recreated. To put them back into effect, they would have to be reinstated manually, by some means. Implementing such checks at domain level has mind-boggling implications.

In some situations, where the dependency applies to a highly static table whose name is low in alphabetical sequence (*gbak* restores tables in alphabetical order), such a CHECK condition might be faintly arguable. The problem remains, that the domain has no control over what happens to data stored in tables beyond the event of checking the data coming into columns that use it.

There is also the possible future situation of wishing to extract SQL metadata scripts directly from the database (perhaps you lost the scripts used to create the database, or they have gone out of synch with the current database structure). In such situations it is very

difficult for any automated extract procedure to produce a script with the correct sequence to rebuild the database structure.

If you absolutely need to use this kind of check condition, apply it as an extra condition when you declare the column. Preferably, evaluate all of the alternatives—including the hand-coding of referential triggers in cases where foreign keys on lookup fields would cause recognized problems with index selectivity.

The CHARSET/CHARACTER SET attribute

For systems that need to be concerned about multiple character sets inside a single database, declaring character-set-specific domains for all of your text columns (CHAR, VARCHAR, BLOB SUB_TYPE 1 and arrays of character types) can be a very elegant way to deal with it. Refer to Chapter 9, *Character Types*, for character set definition syntax.

The COLLATE attribute

A COLLATE clause in a domain creation statement specifies an explicit collation sequence for CHAR or VARCHAR domains. You must choose a collation that is supported for the domain's declared, inherited or implied character set.

Refer to the previous chapter for COLLATE definition syntax. For a list of the collation sequences available for each character set, see Appendix VI, *Character Sets and Collations*.

Using a Domain

A domain can be used in place of a native data type in any column definition and it is the recommended way to design your data. From v.2.1 forward, you can also apply domain definitions to the declaration of variables in procedural SQL (PSQL) modules.

Domains in Column Definitions

The best way to illustrate the use of a domain in a column definition is with an example. In a certain database, SYSUSER is a domain of up to 31 characters, having a DEFAULT value which is to be obtained by reading the context variable CURRENT_USER:

```
CREATE DOMAIN SYSUSER AS VARCHAR(31) DEFAULT CURRENT_USER;
```

A table is defined, having a column UPDATED_BY which uses this SYSUSER domain:

```
CREATE TABLE LOANS (  
    LOAN_DATE DATE,  
    UPDATED_BY SYSUSER,  
    LOAN_FEE DECIMAL(15,2)  
);
```

A client submits an INSERT statement for the LOANS table:

```
INSERT INTO ORDERS (LOAN_DATE, LOAN_FEE)  
VALUES ('16-MAY-2014', 10.75);
```

Because the statement does not name the UPDATED_BY column in its column list, Firebird automatically inserts the user name of the current user, ALICEFBIRD:

```
SELECT * FROM LOANS;
```

returns
16-MAY-2014 ALICEFBIRD 10.75

Domain overrides

Columns defined using a domain can override some attributes inherited from the domain, by replacing an inherited attribute with an equivalent attribute clause. The column definition can also be extended by adding further attributes.

Table 11.1 shows which domain attributes can be overridden.

Table 11.1 Domain attributes and column overrides

Attribute	Override?	Notes
Data type	No	
DEFAULT	Yes	
CHARACTER SET	Yes	Can also be used to revert a column to database default
COLLATE	Yes	
CHECK	No	Use a regular CHECK clause in CREATE or ALTER TABLE statement to extend CHECK clause with more conditions
NOT NULL	No	If you want to be able to have a domain nullable in some usages and non-nullable in others, let the domain default to nullable and apply NOT NULL as a column override during CREATE or ALTER TABLE

The following example shows how to extend the attributes of a column that is being defined to use a domain, using an earlier domain definition example:

```
CREATE DOMAIN TEL_NUMBER AS VARCHAR(18)
CHECK (VALUE LIKE '(0%)%');
```

Let's say we want to define a table having a telecom number column in it. We want the domain's attributes but we also want to allow for telephone numbers where keypad mnemonics have been used to express some numerals as alphabetic characters. For this usage we want to ensure that any non-numeral characters are input in upper case:

```
CREATE TABLE LIBRARY_USER (
  USER_ID INTEGER NOT NULL,
  ... <other columns>,
  PHONE_NO TEL_NUMBER,
  CONSTRAINT CHK_TELNUM_UPPER
    CHECK (PHONE_NO = UPPER(PHONE_NO))
);
```

Now, we have an extra CHECK validation on this particular column. This statement

```
INSERT INTO LIBRARY_USER (USER_ID, PHONE_NO)
VALUES (99, '(02) 4388 wish');
```

fails, because the extra CHECK constraint requires the phone number to be '(02) 4388 WISH'.

Domains in PSQL Variable Declarations

From Firebird 2.0 onward you can use a domain in place of a native data type when declaring arguments and variables in procedural language (PSQL)—stored procedures, triggers and code blocks for use in EXECUTE BLOCK passages. The argument can be declared one of two ways. According to your requirements, you can declare the argument or variable using either

- the domain identifier alone, in lieu of the native data type identifier, to have the variable inherit all of the attributes of the domain; or
- the data type of the domain, without inheriting any CHECK constraints or DEFAULT value defined in the domain, by including the TYPE OF keyword in the declaration

The syntax pattern for declarations is:

`data-type ::= <native-data-type> | TYPE OF <domain_name>`

Examples Using our TEL_NUMBER domain from previous examples:

```
CREATE DOMAIN TEL_NUMBER AS VARCHAR(18)
CHECK (VALUE LIKE '(0%)%');
```

Our (somewhat fatuous!) example shows how to declare two input variables, TEL_NO1 and TEL_NO2, two local variables, V1 and V2, and two output variables, OUT1 and OUT2, using both options. TEL_NO1 and V1 use the complete domain definition and expect the input variable to include the bracketed area prefix, while TEL_NO2 and V2 expect nothing other than a string of up to 18 characters:

```
CREATE PROCEDURE MYPROC (
    TEL_NO1 TEL_NUMBER,
    TEL_NO2 TYPE OF TEL_NUMBER)
RETURNS (
    OUT1 TEL_NUMBER,
    OUT2 TYPE OF TEL_NUMBER)
AS
DECLARE VARIABLE V1 TEL_NUMBER;
DECLARE VARIABLE V2 TYPE OF TEL_NUMBER;
BEGIN
...
END
```



The feature is not supported if you are operating on a database whose on-disk structure is less than 11.1.

Databases of ODS 11.1 or more include a column named RDB\$VALID_BLR in the system tables RDB\$PROCEDURES and RDB\$TRIGGERS to indicate whether the module's stored binary language representation is valid after an ALTER DOMAIN operation. Its value is displayed in the SHOW PROCEDURE and SHOW TRIGGER outputs in isql versions 2.1 and higher.

Using Domains with CAST()

In the “2” series and onward, you can use a CAST() expression to convert a column or an expression to the data type of a domain, in lieu of a native data type.

The syntax pattern is:

`CAST (<expression> AS TYPE OF <domain-name>)`

As with its application to PSQL variable declarations, conversion to TYPE OF applies just the underlying data type. Casting using the singular TYPE keyword is not supported in current versions.

Example Using our TEL_NUMBER domain again:
`CAST :Input_Number as TYPE OF TEL_NUMBER`
 will take a parameter of up to 18 characters or digits.

Where domains won't work

A domain cannot be used

- with the CAST (aValue AS <another type>) function in versions prior to the '2' series
- to define the data type of the elements of an ARRAY. A domain can itself be an ARRAY type, however.
- to declare the type of an argument or variable in a trigger or stored procedure in Firebird servers older than v.2.1 and databases with on-disk structure version lower than 11.1.

Defining a BOOLEAN Domain

Up to and including release 2.5, Firebird does not provide a Boolean type. A Boolean-styled domain is ideal, because you can define attributes that will be consistent across all tables. It is recommended to use the smallest possible data type: a CHAR for T[true]/F[alse] or Y[es]/N[o] switches or a SMALLINT 1/0 pairing. The following examples suggest ways you might implement your Booleans.

Example 1: a two-phase switch that defaults to 'F' (false):

```
CREATE DOMAIN D_BOOLEAN AS CHAR
  DEFAULT 'F' NOT NULL
  CHECK(VALUE IN ('T', 'F'));
```

Example 2: a three-phase switch that allows UNKNOWN (i.e. NULL):

```
CREATE DOMAIN D_LOGICAL AS SMALLINT
  CHECK (VALUE IS NULL OR VALUE IN (1,0));
```

Example 3: a three-phase switch that represents UNKNOWN as a value:

```
CREATE DOMAIN D_GENDER AS CHAR(4)
  DEFAULT 'N/K' NOT NULL
  CHECK (VALUE IN ('FEM', 'MASC', 'N/K'));
```



Don't use BOOLEAN, UNKNOWN, TRUE or FALSE as names for your Boolean domains. They are either already reserved words in Firebird or are likely to become so in a future version.

Changing a Domain Definition

The DDL statement ALTER DOMAIN can be used to change any aspect of an existing domain except its NOT NULL setting. Changes that you make to a

domain definition affect all column definitions based on the domain that have not been overridden at the table level.

Whether altering a domain is a desirable thing to do is your decision. However, keep in mind that every definition or declaration in the database that uses that domain is a dependency on it. It may much more of a major work to remove the dependencies than to define an entirely new domain and use ALTER TABLE ALTER COLUMN statements to switch the columns to the new domain.

Existing data are not changed by ALTER DOMAIN. For some changes, it is often essential to run an update over the tables afterwards to update the existing data so that it will be consistent with new data that will be created under the new rules.

A domain can be altered by its creator, the SYSDBA user or any user with operating system root privileges.



*On Windows, “users with operating system root privileges” refers to Windows users with the optional escalated privileges (RDB\$ADMIN role) granted on the database, on a server that is configured, via `firebird.conf`, for trusted or mixed **Authentication**. It is applicable only to versions 2.1+ and needs to be explicitly provisioned.*

Using ALTER DOMAIN you can:

- Rename the domain (if it has no dependencies)
- Modify the data type
- Drop an existing default value.
- Set a new default value.
- Drop an existing CHECK constraint.
- Add a new CHECK constraint.



The only way to “change” the NOT NULL setting of a domain is to drop the domain and recreate it with the desired combination of features.

The syntax pattern is:

```
ALTER DOMAIN { name | old_name TO new_name }{
  [SET DEFAULT {literal | NULL | USER | etc.}]
  | [DROP DEFAULT]
  | [ADD [CONSTRAINT] CHECK (<match_conditions>)]
  | [DROP CONSTRAINT]
  | TYPE data_type
};
```

Examples of valid changes to domains

This statement sets a new default value for the BOOK_GROUP domain:

```
ALTER DOMAIN BOOK_GROUP SET DEFAULT -1;
```

In this statement, the name of the BOOK_GROUP domain is changed to PUBL_GROUP:

```
ALTER DOMAIN BOOK_GROUP TO PUBL_GROUP;
```

Constraints on altering data types

The TYPE clause of ALTER DOMAIN allows the data type to be changed to another, permitted data type. For permitted type conversions, refer to the topic [Changing column and](#)

domain definitions in Chapter 6, especially the table Valid data type conversions using ALTER COLUMN and ALTER DOMAIN.

Any type conversion that could result in data loss is disallowed. For example, the number of characters in a domain could not be made smaller than the size of the largest value in any column that uses it.



Converting a numeric data type to a character type requires a minimum length for the character type that can accommodate all of the characters “created” by conversion of the highest possible number. For example, The following statement changes the data type of a BIGINT domain, BOOK_ISBN, to VARCHAR(19), in order to accommodate the maximum number of digits in a BIGINT:

```
ALTER DOMAIN BOOK_ISBN TYPE VARCHAR(19);
```

Dropping a Domain

DROP DOMAIN removes an existing domain definition from a database, provided the domain is not currently used for any column definition in the database.

To avoid exceptions, use ALTER TABLE to drop any table columns using the domain before executing DROP DOMAIN. The best way to do this in a single task is with a DDL script. You can read more about DDL scripts in Chapter 24..

A domain can be dropped by its creator, the SYSDBA user or (on Linux/UNIX) any user with operating system root privileges. The syntax pattern is:

```
DROP DOMAIN <domain-name>
```

The following statement deletes an unwanted domain:

```
DROP DOMAIN rather_silly;
```




The
Firebird Book
A Reference for Database Developers

SECOND EDITION

PART III



A Database & Its Objects

DESIGNING AND DEFINING A DATABASE

A database, of course, stores data. However, data alone have no use unless they are stored according to some rules that, first, capture their meaning and value and, next, allow them to be retrieved consistently. A database having an existence within the context of a database management system (DBMS), such as Firebird, comprises a lot of “things” besides data.

Firebird is a *relational* database management system. As such, it is intended to support the creation and maintenance of abstract data structures, not just to store data but also to maintain relationships and to optimize the speed and consistency with which requested data can be retrieved and returned to SQL client applications.

Designing a Database

Although relational databases are very flexible, the only way to guarantee data integrity and satisfactory database performance is a solid database design—there is no built-in protection against poor design decisions. A good database design:

Satisfies the users’ content requirements for the database. Before you can design the database, you must do extensive research on the requirements of the users and how the database will be used. The most flexible database designs today evolve during a well-managed process of analysis, prototyping and testing that involves all of the people who will use it.

Ensures the consistency and integrity of the data. When you design a table, you define certain attributes and constraints that restrict what a user or an application can enter into the table and its columns. By validating the data before it is stored in the table, the database enforces the rules of the data model and preserves data integrity.

Provides a natural, easy-to-understand structuring of information. Good design makes queries easier to understand, so users are less likely to introduce inconsistencies into the data, or to be forced to enter redundant data. .

Satisfies the users' performance requirements. Good database design ensures better performance. If tables are allowed to be too large (wide), or if there are too many (or too few) indexes, long waits can result. If the database is very large with a high volume of transactions, performance problems resulting from poor design are magnified.

Insulates the system from design mistakes in subsequent development cycles.

Description and analysis

A database abstractly represents a world of organization, relationships, rules and processes. Before reaching the point of being capable of designing the structures and rules for the database, the analyst-designer has much to do, working with people involved to identify the real-life structures, rules and requirements from which the database design will be rendered. The importance of scrupulous description and analysis can not be over-emphasized.

Logical data analysis is an iterative process of refining and distilling the world of inputs, tasks and outputs whose scope is to be encompassed by the database. Large, haphazard structures of information are reduced progressively to smaller, more specialized data objects and are gradually mapped to a data model.

An important part of this reduction process involves *normalization*—splitting out groups of data items with the goal of establishing essential relationships, eliminating redundancies and associating connected items of data in structures that can be manipulated efficiently.

This phase can be one of the most challenging tasks for the database designer, especially in environments where the business has been attuned to operating with spreadsheets and desktop databases. Regrettably, even in established client/server environments, too many poorly-performing, corruption-prone databases are found to have been “designed” using reports and spreadsheets as the basis.

Data model <> database

The “world” which evolves during description and analysis provides a logical blueprint for your data structures. It is a “given” that the logical model should discover every relationship and set. It is usually a mistake—and a trap inherent in many CASE tools—to translate the data model blindly into a database schema. In sophisticated data management systems like Firebird, a table structure does not always represent the optimal object from which data should be retrieved. Queries, views, global temporary tables (GTTs), calculated columns and “selectable” stored procedures are just a few of the retrieval and storage mechanisms available that will influence how you implement the model in the physical design.

Even an excellent data model will lack flexibility and will under-perform if it does not take into account the power and economy of the server's dynamic capabilities. Dynamic structures for the selection and manipulation of data are the arteries of a client/server database.

One database or many?

A single Firebird server can control multiple databases within the physical filesystem that is its host. It is not unusual in large enterprises to run multiple databases serving separated divisional subsystems. Because one database is not aware of the objects and dependencies in another, it takes careful design, planning and balancing of system resources and network services to integrate these independent systems. Typically, such databases are synchronized periodically by a replication system.

When designing, bear in mind that Firebird does not support queries that join or union tables across database boundaries. However, it does support simultaneous queries across multiple databases within one single transaction, with two-phase commit. It is thus possible for applications to accomplish tasks that work with data images from two or more databases and perform DML on one database using data read from another. For more details about cross-database transactions and two-phase commit, refer to Chapter 27, *Programming with Transactions*.

The Physical Objects

The time to start thinking about the physical design is when—and only when—the business of describing the requirements and identifying the data required and the processes they will be subject to is done.

Tables

A database table is usually visualized as a two-dimensional block consisting of columns (the vertical dimension) and rows (the horizontal dimension). The storage attributes of individual items of data are specified in columns—usually related to or dependent upon one another in some way—and rows. A table can have any number of rows (limit to 2^{32} for databases with and ODS lower than 11), or even no rows at all. Although every row in one table shares the specification of its columns with every other row, rows do not depend on other rows in the same table.



Firebird does support techniques for implementing self-referencing tables—row structures which enforce dependencies between rows within the same table. For details, refer to the topic Tree Structures in Chapter 15.

Files and pages

If you are moving to Firebird from a database system that implements tables by physically tabulating columns and rows in the file system, Firebird may bring some surprises. In Firebird, all data belonging to a single database are stored in one file, or set of linked files that is logically one file. In multi-file databases, there is no correlation between any specific database object and a particular member of the database file-set.

Within the boundaries of the file, the Firebird server engine manages evenly-sized blocks of disk known as **database pages**. It manages several different page “types”, according to the types of data it needs to store: regular columns for tables (data pages), BLOBs (blob pages) and indexes (index pages), for example. The engine allocates fresh blocks to itself from the host filesystem as required. All pages are the same size, regardless of type. **Page size** can be specified in the CREATE DATABASE statement, if you want a size larger or smaller than the default 4KB. It cannot be altered except by backing up the database and reconstructing it with a new page size, using the *gbak* utility.

Unlike the file-based data management systems, Firebird does not maintain table data in a tabulated format at all. Rows from one table may not be stored contiguously with other rows from the same table. Indeed, row data for a single table may be distributed among several files and several disks. The engine uses various types of **inventory pages** to store information about the physical locations of rows belonging to each table.

Columns and fields

Abstractly, a column is a constellation of attributes, defining the data item that can be stored in one specific cell location in the left-to-right structure of a table's row. However,

columns don't just exist in tables in the database. Each time a query is submitted to the database engine for processing, that query specifies a set of columns and one or more operations to be carried out on those columns. The columns do not have to be in the same left-to-right order as is defined for them in the table. For example, the statement

```
SELECT FIELD3, FIELD1, FIELD2 FROM ATABLE;
```

will output a set in the column order specified in the query. The query may specify columns from multiple tables, through joins, subqueries and unions. It may define columns that do not exist in the database at all, by computing them or even just by specifying them as named constants.

Some people use the term *field* when referring to a column, e.g. "I have a table TABLE1 which has three fields." Relational database textbooks often discourage the use of "field" as a substitute for "column", some preferring to use "field" to mean "the value in the column" or "the reference to a column".

In this book, "field" is used only as a term to generalize the concepts of column, argument and local variable; and to refer to output items that are constructed at run-time. "Column" is used to refer to the physical columns defined for tables.

Keys

Keys are the "glue" of relationships between tables.

The primary key

An essential part of the database design process is to abstract the logical model of the database to the point where, for each table, there is a single, unique column or composite column structure that distinguishes each row from every other row in the table. This unique column or column combination is the logical primary key. When you implement your physical model, you use the PRIMARY KEY constraint to tell the data management system which column or columns form this unique identifying structure. You may define only one PRIMARY KEY constraint per table. Syntax is discussed in Chapter 15, in the topic Constraints.

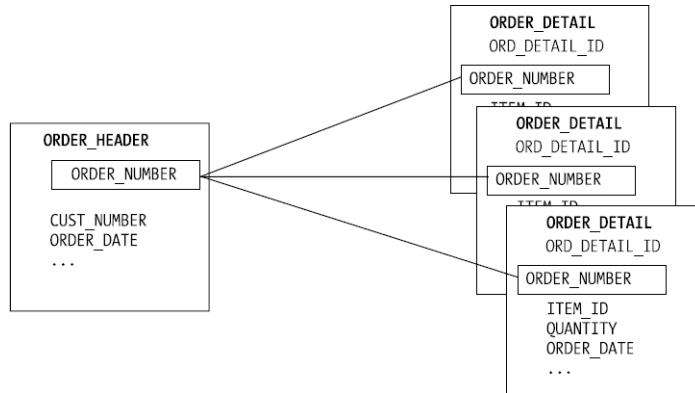
Other unique keys

It can happen in your data modeling process that, for various reasons, you end up with more than one unique column or structure in a table. For enforcing the required uniqueness on such columns or structures, Firebird provides the optional UNIQUE key constraint. It is effectively an alternative primary key and can be used in lieu of the primary key at times, if required.

Foreign keys

The "glue" that makes a relational database "relational" is foreign keys. This is the column or column structure that shows up in your data model on the "many" side of a one-to-many relationship. In the physical design, a foreign key matches up with the column or column structure of the primary key of the table on the "one" side of the relationship.

For example, in the following simple model, the detail lines of an order are linked to the order header by the ORDER NUMBER key.

Figure 12.1 Simple relational link

This model requires that each header row have a unique ORDER_NUMBER and that at least one order detail row exists for each order header row. Other rules may apply to the existence and linking. Firebird provides powerful trigger procedure capabilities for setting, conditioning and applying rules to relationships. Additionally, it can automate many of the typical rules governing relationships, using the FOREIGN KEY constraint with its optional action governing arguments. Underlying this constraint are system-generated referential integrity triggers. Firebird's referential integrity support is discussed briefly below and in detail in Chapter 17, *Referential Integrity*.

Making keys atomic

An important tenet of good relational database design is **keys atomicity**. In the context of primary and foreign keys, atomicity means that no key should have any meaning as data: it should have no other role or function except to be a key.

The column that your analysis determines to be the primary key, or an element of the primary key, almost always stores a data item that has some meaning. Take, for example, a table storing personal details:

```
CREATE TABLE PERSON (
    FIRST_NAME VARCHAR(30) NOT NULL,
    LAST_NAME VARCHAR(50) NOT NULL,
    PHONE_NUMBER VARCHAR(18) NOT NULL,
    ADDRESS_1 VARCHAR(50),
    ...);
```

The designer decides that the combination (FIRST_NAME, LAST_NAME, PHONE_NUMBER) is a good candidate for the primary key. People do share phone numbers but it is extremely unlikely that two people with identical first and last names would share the same number, right? So, the designer does this:

```
ALTER TABLE PERSON
    ADD CONSTRAINT PK_PERSON(LAST_NAME, FIRST_NAME, PHONE_NUMBER);
```

The first problem with this primary key is that every element of the primary has meaning. Every element is maintained by humans and may change or be misspelt. The two keys ('Smith', 'Mary', '43889474') and ('SMITH', 'Mary', '43889474') are not the same and will both be capable of being stored in this table. Which record gets changed if Mary gets married or changes her phone number?

The second problem is that this complex key has to be propagated, as a foreign key, to any tables that are dependent on PERSON. Not only is the integrity of the relationship at risk through alterations or errors in the data, it is a broad channel—potentially 98 characters—across which to implement the foreign key relationship.

Mandatory indexes are created to enforce keys. The real showstopper may occur if index key columns use multi-byte character sets or non-binary collations. Index widths are limited to one quarter of the database page size, or even to as little as 253 bytes if you are still using an old ODS 10 or 10.1 database. It is possible that such a key will be impossible because it is simply too wide.

Surrogate keys

The solution is to add an extra column to tables to accommodate an artificial or surrogate primary key: a unique, narrow column, preferably system-generated, that replaces (surrogates) the function of the theoretical primary key. Firebird provides GENERATOR objects, which can be implemented to maintain the required unique series of BIGINT numbers—a primary key of a mere 8 bytes or less.

Refer in Chapter 30, *Triggers*, to the topic *Implementing Auto-Incrementing Keys* for a common technique using generators to implement an auto-incrementing primary key, untouched by human hand.



The atomicity of keys should be enforced in applications by hiding them from users or, at least, making them read-only.

Surrogate keys vs natural keys – summary

Database developers tend to take strong positions in the arguments for and against using artificial keys. The author's position in favour of atomicity is probably evident. However, in the interest of fairness, the arguments are summarized here in Table 12.1.

Table 12.1 Surrogate (artificial) keys vs natural keys

Feature	Pro	Con
Atomicity	Surrogate keys carry no meaning as data and never change	Natural keys are inherently unstable because they are subject to human error and externally-imposed changes
Convenience	Natural keys carry information, reducing the necessity to perform joins or follow-up reads to retrieve data in context Natural keys are easier to use with interactive query tools	Surrogate keys carry no information beyond their linking function, necessitating joins or subqueries to retrieve the associated “meaningful” data
Key size	Surrogate keys are compact	Natural keys are characteristically large and often escalate into compound keys that complicate querying and schema

Feature	Pro	Con
Navigation	Surrogate keys provide clean, fast-tracking "navigation by code"	Natural keys are generally unsuitable for code-style navigation because of case, collation, denormalization and size issues
Normalization	Surrogate keys can be normalized throughout the database	Natural keys tend toward complexity, propagating denormalized data to foreign keys

Should you design databases with a mix of natural and artificial keys? The extreme view is to advise a consistent design approach—choose natural or artificial and apply the rule without exception. Yet a more moderate approach may offer the best of both worlds. It may be realistic to use a natural key for stable lookup or “control” tables that rarely change, are never required to participate in compound key structures and appear often in output.



When designing keys for a Firebird database, be reminded that keys are enforced by indexes and indexes have a size limit. Be aware that compounding, collation sequences and multi-byte international character sets reduce the number of characters of actual data that can be accommodated in an index.

Keys are not indexes

Keys are table-level constraints. The database engine responds to constraint declarations by creating a number of metadata objects for enforcing them. For primary keys and unique constraints, it creates a unique index on the column(s) assigned to the constraint. For foreign keys, it creates a non-unique index on the assigned columns, stores records for the dependency and creates triggers to implement the actions.

- The keys are the constraints
- The indexes are required to enforce the constraints



You should not create indexes of your own that duplicate the indexes created by the system to enforce constraints. This is such an important precaution from the performance perspective that it is reiterated in several places in this book. The topic Dropping an Index in Chapter 16 explains why duplicating these indexes can wreck the performance of queries.

Referential integrity

Accidental altering or deletion of rows that have dependencies will corrupt the integrity of your data. Referential integrity, generally, is a qualitative expression that describes the degree to which dependencies in a database are protected from corruption. However, in the context of this book, it refers to the inbuilt mechanisms for enforcing foreign key relationships and performing the desired actions when the primary key of a master row is changed or the row is deleted.

The syntax and implications of Firebird's formal referential integrity constraints are discussed in detail in Chapter 17.

Indexes and query plans

Indexes provide the database with navigational aids for searching large amounts of data and quickly assessing the best way to retrieve the sets requested by clients. Good indexing speeds things up, missing or bad indexes will slow down searches, joins and sorting.

As a relational database management engine, Firebird can link almost any column object to almost any other column object—the exceptions being the various BLOB types, including ARRAYS—by reference to their identifiers. However, as the numbers of rows, linking columns and tables in a query increase, so performance tends to slow down.

When columns that are searched, joined or sorted are indexed in useful ways, performance in terms of execution time and resource usage can be dramatically improved. It must also be said that poor indexing can hurt performance!

Firebird uses optimization algorithms that are largely cost-based. In preparing a submitted query, the optimizer calculates the relative costs of choosing or ignoring available indexes and returns a query plan to the client, reporting its choices. Although it is possible to design and submit your own plan to the optimizer—an important feature in RDBMS engines that use rule-based optimization—as a general rule the Firebird optimizer knows best. Custom plans tend to be most useful in detecting and eliminating problem indexes.

Index design and creation are discussed in Chapter 16.

Views

Firebird provides the capability to create and store pre-defined query specifications, known as views, which can be treated in most ways just as though they were tables. A view is a class of derived table that stores no data. For many tasks—especially those where access to certain columns in the underlying tables needs to be denied or where a single query specification cannot deliver the required degree of abstraction—views solve difficult problems.

Views and other types of run-time sets are discussed in Chapter 23.

Stored procedures and triggers

Stored procedures and triggers are modules of compiled, executable code that are executed on the server. The source code is a set of SQL language extensions known as procedural SQL, or PSQL.

Stored procedures can be executable or selectable. They can take input arguments and return output sets. Executable procedures execute completely on the server and optionally return a single-row set (a “singleton”) of constants on completion. Selectable procedures generate multiple-row sets of zero or more rows, which can be used by client applications in most ways like any other output set.

Triggers are a specialized form of PSQL module, which can be declared to execute at one or more of six stage/operation phases—before and after inserting, updating and deleting—during a data manipulation (DML) operation on the table that owns them. Clusters of triggers can be defined for each phase, to execute in a defined sequence. From release 1.5 forward, the behavior for any or all DML operations can be combined, with conditions, in a single “before” or “after” trigger module. Triggers do not accept input arguments nor return output sets.

Stored procedures can call other stored procedures. Triggers can call stored procedures that, in turn, can call other stored procedures. Triggers cannot be called from a client application nor from a stored procedure.

PSQL provides mechanisms for exception handling and callback events. Any number of exception messages can be defined as objects in the database using CREATE EXCEPTION statements. Callback events are created inside the PSQL module and applications can set up structures to “listen” for them.

For detailed discussion of writing and using PSQL modules, exceptions and events, refer to Part Six, *Programming on the Server*.

Generators (Sequences)

Generators—also known as sequences—are ideal for using to populate an automatically incrementing unique key or a stepping serial number column or other series. When Firebird generates a value in a series from a generator or sequence, that number can never be generated again in the same series.

Generators are declared in a database using a CREATE statement, just like any other database object.

```
CREATE GENERATOR AGenerator;
```

or

```
CREATE SEQUENCE ASequence;
```

A generator is a sequence is a generator—they are fully interchangeable. However, the syntaxes from this point on are not parallel. Using the SEQUENCE syntax is preferred because its compliance with standards should provide a benefit in interoperability with other database engines.

Generators can be set to any starting value:

```
SET GENERATOR AGenerator TO 1;
```

or

```
ALTER SEQUENCE Asequence RESTART WITH <new-value>
```

There are strong caveats against resetting generators once a series is in use—see below.

Calling for the next value

To call for the next value, either

- invoke the SQL function `GEN_ID(GeneratorName, n)`, where `GeneratorName` is the name of the generator and `n` is `BIGINT` (or, in dialect 1, an integer), specifying the size of the step. The query

```
SELECT GEN_ID(AGenerator, 2) from RDB$DATABASE;
```

returns a number that is two greater than the last generated number and increments the current value of the generator to the value it just generated.

- use a `NEXT VALUE FOR` operation on the sequence. For example, the following snippet from a trigger pulls the next value from the same generator used in the first method:

```
new.thing_id = NEXT VALUE FOR AGenerator;
```



NEXT VALUE FOR does not support a step (increment) other than 1. If your requirement calls for a different step increment, use the GEN_ID function.

Current value of a generator

```
SELECT GEN_ID(AGenerator, 0) from RDB$DATABASE;
```

returns the current value of the generator, without incrementing it. Since NEXT VALUE FOR does not allow a step value, there is no parallel SEQUENCE syntax to obtain this information.

Populating a variable

PSQL, Firebird's programming language, allows a value to be generated directly into a variable:

```
...
DECLARE VARIABLE MyVar BIGINT;
...
MyVar = GEN_ID(AGenerator, 1);
```

or

```
MyVar = NEXT VALUE FOR AGenerator'
```

More details about using generators in PSQL modules—especially triggers—can be found in Chapters 29 and 30.

Using GEN_ID() for negative stepping

The step argument of GEN_ID(..) can be negative. Thus, it is possible to set or reset a generator's current value by passing a negative argument as either an integer constant or an integer expression. This capability is sometimes used as a “trick” for meddling with generator values in PSQL, since PSQL does not allow DDL commands such as SET GENERATOR.

For example, the statement

```
SELECT
  GEN_ID(AGenerator,
    ((SELECT GEN_ID(AGenerator, 0) from RDB$DATABASE) * -1)
  )
from RDB$DATABASE;
```

causes the generator to be reset to zero.

Caveats about resetting generator values

The general rule of thumb about resetting generator values in production databases—whether through SQL, PSQL or some Admin interface—is DON'T.

The benefit of generator values is that they are guaranteed to be unique. Unlike any other user-accessible operation in Firebird, generators operate outside transaction control. Once generated, a number is “gone” and cannot be reverted by transaction rollback. This absolutely assures the integrity of number sequences, provided the generators are not tampered with.

Reserve the resetting of generators in a production database for the rare circumstances where a design requirement calls for it. For example, some older-style accounting systems pass journals into history tables with new primary keys, empty the journal table and reset

the primary key sequence to zero; or, in multi-site organizations, separated ranges of key values are allocated to each site in “chunks” to ensure key integrity on replication.



Never reset generators in an attempt to correct bugs or input errors or to “fill gaps” in a sequence.

Object Naming Conventions

The following conventions and associated limitations on naming database objects must be observed:

- Start each name with an alphabetic character (A–Z or a–z).
- Restrict object names to 31 characters. Some objects, such as constraint names, are restricted to 27 bytes in length when the on-disk structure is lower than 11.
- Allowable characters for database file names—as with all metadata objects in Firebird—include dollar signs (\$), underscores (_), 0 to 9, A to Z, and a to z..
- Observe uniqueness requirements within a database:
- In all cases, objects of the same type—all tables, for example—must be unique.
- Column identifiers must be unique within a table. All other object identifiers must be unique within the database.
- Avoid the use of reserved words, spaces, diacritic characters and any characters that fall beyond the 7-bit range:
 - In dialect 1, they cannot be used at all.
 - In dialect 3 you can delimit “illegal” identifiers using pairs of double-quote symbols. Details follow.

A full list of reserved words, version by version, is in Appendix II.

Optional SQL-92 delimited identifiers

In Dialect 3 databases, Firebird supports the ANSI SQL convention for optionally delimiting identifiers. To use reserved words, diacritic characters, case-sensitive strings, or embedded spaces in an object name, enclose the name in double quotes. It is then a delimited identifier. Delimited identifiers must always be referenced in double quotes.

Names enclosed in double quotes are case sensitive. For example:

```
SELECT "CodAR" FROM "MyTable"
```

is different from:

```
SELECT "CODAR" FROM "MYTABLE"
```

To quote or not to quote

The double-quoting convention for object identifiers was introduced in dialect 3 for compliance with standards. To those who have been used to the global case-insensitivity of other database systems, the feature is at best confusing and at worst exasperating.

If you define objects with double-quotes, you must use them everywhere and every time with double-quotes and perfect case-matching. Most experienced Firebird developers

recommend avoiding them, except in the occasional cases where you are stuck with using an “illegal” identifier. The choice is yours.

The case-matching exception

If you have double-quoted identifiers in all-uppercase, you can use them in SQL without the quotes and treat them as case-insensitive. The ability to do this comes from the way identifiers are stored in the internal schema tables and the sequence that the engine follows to resolve them during retrieval.



Most GUI database administration tools for Firebird provide the option to apply quoted identifiers automatically to all identifiers. One or two of these tools actually apply quoted identifiers by default to all database objects. Unless you have a strong reason to do this, it is recommended that you look for the ‘OFF switch’ and avoid quoted identifiers.

Dialect 1 In a dialect 1 database, reserved words, diacritic characters and embedded spaces in an object name are not permitted. You will get an exception if you try to use the double-quoting convention and case-sensitive identifiers are not supported.

File-naming Conventions for Databases

The established convention for naming Firebird database files on any platform is to apply the three-character suffix “.fdb” to the primary file and to name secondary files “.f01”, “.f02”, etc. This is only a convention—a Firebird database file can have any extension, or no extension at all.

Because of known problems on Windows servers, involving the SystemRestore feature actively targeting files with the suffix “.gdb”, developers are advised to change the traditional InterBase file suffix when migrating these databases to Firebird.

The name of the security database—security2.fdb since v.2.0, security.fdb in v.1.5.x and isc4.gdb in release 1.0.x—must not be changed. Unfortunately, Firebird 1.0.x requires the “.gdb” suffix and there is no workaround for it.

Metadata

Collectively, objects defined within a database are known as its metadata or, more traditionally, its schema. The process of creating and modifying metadata is referred to as data definition. The term “data definition” is often also applied to the description of a single object and its attributes.

The system tables

Firebird stores metadata in a set of tables which it creates right inside the database—the system tables. All system tables have identifiers beginning with “RDB\$”. For example, the table which stores the definitions and other information about all of the table structures in your database is called RDB\$RELATIONS. A related table, RDB\$RELATION_FIELDS, stores information and definitions for the columns of each table.

This “database within a database” is highly normalized. DDL statements are designed to perform operations on the metadata tables safely and in full cognisance of the cascading effects.

It is possible to alter the data in the system tables by performing regular SQL operations on them. Some admin tools, such as *isql* and *gfx*, do internally change data in the system tables. However, as a sophisticated database management system, Firebird was not designed with raw end-user manipulation of the system tables in mind.

Any human intervention is likely to cause unpredictable types of damage. It cannot be stressed enough how important it is to treat DML on the system tables as a minefield. From Firebird 3 onward, the tables will be read-only.

SELECT queries on the system tables are fine and can be very useful for listing out things like character sets, dependencies and so on. For full schema of the system tables, refer to Appendix V.

Firebird's SQL Language

An SQL statement is used to submit a query to the database. The language of the query is expressed in statements that specify purpose: what is to be done (an operation), the objects to which it is to be done and the details of how it is to be done.. In theory, every possible interaction between the outside world and a database is surfaced through an SQL statement of some kind.

Statement syntaxes are grouped according to two broad purposes:

- 1 Those that CREATE, ALTER or DROP metadata objects (also known as schema objects or schema elements). Such queries are referred to as data definition language, or DDL.
- 2 Those that operate on data. They provide language for defining sets of data in columns and rows and specifying operations to
 - retrieve and transform (SELECT) images of those sets from storage for reading by applications
 - change the state of the database by adding, modifying or deleting the specified sets (INSERT, UPDATE and DELETE operations).

These statements that operate on data are referred to as data manipulation language, or DML.

The Firebird implementation of the SQL language falls into a number of overlapping subsets, each used for a specific purpose and incorporating its own language elements:

Dynamic SQL (DSQL)—the subset in most common use today, it is used by all database interface layers that communicate with the server through the Application Programming Interface (API). It comprises all DDL and DML statements and their syntaxes.

Procedural SQL (PSQL)—consists of all DML statements, with the addition of a number of procedural extensions.

Embedded SQL (ESQL)—the “base” SQL implementation, consisting of DDL and DML syntax that the other subsets incorporate, where possible. It was the original implementation of SQL in the InterBase predecessors, designed to be embedded in client applications and pre-compiled. It is rarely used today and we do not cover it in this book.

Interactive (ISQL)—the language implemented for the command-line *isql* utility. It is based on DSQL with extensions for viewing metadata and some system statistics and to control *isql* sessions.

SQL “Dialects”

The concept of “dialect” arose in Firebird’s long-ago predecessor, InterBase 6, as a transition feature allowing the database engine to recognise, accept and correctly process the features and elements of legacy InterBase 5 and older databases (Dialect 1), to access these older data for conversion to the new features and elements (Dialect 2) or to apply the full Firebird set of features, elements and rules to converted or newly created databases (Dialect 3).

Firebird creates a database in Dialect 3 by default. It is actually possible to create a new database in Firebird as Dialect 1 but it is strongly recommended to avoid doing so. Dialect 1, to which newer language features are unavailable, it will be deprecated eventually.



It is not possible to create a Dialect 2 database, since Dialect 2 is a client attribute intended for converting Dialect 1 databases to Dialect 3. The implementation of Dialect 2 is fairly imperfect and it will never be improved.

Firebird and the ISO Standards

SQL (pronounced “ess cue ell”) is a data sub-language for accessing relational database management systems. Its elements, syntax and behavior became standardized under ANSI and ISO in 1986.

The standard SQL query language is non-procedural; that is, it is oriented toward the results of operations rather than toward the means of getting the results. It is intended to be a generic sub-language, to be used alongside a host language. The purpose of standards is to prescribe a surface-level mechanism whereby a specified query request returns a predictable result, regardless of how database engines manipulate input internally.

Since its introduction, the SQL standard has undergone three major reviews: SQL-89 (published in 1989), SQL-92 (1992 or thereabouts) and more recently “SQL 3”, an ongoing work published in part as “SQL-99”. While the standards prescribe in great detail how the elements of the language shall be expressed and how the logic of operations is to be interpreted, it makes no rules about how a vendor should implement it.

Conformance with the standard is a matter of degree, rather than an absolute. Vendors may freely implement language features that the standard does not prescribe. Conformance is about the ways that features recognised and described by the standards are implemented, if they are present. Conformance varies inversely with the number of standard features that are implemented in an idiosyncratic fashion.

A “conforming” implementation is expected to provide the basic set of features and may include other features, grouped and standardized at levels above “entry level”. Firebird’s SQL language adheres closely to the SQL-92 and SQL-3 standards at entry level.

Data Definition Language (DDL)

The underlying structures of a database—its tables, views, and indexes—are created using a subset of the Firebird SQL language known as Data Definition Language (DDL). A DDL statement begins with one of the keywords CREATE, ALTER, RECREATE or DROP, causing a single object to be created, modified, reconstructed or destroyed, respectively. The database and, thereafter, its objects, rules and relationships interlock to form the structure of a relational database.

The next six chapters cover the creation and maintenance of databases and the objects in them, using DDL.

Data Manipulation Language (DML)

Data manipulation language (DML) comprises the syntaxes for statements and expressions that store, modify and retrieve data. DML statements begin with a verb—either INSERT, UPDATE, DELETE, EXECUTE or SELECT. Along the way are functions and operations that can group, sort and limit output sets.

Dynamic vs Static SQL

SQL statements embedded and precompiled in code are sometimes referred to as static SQL. By contrast, statements that are generated by a client program and submitted through the API to the server for execution during run-time are known as dynamic SQL (DSQL).

Unless you are writing code for ESQL applications, you are using DSQL. Statements executed by the interactive SQL utility (isql), or other interactive desktop utility programs are DSQL, as are those processed through client applications that use the API directly or indirectly (through database access drivers like ODBC, JDBC and the BDE).

In ESQL applications, static SQL allows queries to bypass the Firebird API, instead being pre-compiled to use macro calls to the API structures. ESQL is rarely used these days and is not covered in this book.

Chapters 18 to 23 cover the many aspects of DML.

Procedural language (PSQL)

The standard does not prescribe procedural language features since, in principle, it assumes that general programming tasks will be accomplished using the host language. There is no specification for language constructs to manipulate, calculate or create data programmatically inside the database management system.

Those RDBMS engines which support server-based programming usually provide SQL-like statement formats and syntaxes to extend SQL. Each vendor's implementation freely provides its own variants of these constructs. Typically, such code modules in the database are called **stored procedures**.

Firebird provides them as procedure language (sometimes referred to as PSQL), a set of SQL extensions which programmers use, along with a variant of the ESQL language set, to write the source code for stored procedures and triggers. PSQL is extended to include flow control, conditional expressions, and error handling. It has the unique ability to generate multi-row output sets that can be directly accessed using SELECT statements.

Certain SQL constructs, including all DDL statements, are excluded from PSQL workflow. However, the EXECUTE STATEMENT syntax of PSQL enables the execution of DSQL commands, including some DDL for those willing to take the risk.

PSQL for stored procedures and triggers is described in detail in Part Six, *Programming on the Server*.

Interactive SQL (ISQL)

The interactive query tool *isql* uses DSQL statements, along with a two subsets of extension commands (the SET XXX and SHOW XXX groups) which allow certain settings and schema queries, respectively, to be performed interactively. Certain SET commands can also be included in data definition scripts for batch execution in *isql*.

The *isql* language set also includes other specific commands for use in scripting, such as INPUT, OUTPUT, EDIT and more.

The *isql* utility, including its command set and scripting, is fully covered in Chapter 24, *Interactive SQL Utility (isql)*.

Schemas and Scripts

It is very good practice to use DDL scripts to create your database and its objects. Some of the reasons include

- Self-documentation. A script is a plain text file, easily handled in any development system, both for updating and reference. Scripts can—and should—include detailed comment text. Alterations to metadata can be signed and dated manually.
- Control of database development and change. Scripting all database definitions allows schema creation to be integrated closely with design tasks and code review cycles.
- Repeatable and traceable creation of metadata. A completely reconstructable schema is a requirement in the quality assurance and disaster recovery systems of many organizations.
- Orderly construction and reconstruction of database metadata. Experienced Firebird programmers often create a set of DDL scripts, designed to run and commit in a specific order, to make debugging easy and ensure that objects will exist when later, dependent objects refer to them.

Using isql to run scripts

A large section of Chapter 24 describes how you can use *isql* to create and run scripts of DDL statements and more, both interactively and as a “job” using command-line (console) mode. Refer to the topic *Creating and Running Scripts*.



Scripts created for use with isql can be used with many of the more sophisticated third-party administration software products available out there in the Firebird tools market.

Resources

For a list of resources to help you on your Firebird journey, refer to Appendix XIV, *Resources*.

DATA DEFINITION LANGUAGE—DDL

When defining metadata for use in a Firebird database, we use a lexicon of standard SQL statements and parameters that provide for the creation of an object by its type and name—or identifier—and for specifying and modifying its attributes. Also in this lexicon, known as data definition language (DDL), are statements for removing objects.

Queries using DDL are reserved for the purpose of metadata definition—so

- control them carefully if implementing them in end-user applications
- expect compiler exceptions if you attempt to use them directly in stored procedures or to pass their names as input parameters

Firebird's DDL is described in the next five chapters. View definitions, other run-time predefined set structures and the granting and revoking of SQL permissions are also DDL. Views, which incorporate both DDL and DML, are discussed in Chapter 23, *Views and Other Run-time Set Objects*. Defining and manipulating SQL permissions is described in Chapter 37, *Database-Level Security*.

SQL Data Definition Statements

Following is a round-up of the verbs that comprise the DDL lexicon.

CREATE

`CREATE <object>`

All objects you create in a database come into being by way of a CREATE statement, including the database itself, of course.

The object's owner

Every object has an owner. The owner is the user who creates the object, regardless of which user owns the database. Whilst a database's owner can change (by being restored

from a backup by a different user), all of the objects within the database retain their original owner, even after a restore.

Many objects require you to be logged in as the owner of an object in order to create associate objects, e.g., indexes or triggers for a table. You also need to be the owner of an object to change its metadata or to delete it.

SYSDBA and other privileged users

A user logged in as SYSDBA (or another with SYSDBA-like privileges in the database) can perform the owner’s tasks without incurring exceptions. It is important to keep in mind that “ownership sticks”. If SYSDBA creates an index for a table owned by another user, SYSDBA is the owner of the index but not of the table. This is not a desirable situation.

RECREATE

RECREATE <object>

The RECREATE verb is available for some types of objects but not others. The implementation of RECREATE syntax for the different object types has been a progressive process across versions—whether it is available depends on server version.

RECREATE is exactly like its CREATE counterpart except that, if the object already exists, the old object will be dropped (removed) before the new one is created from scratch.

If there is an existing object of the same type and name,

- RECREATE will fail if it has other objects dependent on it. For example, you could not RECREATE a table if another table has a FOREIGN KEY or a CHECK constraint defined that references it.
- All existing related objects and data will be lost.

ALTER

also CREATE OR ALTER

ALTER <object>

CREATE OR ALTER <object>

ALTER and its near relative, CREATE OR ALTER, are the less “aggressive” than RECREATE. The ALTER syntax will make the metadata changes without requiring the dissociation of dependent objects. Data, indexes and triggers for tables are preserved.

CREATE OR ALTER works exactly like ALTER if an existing object of the same name or type exists; otherwise, it works exactly like CREATE.

In some cases, ALTER syntax is provided for objects whose definition you cannot actually alter. For example, ALTER INDEX can only activate or deactivate the index, which sets a flag on its metadata record in the system table RDB\$INDICES to ACTIVE | INACTIVE. A similar trick is at work with ALTER TRIGGER, which can be used with nothing but the ACTIVE | INACTIVE parameter to enable and disable the trigger.

DECLARE

Some object “types” are actually declarations of external pieces that are for optional usage in databases. External functions (UDFs) and BLOB filters are declared in a database, not created, because they live in separate shared libraries, in locations known to the engine by way of parameters in `firebird.conf`. If they are not declared, they are not available.

Example

```
DECLARE EXTERNAL FUNCTION getExactTimestampUTC
TIMESTAMP
RETURNS PARAMETER 1
ENTRY_POINT 'getExactTimestampUTC' MODULE_NAME 'fbudf';
```

Other, unrelated parts of the wider lexicon use the `DECLARE` verb. For example, in procedural SQL (PSQL), the same verb is used for declaring variables and cursors.

DROP

`DROP <object>`

`DROP` is the universal verb for killing objects permanently. It is the easiest statement to type in SQL—it takes no arguments other than the name of the object you want to remove.



Dropped objects are gone for ever!

Storing Descriptive Text

Firebird does not come with a formal data dictionary tool but the metadata of every metadata object, including the database itself, has a `TEXT` BLOB column named `RDB$DESCRIPTION`. Any amount of descriptive text can be stored there.

Objects with `RDB$DESCRIPTION` columns

The object types that have `RDB$DESCRIPTION` columns are

The database itself (in <code>RDB\$DATABASE</code>)	Tables	Views
Tables	Views	Domains
Indexes	Triggers	Stored Procedures
Character sets	Collations	Exceptions
Generators (Sequences)	External functions (UDFs)	Roles
BLOB Filters		

Many graphical administration tools enable viewing and editing of this description text. The open source *FlameRobin* displays them with table properties and provides an editing interface. Mario Zimmerman developed a free tool, called *IBDesc*, that produces reports containing the information stored in `RDB$DESCRIPTION`.

Without dedicated tools, this column can be queried interactively by applications for any of the object types that have this column.



It is in the development plans for Firebird to make the system tables read-only in future versions. If you intend developing your own tools for maintaining a data dictionary for your databases, keep in mind that writing and updating these BLOBs by direct DML has a limited life expectancy.

The COMMENT statement

From the “2” series onward, you can populate the RDB\$DESCRIPTION column using the explicit DSQL verb COMMENT. It takes as its argument a string expression that will be converted to a BLOB and stored persistently.

While implementing the COMMENT verb is a step in the right direction towards including provision for documenting databases without recourse to performing DML on the system tables, it is a small step. BLOBs cannot be updated: if you need to “change” the content of the descriptive text, you have to start over. Once the system tables become read-only, application tools that currently provide a direct BLOB editing interface will not work, unless the Firebird developers implement some special exclusion, perhaps by providing a system view to enable privileged users write access to RDB\$DESCRIPTION. The 32Kb limit on string size (bytes, not characters) is a limitation, given that metadata text is stored in the 3-byte UNICODE_FSS character set.

```
Syntax      COMMENT ON <object> IS {'Supply text here' | NULL}
            <object> ::= DATABASE | <basic-type> objectname |
            COLUMN relationname.fieldname | PARAMETER procname.paramname
            <basic-type> ::= CHARACTER SET | COLLATION | DOMAIN | EXCEPTION |
            EXTERNAL FUNCTION | FILTER | GENERATOR | INDEX |
            PROCEDURE | ROLE | SEQUENCE | TABLE | TRIGGER | VIEW
```

The text

The text can be anything you want, as long as it does not exceed 32,767 bytes. It will be transliterated to UNICODE_FSS for storage. For retrieval, it will be transliterated to the character set of the client.

NULL is the default content when a row is created in the relevant system table. If you supply an empty string (' ') it will be stored as NULL.

```
Examples    COMMENT ON DATABASE IS 'I am valuable: please back me up regularly'
            COMMENT ON TABLE EMPLOYEE IS 'If you work here we know who you are.'
            COMMENT ON COLUMN EMPLOYEE.FULL_NAME IS
            'Read-only: concatenates LAST_NAME and FIRST_NAME'
```

Comments in scripts

Writing DDL for a database in a script for input to the *isql* utility is highly recommended. The subject is covered in Chapter 24, *Interactive SQL Utility (isql)*.

In scripts—and anywhere you store SQL statements—you have two ways to include comments.

- the C-style comment marker-pair `/* comment */` can be used to enclose one or more lines of comment, as in

```
/* This is a line of useful information:
   Make sure you read it! */
```

This method can also be used for in-line comments, such as “commenting out” code that you don’t want to delete, as in

```
ALTER TABLE VEGETABLES
```

```
ADD SEASON VARCHAR(10) /* , SUGAR_CONTENT DOUBLE PRECISION, */
```

- the one-line comment marker, consisting of two hyphens '--' at the beginning of a line. Everything following that marker will be ignored. For example,
-- This is one line of useful information

Object Dependencies

In a relational database management system like Firebird, many kinds of objects have dependencies on other objects. For example, when a FOREIGN KEY constraint references a PRIMARY KEY or UNIQUE constraint in another table, the referenced table acquires the referencing table as a dependency.

Dependencies occur when you place a CHECK constraint on a table, column or domain that refer to another object in the database. The object referred to by the CHECK constraint acquires that constraint as a dependency.

It is also possible—although a very bad idea unless it is a self-referencing relationship protected by referential integrity—to cause dependencies between rows within the same table. An example would be a CHECK constraint that validates an incoming value by checking a value in another row in the same table.

Generators (a.k.a. sequences) live an independent life. When created, they owe nothing to any other database object. It is not unusual for the same generator to serve multiple purposes in a database.

However, generators (sequences) are widely used in triggers to automate the creation of keys for tables. (For details of that technique, refer to *Implementing Auto-Incrementing Keys* in Chapter 30.) Thus, a generator is usually a dependency for a trigger somewhere.

The engine is very thorough about protecting dependencies. You will get exceptions if you try to change or delete any object that another object depends on. When you must do that change or deletion, you must always remove the dependencies first—drop a constraint that depends on an unwanted object, for example.

Using DDL to Manage User Accounts

Firebird 2.5 introduced DDL syntax to enable user accounts on the server to be managed by submitting SQL statements when logged in to a regular database.

CREATE/ALTER/DROP USER

The CREATE USER, ALTER USER and DROP USER statements can be used by the SYSDBA or another privileged user as a direct alternative to using the gsec utility for managing the global user table in the security database. CREATE USER and ALTER USER also include the optional parameters GRANT ADMIN ROLE and REVOKE ADMIN ROLE to enable a privileged user to grant the RDB\$ADMIN role in the security database to an ordinary user.

For the full overview of the RDB\$ADMIN role, refer to Chapter 36, *Protecting the Server*.

Syntax Patterns

The SYSDBA, or a user with SYSDBA privileges in both the current database and the security database, can add a new user:

```
CREATE USER <username> {PASSWORD 'password'} [FIRSTNAME 'firstname']
[MIDDLENAME 'middlename'] [LASTNAME 'lastname'] [GRANT ADMIN ROLE];
```



The PASSWORD clause is required when creating a new user. It should be the initial password for that new user. The user can change it later, using ALTER USER.

The SYSDBA, or a user with SYSDBA privileges in both the current database and the security database, can change one or more of the password and proper name attributes of an existing user. Non-privileged users can use this statement to alter only their own attributes.

```
ALTER USER <username> [PASSWORD 'password'] [FIRSTNAME 'firstname']
[MIDDLENAME 'middlename'] [LASTNAME 'lastname'] [{GRANT | REVOKE} ADMIN ROLE];
```



At least one of PASSWORD, FIRSTNAME, MIDDLENAME or LASTNAME must be present.

ALTER USER does not enable the changing of user names. If a different user name is required, the old one should be dropped and a new one created.

The GRANT/REVOKE ADMIN ROLE arguments

GRANT ADMIN ROLE and REVOKE ADMIN ROLE are optional arguments to the CREATE USER and ALTER USER statements. These arguments provide a way to elevate the privileges of an ordinary user.

See also ALTER ROLE, discussed in detail in Chapters 36, *Managing Users*, and 37, *Database-level Security*.

The SYSDBA, or a user with SYSDBA privileges in both the current database and the security database, can delete a user:

```
DROP USER <username>;
```

Restrictions

CREATE USER and DROP USER statements and the arguments GRANT | REVOKE ADMIN ROLE are available only for the SYSDBA, or a user that has acquired the RDB\$ADMIN role in both the current database and the security database.

An ordinary user can ALTER his own password and elements of his proper name. An attempt to modify another user will fail.

Examples

SYSDBA, or a user with equivalent privileges in both the current database and the security database, can do:

```
CREATE USER fluffy PASSWORD 'test';
..
ALTER USER fluffy FIRSTNAME 'Foufou' LASTNAME 'Curlychops';
ALTER USER fluffy PASSWORD 'MePoodle';
```

The following statement escalates the privileges of user 'FLUFFY' globally, i.e., gives it SYSDBA-equivalent rights over all objects in all databases:

```
ALTER USER fluffy GRANT ADMIN ROLE;
```

To drop user FLUFFY:

```
DROP USER fluffy;
```


Reference Material

When the first edition of this book was published in 2004, the Firebird Project had virtually no reference material of its own for SQL. Users relied heavily on two volumes of the beta documentation that had been written for the InterBase 6 code that was subsequently released as open source: the Data Definition Guide and the Language Reference, along with the detailed release notes produced for each release and sub-release of Firebird.

The Firebird Project has been inhibited in producing its own updated documentation by the fact that the proprietors of InterBase have never permitted the content of that beta documentation to be re-used without threat of legal action. However, copies of the original PDF books are available in numerous locations on the Web and links can be found at the Firebird website, in the Documentation section.

Since 2004, the coordinator of Firebird’s documentation project, Paul Vinkenoog, has dedicated hundreds of voluntary hours and much talent to producing the “Firebird Language Reference Update” books for each major Firebird version, starting with v.1.5. They merge both the Data Definition and the Language Reference changes in a cumulative series. They are mind-bogglingly good in their detail.

The latest, at the time of this writing, is the **Firebird 2.5 Language Reference Update**, comprehending all of the changes in DDL, DML and PSQL since the InterBase 6 beta was forked to become Firebird at the end of July, 2000. It is authoritative and is updated frequently, always after the publication of release notes—which means its content at any point can be counted more reliable than the release notes it was sourced from.

Since 2010, the language reference update PDFs have been distributed in the binary kits and scripted to install in the /doc/ subdirectory beneath the Firebird root.

If you need to download any of these volumes, you can find the links at the Firebird website:

Firebird Language Reference Updates: <http://www.firebirdsql.org/en/reference-manuals/>, scroll down to “Reference Material”

Latest Release Notes for all versions: <http://www.firebirdsql.org/en/release-notes/>

InterBase 6 Beta Manuals, including Data Definition Guide and Language Reference: <http://www.firebirdsql.org/en/reference-manuals/>, scroll down to “InterBase 6.0 Manuals”

CREATING AND MAINTAINING A DATABASE

A Firebird database is, first and foremost, a filesystem file under the control of the I/O subsystem of the host machine on which the Firebird server runs. Once the server has created this file, its management system takes control of managing the space, using a low-level protocol to communicate with the I/O subsystem.

Because of this protocol, a Firebird database must exist on the same physical machine as the Firebird server. It cannot be located on any sort of storage device whose physical I/O system is not directly controlled by server's host.

A new, “empty” database occupies about 540-600 Kb of disk. The database file is not empty at all, in fact, because the act of creation—the `CREATE DATABASE` statement—causes more than 40 system tables to be created. These tables will store every detail of the metadata as database objects are added and altered. Because the system tables are regular Firebird database objects, they already contain the metadata records for themselves. The server has already allocated database pages from disk for these data and has set up inventory pages for various types of object—tables, indexes, BLOBs—as well as for transactions and for the pages themselves.

Amongst the tables created in versions 2.1 and above are the monitoring tables, the suite of tables designed to be populated with the transient information about the running database, whenever requested by a client.

The schemata of the system and monitoring tables can be found in Appendix V.

Physical Storage for a Database

Before creating the database, you should know where you are going to create it! This is not as silly as it sounds. The `CREATE DATABASE`—alias `CREATE SCHEMA`—statement will create the file or files you name, but it cannot create

directories and it cannot change filesystem permissions. These details must be attended to first.

Additionally, a Firebird server may be configured to restrict the locations where databases may be accessed. Check the **DatabaseAccess** parameter in the `firebird.conf` file to discover whether your server is restricting access. If the setting is the default *Full* then you can create the database anywhere. Otherwise:

Restrict will indicate the filesystem tree-roots under which database access is permitted. Ensure that the user that starts your server has sufficient permissions to create a file there (or, in the case of the Windows embedded server, the user under which you are logged in).

Note *In Firebird 2.5 and above, if **DatabaseAccess** is **Restrict** then new databases, by default, are created in the first location listed in the arguments for that parameter. Supplying a path or alias in the **CREATE DATABASE** statement overrides it.*

None permits the server to attach only databases that are listed in `aliases.conf`. You can create a database anywhere but, except at creation, no client will be able to attach to it unless its alias and its absolute path are present in `aliases.conf`.

None is the default setting when Firebird is installed. It is strongly recommend that you retain this option and make use of the database-aliasing feature. For more information, refer to the topic Database aliasing in Chapter 4.

About Security Access

It is not always obvious to newcomers that there is a distinction between server access and database security. When you “log in” to a Firebird database using *isql* or your favorite admin tool you always supply a user name and password, along with server, port (sometimes) and path parameters. Whenever you do this, you are logging in to the *server* and opening an attachment to a database.

If the database doesn't exist yet and you have started *isql* from the command line with no parameters, then two things are “givens”:

- 1 you are logged in to the server
- 2 until you submit a **CONNECT** or **CREATE DATABASE** request to the server, the program is not attached to a database

Password access *in some form* is always required to log in to the server unless the user has root or Administrator privileges from the operating system. Once in, you can attach to any database. What you can do, once attached, depends on SQL privileges, which are stored within the database.

The SYSDBA user has full destructive rights to every database and every object within it. The owner—the user that created the database—has automatic rights to the database, although not to any objects within it that were created by other users. SQL privileges are “opt-in”. That means that, although any user with server access can attach to any database, it will have no rights to do anything to anything, other than what has been explicitly or implicitly granted to it as access privileges, using **GRANT** statements.

For detailed information about server access, see Chapter 36, Protecting the Server and its Environment. SQL privileges are discussed in Chapter 37, Database-Level Security.

ISC_USER and ISC_PASSWORD

It is possible to set up the two environment variables `ISC_USER` and `ISC_PASSWORD` on the server, to avoid the need to log in when working with databases locally. You will be able to do everything that the named user is allowed to do, without needing to supply credentials each time. This feature is handy for administrative tasks but it must be used with a high level of caution because it leaves your database access open to any local user who happens upon your command shell.

If you want to play with fire, set these two variables permanently. If you want to have that extra level of convenience and script security, set them temporarily each time you want them and be certain to reset them whenever you leave your console:

On Linux, in the same shell from which you will launch the application:

```
]# setenv ISC_USER=SYSDBA
]# setenv ISC_PASSWORD=masterkey
```

To unset, either

```
]# setenv ISC_USER=
]# setenv ISC_PASSWORD=
```

or simply close the shell.

On Windows, go to the command prompt and type:

```
set ISC_USER=SYSDBA
set ISC_PASSWORD=masterkey
```

To unset:

```
set ISC_USER=
set ISC_PASSWORD=
```

Creating a Database

You can create a database interactively in `isql`. Some other database administration tools can meet the API requirements (see below) and let you create databases interactively, while others require a script.

In any case, it is preferable to use a data definition file (DDL script) because it provides an easy way to “replay” your statements if a statement fails—it is easier to start over from a corrected source file than to retype interactive SQL statements.

Refer to Chapter 24, *Interactive SQL Utility (isql)* for usage instructions.

Dialect

default dialect 3

Firebird creates a dialect 3 database by default. You should retain the default unless you have a genuine reason to create the database in dialect 1.

If you wish to create a dialect 1 database, the first statement in the script (or the prior action in your admin tool) should be

```
SET SQL DIALECT 1;
```



If isql is currently attached to a database, it will prompt you to commit the current transaction. Answer Yes to proceed with creating the new database. Some third-party tools may require that you disconnect from any existing attachment first.

CREATE DATABASE Statement

The next statement—or the first, for a dialect 3 database—must be the CREATE DATABASE or CREATE SCHEMA statement, using the prescribed syntax. First, we consider the syntax for a single-file database. We look at the extra syntax for a multi-file database a little later.

The database created will have the on-disk structure (ODS) associated with the version of the Firebird server that is running. For information about the ODS, refer to the topic *Version Lineage* in Chapter 5, **Migration Notes**.

Syntax

```
CREATE {DATABASE | SCHEMA} 'file-specification'
    [USER 'username' [PASSWORD 'password']]
    [PAGE_SIZE [=] int]
    [LENGTH [=] int [PAGE[S]]]
    [DEFAULT CHARACTER SET character-set [COLLATION collation]]
    [DIFFERENCE FILE 'difference-file-path']
```



Use single quotes to delimit strings such as file names, user names, and passwords.

'DATABASE' or 'SCHEMA'?

CREATE DATABASE and CREATE SCHEMA are the same statement. It is merely a question of preference which you use.

Mandatory and optional parameters

The only mandatory parameter for the CREATE statement is the file-specification—the name of the primary database file and the filesystem path to its location, or its alias.

Database path and name

The file-specification must be one of the following:

- the fully-qualified, absolute path to the file, in a valid format for the operating system platform, for example:

POSIX

```
CREATE DATABASE '/opt/databases/mydatabase.fdb'
```

Win32

```
CREATE SCHEMA 'd:\databases\mydatabase.fdb'
```

You can use either forward slashes (/) or backslashes (\) as directory separators. Firebird automatically converts either type of slash to the appropriate type for the server operating system.

The enclosing single-quotes for file_specification are not optional. All elements of the file specification are case-sensitive on POSIX platforms.

- an alias that has already been registered in aliases.conf, that complies with the rules above and points to a valid location

- under Firebird 2.5 or higher, a file name alone, provided the **DatabaseAccess** parameter in `firebird.conf` is configured to *Restrict* and designates at least one valid directory path. This option does not work if you are creating the database remotely.

If you use an unqualified file name in lower versions, the database file will be created in the current working directory if **DatabaseAccess** is configured to *Full*; otherwise, an exception is thrown.

If creating the database remotely, i.e., from a client workstation, or locally on Linux Superserver, either interactively or using a script, you must include the host name. For example:

POSIX Classic or Superclassic

```
CREATE DATABASE 'myserver:/opt/databases/mydatabase.fdb'
```

POSIX SS local, as above, or

```
CREATE DATABASE 'localhost:/opt/databases/mydatabase.fdb'
```

Windows, TCP/IP protocol:

```
CREATE SCHEMA 'WinServer:d:\databases\mydatabase.fdb'
```

Windows, Named Pipes (NetBEUI) protocol:

```
CREATE SCHEMA '\\WinServer\d:\databases\mydatabase.fdb'
```

Ownership

If you are logged in as SYSDBA then SYSDBA will own the new database unless you include the clause specifying the USER and PASSWORD. Although it is optional to designate an owner, it is highly desirable to do so. However, for security reasons, you will probably wish to remove the user's password from the script before archiving it with other system documentation.

```
CREATE DATABASE '/opt/databases/mydatabase.fdb'
USER 'ADMINUSR' PASSWORD 'yyuryyub';
```

Page size

The option `PAGE_SIZE` attribute is expressed in bytes. If you omit it, it will default to 4096 bytes with *isql*. Some other tools apply their own defaults, so there is a strong argument for specifying it explicitly in the script. The page size can be 4096, 8192 or 16384. Any other numbers will be resolved back to the next lowest number in this range. For example, if you specify 5000, Firebird will create a database with a page size of 4096.

If the size specified is smaller than 4096, it will be silently converted to 4096.

Example

```
CREATE DATABASE '/opt/databases/mydatabase.fdb'
USER 'ADMINUSR' PASSWORD 'yyuryyub'
PAGE_SIZE 8192
...
```

V.I.X 1024 and 2048 are valid page sizes under the v.1.0.x servers and 2048 is valid under v.1.5.x.

Factors influencing choice of page size

Choosing a page size is not a question of applying some rule. It will do no harm to begin with the default size of 4 Kb. When the time comes to tune the database for performance improvements, you can experiment by backing up the database and restoring it with different page sizes. For instructions, refer to the section *The gbak Utility* in Chapter 39.

The page size you choose can benefit performance or affect it adversely, according to a number of factors having mostly to do with the structures and usage of the most frequently accessed tables. Each database page will be filled to about 80 per cent of its capacity, so think in terms of an actual page size that is around 125 percent of the size you determine to be the minimum.

The row size of the most-frequently accessed tables may have an effect. A record structure that is too large to fit on a single page requires more than one page fetch to read or write to it, so access can be optimized by choosing a page size that can comfortably accommodate one row or simple row multiples of these high-volume tables.

The number of rows that your main tables can be predicted to accommodate over time may have an influence. If multiple rows can be accommodated in a single page, a larger page size may reduce the overall tally of data and index pages that need to be read for an operation.

Default character set

Strongly recommended unless all—or nearly all—of your text data will be in US ASCII. From v.2.5 onward, you can also specify the default COLLATION sequence for character columns that use the default character set.



If you are not sure about the availability of the character set and/or collation that you want to be the default, check the manifest file `fbintl.conf` in the `/intl/` sub-directory of your Firebird 2.1 or higher installation. (COLLATION is not available under v.2.1.)

```
CREATE DATABASE 'opt/databases/mydatabase.fdb'
  USER 'ADMINUSR' PASSWORD 'yyryyub'
  PAGE_SIZE 8192
  DEFAULT CHARACTER SET ISO8859_1
  COLLATION FR_CA;
```

For details regarding character sets refer to the topic [*Character Sets and Collation Sequences*](#) in Chapter 9, **Character Types**.

Difference file

If you are planning to use the *nBackup* incremental backup tool, you have the option of specifying the full path to the file that will be used to store “delta” files whilst *nBackup* has the database in a LOCKED (i.e., read-only) state for a copy operation. For more information, see the topic [*Incremental Backup Tool \(nBackup\)*](#) in Chapter 39.

Getting information about the database

Once you have created and committed the database, you can display its details in *isql* using the SHOW DATABASE command:

```
SQL> SHOW DATABASE;
Database: /opt/databases/mydatabase.fdb
      Owner: ADMINUSR
PAGE_SIZE 8192
Number of DB pages allocated = 176
Sweep interval = 20000
Forced Writes are ON
Transaction - oldest = 5
```



```

Transaction - oldest active = 6
Transaction - oldest snapshot = 6
Transaction - Next = 9
Default character set: ISO8859_1
Default collation: FR_CA
ODS = 11.2

```

Database Attributes

The list of database attributes in the output comes from the database header page of the database. You can get almost exactly the same information from calling `gstat -b`. Some of those attributes have come from the CREATE DATABASE specification; others are installed as defaults at creation time.

Sweep interval and transactions

For information about sweeping and sweep interval refer to the topic [*Keeping a Clean Database*](#) later in this chapter.

The values of the oldest (“oldest interesting”), oldest active and next transactions are important for performance and server behavior. For details, refer to the chapters in Part Five, **Transactions**.

Forced writes

Forced Writes is synonymous with synchronous writes. On platforms that support asynchronous writes, Firebird databases are created with Forced Writes enabled by default. The term “disabling Forced Writes” means switching the write mode from synchronous to asynchronous, an action that should only be done by a privileged user with exclusive access.

- With forced (synchronous) writes enabled, new records, new record versions and deletions are physically written to disk immediately upon posting or, at the latest, upon committing..
- Asynchronous writes cause new and changed data to be withheld in the filesystem cache, relying on the flushing behavior of the operating system to make them permanent on disk.

For discussion of the implications of disabling Forced Writes and instructions for setting it using *gfix*, see the topic [*Forced Writes*](#) in Chapter 35, **Configuring Databases**.



The Windows 95 platform does not support asynchronous writes. The same applies to Linux for the following versions of Firebird, due to a previously undiscovered bug in the Linux kernel:

- all 1.X versions
- all 2.0.x versions prior to v.2.0.6
- all 2.1.x versions prior to v.2.1.4

Single and Multi-file Databases

Any Firebird database can be multi-file. You don't have to decide between single and multiple at the beginning. A single file database can be converted to multi-file at any time, using ALTER DATABASE (see below) or the *gbak* tool.

Specifying file size for a single-file database

You can optionally specify a file length, in pages, for the primary file, following the `PAGE_SIZE` attribute. For example, the following statement creates a database that is stored in one 10,000-page-long file:

```
CREATE DATABASE '/opt/databases/mydatabase.fdb'
  USER 'ADMINUSR' PASSWORD 'yyuryyub'
  PAGE_SIZE 8192
  LENGTH 10000 PAGES /* the PAGES keyword is optional */
  DEFAULT CHARACTER SET ISO8859_1
  COLLATION FR_CA;
```

If the database grows larger than the specified file length, Firebird extends the primary file beyond the `LENGTH` limit until the filesystem size limit for shared access file is reached or disk space runs out. To avoid this, you can store a database in more than one file, called secondary files. The files can be on separate disks.

Creating a Multi-file Database

Multi-file databases are more of an issue on older filesystems where the absolute limit for a shared-write file is 2 Gb (FAT32, ext2) or 4 Gb (NTFS systems with 32-bit I/O). A Firebird database cannot be corrupted by “blowing the limit” of its primary or secondary file’s capacity—it simply denies all writes when the last file hits the limit. Any outstanding writes will be lost.

In the following example, a database is created consisting of three files, each potentially of 2 Gb. If the filesystem supports a larger shared-access file, the last file will continue to grow until the filesystem limit (if any) is reached.

```
CREATE DATABASE 'localhost:/data/sample.fdb'
  PAGE_SIZE 8192
  DEFAULT CHARACTER SET ISO8859_1
  LENGTH 250000 PAGES
  FILE '/data/sample.fdb1'
  FILE '/data/sample.fdb2'
  STARTING AT 250001;
```

You must specify a range of pages for each file either by providing the *number of pages* in each file, or by providing the *starting page number* for the file. For the last file, you don’t need a length, because Firebird always dynamically sizes the last file and will increase the file size as necessary until all the available space is used or until it reaches the filesystem limit.

In the example, the first secondary file will “kick in” when the first primary file is nearing the 2 Gb limit. The next file in the chain comes into use when a requested operation is likely to need more pages allocated than the previous files could satisfy without exceeding their specified limits.

It is the responsibility of the database administrator to monitor database growth and ensure that a database always has ample capacity for extension. Deciding if and when to split a database file depends on how large you expect the database to grow and how quickly. More files can be added at any time using the `ALTER DATABASE` statement (below).

With multi-file databases you can avoid confining databases to the size of a single disk if the system does not support spanning a single, huge file across multiple disks. There will be no problems installing a RAID array and distributing a multi-file Firebird database across several disks on any supported platform.

Important *All files must be located on disks that are under the direct physical control of the Firebird server's host machine.*

The Database Cache

Database cache is a chunk of memory reserved for each database running on the server. Its purpose is to cache all of the database pages (sometimes referred to as “page buffers”) that have been most recently used. It is configured as a default for new databases and for all databases that are not individually configured. You can check this default setting by looking up the parameter **DefaultDbCachePages** in `firebird.conf`.

The value constitutes a number of blocks of memory, or “buffers”, each the size of one database page. It follows, then, that databases with large page sizes will use more cache memory than those with smaller page sizes.

V.1.0.x For v.1.0.x, check **database_cache_pages** in `isc_config` (POSIX) or **ibconfig** (Win32).

It should be emphasized that configuring the cache is not a “must-do”. The default configuration for Superserver fits most normal needs and server level reconfiguration might never be necessary. On Classic, the default is more worthy of attention, since it may be too high for a system with more than a few concurrent users.



Pay special attention to the cache sizes if you are likely to be swapping from Classic to Superserver, or vice versa.

Database-level setting

A newly created database has a database-level cache size of zero pages. If the cache setting is left at zero, connections to that database will use the **DefaultDbCachePages** setting from `firebird.conf`.

Cache size can be configured individually and permanently, per database. It can be changed again, if required. Other databases that retain (or are changed to) zero-cache will use the server default.

Sizing the cache

The number of cache buffers required is approximate—there is no one “thing” that dictates what size you should choose. It needs to be large enough to cater for the page requirements of databases but not so large as to consume memory that is needed for other operations. Up to a point, the more activity that can be handled in the cache, the better the overall performance. The axiom “Database servers love RAM” is true for Firebird. But Firebird uses RAM for other activities that are at least as important as caching. Transaction inventory and index bitmaps are maintained in RAM; sorting and merging are done in memory, if it is available, in all versions except v.1.0.x.

It is important to realize that every system has a critical point where a too-large cache configuration will consume more memory resources than the system can spare. Beyond this point, enlarging the cache will cause performance to degrade.

Limits and defaults

The minimum cache size is 50 pages. There is no maximum, as long as the allocation in total does not exceed the RAM available.

Default cache allocation is

- Superserver 2048 pages for each running database. All users share this common cache pool.

As an indication of how resources can be consumed, a single database running at the default settings for `PAGE_SIZE` (4Kb) and `DefaultDbCachePages` (2 Kb) requires 8Mb of memory. Two databases running with the same settings require 16Mb, and so on. Default cache usage is calculated by:

$$\text{PAGE_SIZE} * \text{DefaultDbCachePages} * \text{number of databases}$$

- Classic Server 75 cache pages per client attachment. Each attachment is allocated its own cache. The amount of memory required is the total of the cache requirements of all client attachments to each database. Cache usage is calculated by:

$$\text{PAGE_SIZE} * \text{DefaultDbCachePages} * \text{number of attachments}$$

Cache usage

When Firebird reads a page from the database from disk, it stores that page in the cache. Ordinarily, the default cache size is adequate. If your application includes joins of five or more tables, Firebird Superserver automatically increases the size of the cache. If your application is well localized, that is, it uses the same small part of the database repeatedly, you might want to consider increasing the cache size so that you never have to release one page from cache to make room for another.

Because the database cache is configured in pages, obviously a database with a larger `page_size` consumes more memory than one with a smaller `page_size`. When there are multiple databases running on the same server, it may be desirable to override the server-wide cache size at database level or, in some cases, at application level (details below).

An application that performs intensive indexed retrievals requires more buffers than one that performs mainly inserts.

Where many clients are accessing different tables, or different parts of a single table, the demand for cache memory is higher than where most clients are working with the same, or overlapping, sets of data.

Estimating the size requirement

Estimating the size of the cache is not a simple or precise science, especially if you have multiple databases that have to run concurrently. The likely server cache usage is driven by the database with the largest page size. Classic allocates a cache for each attachment, whereas Superserver pools cache for all attachments to a particular database. As a starting point, it will be useful to work with the numbers and needs for the database with the biggest page size. Actual usage conditions will determine whether any adjustments are needed.

It is not necessary to have a cache that will accommodate an entire database. Arrive at a reduction factor for each database by estimating the proportion that is likely to be accessed during normal usage. The estimation suggested here is just that—there is no “rule”. Assume when we talk about DB cache pages here, or “buffers”, we are talking about the size of the cache for a particular database but not necessarily the default server setting for new and unconfigured databases.

Reduction factor, *r*, should be a value between 0 and 1.

Size of database, in pages, (size) can be established as follows:

- For a single-file database, take the maximum file size allowed by the filesystem, minus 1 byte, and divide it by the page size.

On operating systems which support huge files, use the database file size instead of the maximum file size.

- For a multi-file database, take the STARTING AT value of the first secondary file and add the LENGTH values of all of the secondary files.

Let DB cache pages = number of cache pages (buffers) required by this database.

For each database, calculate

DB cache pages = $r * \text{size}$

Calculate and record this figure for each individual database.



Keep these records with other database documentation for use when you need to tune the cache for an individual database.

It can happen that too many cache buffers are allocated for available RAM to accommodate. With many databases running simultaneously, a request could demand more RAM than was available on the system. The cache would be swapped back and forth between RAM and disk, defeating the benefits of caching. Other applications (including the server) could be starved of memory if the cache were too large.

It is important, therefore, to ensure that adequate RAM is installed on the system to accommodate the database server's memory requirements. If database performance is important for your users, then avoid creating competition for resources by running other applications on the server.

Calculating RAM requirements for caching

To calculate the amount of RAM required for database caching on your server, take the PAGE_SIZE of each database and multiply it by the **DefaultDbCachePages** value. These results for all databases, when added together, will approximate the minimum RAM required for database caching.

If you are running Firebird on a 32-bit system, or 32-bit Firebird on a 64-bit system, keep in mind that the total RAM available to a 32-bit process is limited 2 GB. Cache is only one of several ways that Firebird uses RAM.

Setting cache size at database level

There are several ways to configure the cache size for a specified database. Changes do not take effect until the next time a *first connection* is made to Firebird Superserver or the next client connects to the Classic server.

Only the SYSDBA or a user with equivalent privileges in the database can change the cache size.

Use gfix

The recommended way to set a database-level override to **DefaultDbCachePages** is to use the *gfix* utility with the following switches:

```
gfix -buffers n database_name
```

where *n* is the required number of database pages. This approach permits fine tuning to reduce the risk of under-using memory or working with a cache that is too small. The override is written permanently to the database header page and will remain in place until the next time it is changed with *gfix*.

For more information about using *gfix*, see [The gfix Tool Set](#) in Chapter 35, **Configuring and Managing Databases**.

Use the *isql* command-line query tool

To increase the number of cache pages for the duration of one session of the command-line utility *isql*, logging in as a super user, you have two options:

Either:

- include the number of pages (*n*) as a switch when starting *isql*

```
isql -c n database_name
```

n is the number of cache pages to be used for the session and temporarily overrides any value set by the **DefaultDBCACHEPages** (server default) configuration or *gfx* (database default). It must be greater than 9.

or

- include CACHE *n* as an argument to the CONNECT statement once *isql* is running:

```
SQL> connect database_name CACHE n
```

The value *n* can be any positive integer number of database pages. For Superserver, if a database cache already exists because of another attachment to the database, the cache size is increased only if *n* is greater than current cache size.

Use the database parameter buffer (DPB)

In an application for use by a super user, the cache size can be set in a database parameter buffer (DPB) using either the *isc_dpb_num_buffers* or the *isc_dpb_set_page_buffers* parameter, according to your server's requirements.

isc_dpb_num_buffers sets the number of buffers (pages of cache memory) to be used for the current connection. It makes most sense in a Classic architecture, where each connection gets a static allocation of cache memory. In Superserver, it will set the number of buffers to be used for the specific database, if that database is not already open, but will not persist after the server closes the database.

isc_dpb_set_page_buffers is useful in both Classic and Superserver. It has the same effect as using *gfx* to perform a persistent override of **DefaultDbCachePages**, for the cited database.



Be cautious about providing end-user applications with the ability to modify the cache. Although any request to change the cache size will be ignored on all connection requests except the first, many versions of Firebird allow non-technical users to change settings in the database header via the DPB. It is likely to have unpredictable effects on performance, besides exposing databases to ill-advised, ad hoc manipulations.

The vulnerable versions are:

- v.2.1 and 2.1.1
- all versions lower than v.2.0.5

Changing the server default

Setting the value for the server-level **DefaultDBCACHEPages** to be the largest of the DB Cache Pages values you have recorded may be overkill. When you change the server-level default setting in the configuration file, it becomes the default for every new and zero-configured database on the server.

To change it, open the configuration file in a text editor, find the parameter **DefaultDBCACHEPages** and change the number.. If it is commented with a '#' symbol, delete the '#' symbol.

V.1.0.x In the v.1.0.x config files, it is ***default_cache_pages***. Uncomment the line if necessary and make the entry **database_cache_pages=nnnn**, where *nnnn* is the new cache size.

For Superserver, the change will take effect the next time a first connection is made to the affected databases. For Classic, it will affect all connections made after the reconfiguration.

A pragmatic approach

Do not overrate the importance of the database cache. Any cache imposes its own overhead on the overall memory resources and the filesystem cache plays its own role in optimizing the system's read-write performance. There is always a point at which the real gain in overall server performance does not justify the cost of tying up resources for the “worst-case” demand scenario.

The best advice is: don't rush into cache size optimization as a “must-do” for Firebird. Work with the default settings during development and, for deployment, just verify that the amount of RAM available can accommodate the defaults.

Once into production conditions, use a monitoring tool to observe and record how reads and writes are satisfied from the cache for typical and extreme situations. If the statistics are not satisfactory, then begin considering optimization.

The first broad-brush optimization you can try is to increase the default cache to a size that will occupy approximately two-thirds of available free RAM. If there isn't enough RAM installed, install more.

At that point, start monitoring again. If it fails to improve things, repeat the exercise.

Verifying cache size

To verify the size of the database cache currently in use, execute the following commands in *isql*:

```
ISQL> CONNECT database_name;
ISQL> SET STATS ON;
ISQL> COMMIT;
Current memory = 415768
Delta memory = 2048
Max memory = 419840
Elapsed time = 0.03 sec
Buffers = 2048
Reads = 0
Writes 2
Fetches = 2
ISQL> QUIT;
```

After SET STATS ON, the empty COMMIT command prompts *isql* to display the information about memory and buffer usage. Read the Buffers= line to determine the current size of the cache, in pages.

Read-only Databases

By default, databases are in read-write mode when created. Read-write databases can not be on a read-only filesystem, even if they are used only for SELECT, because Firebird writes information about transaction states to a data structure in the database file.

A Firebird database can be deployed as a read-only file, providing the ability to distribute catalogs, albums of files and other non-maintained types of database on CDs and other read-only filesystems. Read-only databases can, of course, be accessed on read-write systems as well.



A read-only database is not the same thing as a database file that has its read-only attribute set on. File-copying a read-write database to a CD-ROM does not make it into a read-only database.

The application will need to be written so as to avoid requests that involve writing to the database or to trap the exceptions raised when they are attempted. The following will throw the error 335544765, “Attempted update on read-only database”:

- UPDATE, INSERT or DELETE operations
- metadata changes.
- operations that try to increment generators

External files

Any accompanying files linked to the database by having been declared with CREATE TABLE tablename EXTERNAL FILE 'filename' will also be opened read-only, even if the file itself is not carrying the filesystem read-only attribute.

Making a database read-only

Exclusive access is required to switch a database between read-write and read-only modes—see the instructions in Chapter 35, **Configuring Databases**, in the topic [Using the Tools](#). The mode-switch can be performed by the database owner or a user with SYSDBA rights.

Either *gfix* or *gbak* can be used :

- Using *gbak*, back up the database and restore it in read-only mode using the `-c[reate]` option. For example,
`gbak -create -mode read_only db1.fbk db1.fdb`
- Using *gfix*, issue a `-m[ode] read_only` command. For example,
`gfix -mode read_only db1.fdb`



Restore read-only databases with full page usage—use the `-use` switch to specify “use all space”. In a read-write database, pages are filled by default to 80% of page capacity because it can help to optimize the re-use of pages. Space reservation makes no sense in a read-only database and fully-filled pages are denser and faster.

Databases with ODS lower than 10, e.g., old InterBase 5 databases, cannot be made read-only.

Keeping a Clean Database

Firebird uses a **multi-generational architecture**. This means that multiple versions of data rows are stored directly on the data pages. When a row is updated or deleted, Firebird keeps a copy of the old state of the record and creates a new version. This proliferation of record back versions can increase the size of a database.

Background garbage collection

To limit this growth, Firebird continually performs garbage collection (GC) in the background of normal database activity. In Superserver, GC is literally a background operation performed by threads during idle times. In Classic and Superclassic, GC is performed cooperatively: everyone cleans up after everyone else. Each time a Firebird process or thread touches a table, it collects garbage left from the committing of recent transactions.

Garbage collection does nothing to get row versions that are caught up in unresolved transactions into a state where they can be flagged obsolete—they will not be visited during normal housekeeping activity. To completely sanitize the database, Firebird can perform database sweeping.

Sweeping

Database sweeping is a systematic way of removing all outdated row versions and freeing the pages they occupied so that they can be reused. Periodic sweeping prevents a database from growing unnecessarily large. Not surprisingly, although sweeping occurs in an asynchronous background thread, it can impose some cost on system performance.

By default, a Firebird database performs a sweep when the sweep interval reaches 20,000 transactions. Sweeping behavior is configurable, however: it can be left to run automatically, the sweep interval can be altered, or automatic sweeping can be disabled, to be run manually on demand instead.

Manual sweeping can be initiated from the command-line housekeeping program, *gfix*. Details are in Chapter 35, in the topic entitled [Sweeping](#). Several third-party desktop tools are available that provide a GUI interface for initiating manual sweeps.

Sweep interval

The Firebird server maintains an inventory of transactions. Any transaction that is uncommitted is known as an interesting transaction. The oldest of these “interesting” transactions (Oldest Interesting Transaction—OIT) marks the starting point for the sweep interval. If the sweep interval setting is greater than zero, Firebird initiates a full sweep of the database when the difference between the OIT and the Oldest Snapshot transaction passes the threshold set by the sweep interval.

For instructions to set (or disable) the sweep interval, see the topic [Sweep Interval](#) in Chapter 35.

Garbage collection during backup

Sweeping a database is not the only way to perform systematic garbage collection. Backing up a database achieves the same result, because the Firebird server must read every record, an action that forces garbage collection throughout the database. As a result, regularly backing up a database can reduce the need to sweep and helps to maintain better application performance.



Backup is NOT a replacement for sweep. While backup performs full database garbage collection, it does not clean up the transaction accounting as sweep does. The effects of sweeping—or neglecting to do so—are discussed in several topics throughout Chapter 25, Overview of Firebird Transactions.

For more information about the benefits of backing up and restoring, see Chapter 39, *Backing Up Databases*.

Objects and counters

The structures of objects that you create in a Firebird database can be modified, using ALTER statements. Firebird keeps a count of the ALTER statements for tables, views and stored procedures and bumps up the “format version” of the object each time an ALTER statement for it is committed. The allowed number of format versions per object is limited—currently 255, for all versions.

If *any one object* reaches the limit, the whole database is rendered unusable. A backup and restore with *gbak* will be required to make it serviceable.

Transaction counter

All operations in Firebird occur in transactions. Each transaction is uniquely numbered with a Transaction ID that is a 32-bit unsigned integer. When the integers “run out”, the database will be rendered unusable. Again, backup and restore will be needed to fix the situation.



The highest 32-bit integer is 2,147,483,647.

Validation and repair

Firebird provides utilities for validating the logical structures in databases, identifying minor problems and, to a limited extent, repairing them. A variety of such errors may appear from time to time, particularly in environments where networks are unstable or noisy or the power supply is subject to fluctuation. User behavior and application or database design deficiencies are also frequent causes of logical corruption.

Abnormal termination of client connections does not affect database integrity, since the Firebird server will eventually detect the lost connection. It preserves committed data changes and rolls back any left pending. Cleanup is more of a housekeeping issue, since data pages that were assigned for uncommitted changes are left as “orphans”. Validation will detect such pages and free them for reassignment.

The validation tools are capable of detecting and removing minor anomalies caused by operating system or hardware faults. Such faults usually cause database integrity problems, due to corrupt writes to or loss of data or index pages.

Data thus lost or damaged are not recoverable but their artefacts must be removed to restore database integrity. If a database containing these compromised structures is backed up, the database will not be capable of being restored. It is important, therefore, to follow a controlled course of action to detect errors, eliminate them if possible and get the database back into a stable state.

The issues of validation and repair and how the tools can help are discussed in detail in the topic *Analysing and Repairing Logical Corruption* in Chapter 35.



If you suspect you have a corrupt database, it is important to follow a proper sequence of recovery steps in order to avoid further corruption. The first, most important thing to do is ask or, if necessary, force all users to cancel their work and log out.

Appendix IX Database Repair How-To provides a detailed procedure for attempting to repair a corrupt or suspect database.

How to corrupt a Firebird database

Firebird is famously tolerant of trauma that are fatal to other DBMS systems. However, experience has shown up a few techniques that have proven useful if destruction of your database is among your objectives. The author wishes to share these database-killers with the reader.

1 Modify the system tables¹

Firebird stores and maintains all of the metadata for its own and your user-defined objects in—a Firebird database! More precisely, it stores them in relations (tables) right in the database itself. The identifiers for the system tables, their columns and several other types of system objects begin with the characters ‘RDB\$’.

Because these are ordinary database objects, they can be queried and manipulated just like your user-defined objects. However, just because you can does not say you should.

It cannot be recommended too strongly that you use DDL—not direct SQL operations on the system tables—whenever you need to alter or remove metadata.

2 Restore a backup to a running database

Two of the restore options in the *gbak* utility (**-r[ecreate_database]** and **-rep[lace_database]**) allow you to restore a *gbak* file over the top of an existing database by overwriting it. It is possible for this style of restore to proceed without warning while users are logged in to the database. Database corruption is almost certain to be the result.

Prior to v.2.0, the **-r** abbreviation was for **-replace_database**; from v.2.0 the **-r** abbreviation is applicable to **-r[ecreate_database]**, which acts like the **-c[reate]** switch and stops if the named database already exists, unless the **OVERWRITE** argument is present. No extra arguments are required for **-replace_database**. For now, it remains, in a deprecated status.

Your Admin tools and procedures must be designed to prevent any user (including SYSDBA) from overwriting your active database if any users are logged in.

If is practicable to do so, it is recommended to restore to spare disk space using the *gbak* **-c[reate]** option. Before making the restored database live, test it in the spare location using *isql* or your preferred admin tool.

3 Allow users to log in during a restore

If your organization likes living on the edge, then use the **-restore** switch and let users log in and perform updates. Restore recreates the database from scratch and, as soon as the tables are recreated, your users can, potentially at least, hit them with DML operations while referential integrity and other constraints are still in the pipeline. At best, they will cause exceptions and a stack of uncommitted transactions in the partly-constructed database. At worst, they will thoroughly break data integrity in divers irrecoverable ways.

4 Copy database files while users are logged in

Use any filesystem copying or archiving utility (DOS copy, xcopy, tar, gzip, WinZip, WinRAR, etc.) to copy files while any user (including SYSDBA) is logged in. The copy will be damaged but, worse, sector locking and/or caching by these programs can cause data loss and possibly corruption within the original file.

1. In a future version, probably v.3, the system tables will be made read-only.

Backup and Stand-by

Firebird 2 and higher come with two backup utilities: *gbak* and *nBackup*. They achieve backup in entirely different ways:

- For a backup, *gbak* writes a text file in a compressed format (XDR) that contains instructions for reconstructing the entire database, including all data. A *gbak* restore recreates the database with completely fresh metadata and uses DML to populate it with the data that has been compressed into the backup file. It also has roles in maintaining database health.

gbak can back up single-file or multi-file databases and restore them as single-file or multi-file. It can enable changes on restore of attributes such as page size and Owner.

For details and instructions, refer to [The *gbak* Utility](#) in Chapter 39.

- *nBackup*, available from the “2” series onward, is an incremental backup utility that works by keeping images of database pages. Working from a base image of the full database, it makes incremental files of changed pages, according to a user-defined cycle that can be at multiple levels. A restore is achieved by tracing a path from the base file through the latest incremental files down the levels until the most recent images of all pages are reconstituted.

nBackup does not have any capabilities to clear garbage or monitor data in any fashion. It does not support multi-file databases.

For details and instructions, refer to [Incremental Backup Tool \(*nBackup*\)](#) in Chapter 39.

Stand-by

Firebird has the ability to maintain a “database shadow” for each database under the purview of the server. A shadow is a file whose internal structure is almost identical to a database and whose data is an exact copy of the latest data and metadata in the database it shadows, including, unfortunately, any data that were corrupted in writing. Firebird has tools to create shadows and to activate them when needed. Shadowing is neither replication nor backup but it might have its uses in the event of a hard disk crash, provided it is being maintained on a physically separate hard disk, of course.

For details and instructions, refer to [Database Shadowing](#) in Chapter 39.

Dropping a Database

When a database is no longer required, it can be deleted (dropped) from the server. Dropping a database deletes all files pertaining to the entire database—primary and secondary files, shadow files, log files—and with it, of course, all its data.

The command to delete a database is DROP DATABASE and it takes no parameters. In order to execute the command, you need to be logged in to the database as its owner, as SYSDBA or as a user with root or Administrator system root privileges.

Syntax DROP DATABASE;

A dropped database cannot be recovered, so

- 1 be certain that you really want it to be gone forever
- 2 take a backup first if there is even a remote chance that you will need anything from it in future

CHAPTER

15

TABLES

In SQL terms Firebird tables are persistent base tables. The standards define several other types, including viewed tables, which Firebird implements as views (see chapter 23) and derived tables, which Firebird implements in more than one way.

About Firebird Physical Tables

Unlike desktop databases, such as Paradox and xBase databases, a Firebird database is not a series of “table files” physically organized in rows and columns. Firebird stores data, independently of their structure, in a compressed format, on database pages. It may store one or many records—or, correctly, rows—of a table’s data on a single page. In cases where the data for one row are too large to fit on one page, a row may span multiple pages.

Although a page which stores table data (a *data page*) will always contain only data belonging to one table, there is no requirement to store pages contiguously. The data for a table may be scattered all around the disk and, in multi-file databases, may be dispersed across several directories or disks. BLOB data are stored apart from the rows that own them, in another style of database page (a *BLOB page*).



Firebird can access text files that have data arranged as fixed length records and can read from or insert into them as though they were tables. For more details, refer to the later topic, [Using External Files as Tables](#) in this chapter.

Structural descriptions

Metadata—the physical descriptions of tables and their columns and attributes, as well as those of all other objects—are themselves stored in ordinary Firebird tables inside the database. RDB\$RELATIONS stores a row for each table, while RDB\$RELATION_FIELDS stores descriptive data for columns and links to the domains—stored in RDB\$FIELDS—that provide the technical details of their definitions.¹ The Firebird engine writes to these tables when database objects are created,

modified or destroyed. It refers to them constantly when carrying out operations on rows. These tables are known as *system tables*. Schemata for the system tables are in Appendix V.

Creating Tables

It is assumed that, having reached the point where you are ready to create tables, you have already prepared your data analysis and modeling and have very clear blueprint for the structures of your main tables and their relationships. In preparation for creating these tables, you need to have performed these steps:

- You have created a database to accommodate them. For instructions, refer to the previous chapter.
- You have connected to the database
- If you plan to use domains for the data type definitions of your tables' columns, you have already created the domains—refer to Chapter 11

Table ownership and privileges

When a table is created, Firebird automatically applies the default SQL security scheme to it. The person who creates the table (the owner), is assigned all SQL privileges for it, including the right to grant privileges to other users, triggers, and stored procedures. No other user, except the SYSDBA or equivalent, will have any access to the table until explicitly granted privileges.



This security is as good (or bad) as the security of access to your server. Anyone who can log in to your server can create a database. Anyone who can attach to a database can create tables in it. You can mitigate the inherent risks by limiting the locations where databases can be created and accessed. See the [DatabaseAccess](#) parameter in [firebird.conf](#).

For information about SQL privileges, refer to Chapter 37, *Database-Level Security*.

CREATE TABLE statements

The DDL for creating a table is the CREATE TABLE statement.

Syntax `CREATE TABLE table [EXTERNAL [FILE] 'filespec']
 (<col_def> [, <col_def> | <table-constraint> ...]);`

The first essential argument to CREATE TABLE is the table identifier . It is required, and must be unique among all table, view and procedure names in the database, otherwise you will be unable to create the table. You must also supply at least one column definition.

Defining columns

When you create a table in the database, your main task is to define the various attributes and constraints for each of the columns in the table.

Syntax for defining a column:

`<col_def> = col {datatype | COMPUTED [BY] (<expr>) | domain}`

1. RDB\$FIELDS stores a dedicated domain definition for every column created unless that column was defined using a user-defined domain.

```
[DEFAULT {literal |NULL |USER}]
[NOT NULL] [<col_constraint>]
[COLLATE collation]
```

The next sections list the required and optional attributes that you can define for a column.

Required attributes

- a column identifier (name), unique among the columns in the table.
- one of the following:
 - an SQL data type
 - an expression (expr) for a computed column marked by the `COMPUTED [BY]` keyword
 - a domain definition (domain) for a domain-based column

Columns are separated by commas.

Example

```
CREATE TABLE PERSON (
    PERSON_ID BIGINT NOT NULL,
    FIRST_NAME VARCHAR(35),
    LAST_NAMES VARCHAR (80),
    FULL_NAME COMPUTED BY FIRST_NAME || ' ' || LAST_NAMES,
    PHONE_NUMBER TEL_NUMBER);
```

The column `FULL_NAME` is a computed column, calculated by concatenating two other columns in the definition, `FIRST_NAME` and `LAST_NAMES`. We will come back to computed columns a little later. A `NOT NULL` constraint is applied to `PERSON_ID` because we want to make it a primary key—details later.

For the `PHONE_NUMBER` column we use the domain that was defined in our Chapter 11 example:

```
CREATE DOMAIN TEL_NUMBER AS VARCHAR(18)
CHECK (VALUE LIKE '(0%)%');
```

Columns based on domains

If a column definition is based on a domain, it can include a new default value, additional `CHECK` constraints, or a `COLLATE` clause that overrides one already defined in the domain definition. It can also include additional attributes or column constraints. For example, you can add a `NOT NULL` constraint to the column if the domain does not already define one.

A domain that is configured as `NOT NULL` cannot be overridden at column-level to be nullable.

For example, the following statement creates a table, `COUNTRY`, referencing a domain called `COUNTRYNAME`, which doesn't have a `NOT NULL` constraint:

```
CREATE TABLE COUNTRY (
    COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
    CURRENCY VARCHAR(10) NOT NULL);
```

We add the `NOT NULL` constraint to the column definition of `COUNTRYNAME` because we know it is going to be needed as the primary key of the `COUNTRY` table.

Optional attributes

One or more of the following optional attributes can be included in a column definition.

DEFAULT value

Defining a default value can save data entry time and prevent data entry errors when new rows are inserted into a table. If the row is inserted without including the column in the column list, a default value—if defined—can be automatically written into the column. In a column based on a domain, the column can include a default value that locally overrides any default defined for the domain.

For example, a possible default for a `TIMESTAMP` column could be the context variable `CURRENT_TIMESTAMP` (server date and time). In a (True/False) Boolean-style character column the default could be set to 'F' to ensure that a valid, non-null state was written on all new rows.

A default value must be compatible with the data type of the column and consistent with any other constraints on the column or its underlying domain. A default, as appropriate to data type, can be:

- a constant, e.g. some string, numeric or date value.
- a context variable, e.g. `CURRENT_TIMESTAMP`, `CURRENT_USER` , `CURRENT_CONNECTION`, etc.
- a predefined date literal such as 'NOW', 'TOMORROW', etc.
- `NULL` can be set as the default for any nullable column . Nullable columns default to `NULL` automatically but you may wish to override an unwanted domain-level default. Don't define this default on a column that has a `NOT NULL` constraint.



When relying on defaults, it must be understood that a default will be applied only upon insertion of a new row AND only if the INSERT statement does not include the defaulted column in its column list. If your application includes the defaulted column in the INSERT statement and sends NULL in the values list, then NULL will be stored, regardless of any default defined. If the column is not nullable, passing NULL will always cause an exception.

Example The following example defines a column `CREATED_BY` that defaults to the context variable `CURRENT_USER`:

```
CREATE TABLE ORDER (
    ORDER_DATE DATE,
    CREATED_BY VARCHAR(31) DEFAULT CURRENT_USER,
    ORDER_AMOUNT DECIMAL(15,2));
```

A new row is inserted by user `JILLIBEE`, omitting `CREATED_BY` from the column list:

```
INSERT INTO ORDER (ORDER_DATE, ORDER_AMT)
VALUES ('15-SEP-2014', 1004.42);
```

The table is queried:

```
SELECT * FROM ORDER;

...
ORDER_DATE  CREATED_BY      ORDER_AMOUNT
=====
...
15-SEP-2014  JILLIBEE         1004.42
...
```


CHARACTER SET

A CHARACTER SET can be specified for an individual character or text BLOB column when you define the column. If you do not specify a character set, the column assumes the character set of the domain, if applicable; otherwise, it takes the default character set of the database.

Example

```
CREATE TABLE TITLES_RUSSIAN (
    TITLE_ID BIGINT NOT NULL,
    TITLE_EN VARCHAR(100),
    TITLE VARCHAR(100) CHARACTER SET WIN1251);
```

Refer to Chapter 9, *Character Types*, for details about character sets and to Appendix VI for a list of available character sets.

A COLLATE clause

A COLLATE clause can be added to a CHAR or VARCHAR column to override a collation sequence otherwise defined for the column's character set by the underlying domain, if applicable. Collation sequence is not applicable to BLOB types.

Extending the example above to include a COLLATE clause:

```
CREATE TABLE TITLES_RUSSIAN (
    TITLE_ID BIGINT NOT NULL,
    TITLE_EN VARCHAR(100),
    TITLE VARCHAR(100) CHARACTER SET WIN1251 COLLATE PXW_CYRL);
```

Caution *Take care when applying COLLATE clauses to columns that need to be indexed. The index width limit of one-quarter of page size (or, worse, 253 bytes for an ODS 10.n database) can be drastically reduced by some collation sequences. Experiment first!*

Refer to Appendix VI for a list of available character sets and the collations for each.



You can get your own list, which may include more recently added collation sequences, by creating a new database and running the query listed in Chapter 9, under the topic Listing available collation sequences.

COMPUTED columns

A computed column is one whose value is calculated each time the column is accessed at run time. It can be a convenient way to access redundant data without the negative effects of actually storing it. Not surprisingly, such columns cannot perform like hard data—refer to the restrictions enumerated below.

Syntax 1

```
..,
<col_name> COMPUTED [BY] (<expr>)
..
```

Syntax 2 In versions 2.1 and higher:

```
..,
<col_name> GENERATED ALWAYS AS (<expr>)
...
```

There is no need to specify the data type—Firebird calculates an appropriate one. **<expr>** is any scalar expression that is valid for the data types of the columns involved in the calculation. External functions are fine to use, as long as you are sure that the libraries used by the functions will be available on all platforms where the database might be installed. (For more information about external functions, a.k.a. “UDF's”, refer to Chapter 20, *Expressions and Predicates*.)

Other restrictions exist for computed columns:

- Any columns that the expression refers to must have been defined before the computed column is defined, so it is a sensible practice to place computed columns last.
- A computed column cannot be defined as an ARRAY type or return an array.
- You can define a computed BLOB column by using a SELECT statement on a BLOB in another table but it is strongly recommended that you don't do this.
- Computed columns cannot be indexed
- Constraints placed on computed columns will be ignored or, in some cases, cause exceptions.
- Computed columns are output-only and read-only: including them in INSERT, UPDATE or INSERT OR UPDATE statements will cause exceptions



It is possible to create a computed column using a SELECT statement into another table. It is a practice to be avoided because of the undesirable dependencies involved. A properly normalized database model should not require it. DML syntax is available to obtain such subqueries at run-time or in a view.

Examples The following statement creates a computed column, FULL_NAME, by concatenating the LAST_NAME and FIRST_NAME columns.

```
CREATE TABLE PERSON (  
    PERSON_ID BIGINT NOT NULL,  
    FIRST_NAME VARCHAR(35) NOT NULL,  
    LAST_NAMES VARCHAR (80) NOT NULL,  
    FULL_NAME COMPUTED BY FIRST_NAME || ' ' || LAST_NAMES);  
  
/**/  
SELECT FULL_NAME FROM PERSON  
WHERE LAST_NAMES STARTING WITH 'Smi';  
FULL_NAME  
=====
```

Arthur Smiley
John Smith
Mary Smits



Notice the NOT NULL constraints on the two names being concatenated for this computed column. It is important to attend to such details with computed columns, because NULL as an element in a concatenation will always cause the result to be NULL.

The next statement computes two columns using context variables. This can be useful for logging the particulars of row creation:

```
CREATE TABLE SNIFFIT  
(SNIFFID INTEGER NOT NULL,  
 SNIFF COMPUTED BY (CURRENT_USER),  
 SNIFFDATE COMPUTED BY (CURRENT_TIMESTAMP));  
  
/**/  
SELECT FIRST 1 FROM SNIFFIT;  
SNIFFID SNIFF SNIFFDATE  
=====
```

1 SYSDBA 2014-08-15 08:15:35.0000

The next example creates a table with a calculated column (NEW_PRICE) using the previously created OLD_PRICE and PERCENT_CHANGE definitions:

```
CREATE TABLE PRICE_HISTORY (  
    PRODUCT_ID D_IDENTITY NOT NULL, /* uses a domain */  
    CHANGE_DATE DATE DEFAULT CURRENT_TIMESTAMP NOT NULL,  
    UPDATER_ID D_PERSON NOT NULL, /* uses a domain */  
    OLD_PRICE DECIMAL(13,2) NOT NULL,  
    PERCENT_CHANGE DECIMAL(4,2)  
        DEFAULT 0  
        NOT NULL  
    CHECK (PERCENT_CHANGE BETWEEN -50.00 AND 50.00),  
    NEW_PRICE COMPUTED BY  
        (OLD_PRICE + (OLD_PRICE * PERCENT_CHANGE / 100)) );
```

Constraints

In the parlance of relational databases, any restriction imposed on the format, range, content or dependency of a data structure is known as a constraint. Firebird provides several ways to implement constraints, including both formal, standards-defined integrity and referential constraints and user-defined CHECK constraints.

Constraints are visible to all transactions that access the database and are automatically enforced by the server. They exist in their own right as objects in a Firebird database and can be assigned custom identifiers. Each constraint is uniquely represented in the metadata, with the rules and dependencies of each being defined through regular relationships between the system tables.

Scope of Constraints

Constraints vary in their scope of action. Some—such as NOT NULL, for example—are applied directly to a single column (*column constraints*) while others—such as PRIMARY KEY and some CHECK constraints—take effect at table level (*table constraints*). The FOREIGN KEY constraint has table-to-table scope.

Integrity Constraints

Integrity constraints impose rules that govern the state of acceptable data items or a relationship between the column and the table as a whole—often both. Examples are NOT NULL (rejects input that has unknown value), UNIQUE (requires that an incoming item has no matching value in that column anywhere in the table) and PRIMARY KEY (combines both of the others and also “represents” the table for referential relationships with other tables).

Each of the integrity constraints is discussed individually in detail later in this chapter.

The referential constraint

The referential constraint is implemented as FOREIGN KEY. A foreign key constraint exists only in the context of another table and a unique key from that table, signalled implicitly or explicitly by the REFERENCES clause of its definition.

Tables that are linked in a foreign key relationship are said to be bound by a referential integrity constraint. Thus, any column or group of columns constrained by a PRIMARY KEY or UNIQUE constraint is also potentially subject to referential constraints.

The subject is discussed in detail in Chapter 17, *Referential Integrity*.

Named constraints

When declaring a table-level or a column-level constraint, you can optionally name the constraint using the CONSTRAINT clause. If you omit the CONSTRAINT clause, Firebird generates a unique system constraint name. Constraints are stored in the system table, RDB\$RELATION_CONSTRAINTS.

Although naming a constraint is optional, assigning a meaningful name with the CONSTRAINT clause can make the constraint easier to find for changing or dropping or when its name appears in a constraint violation error message. Consider, for example, a constraint with the assigned name INTEG_99: you could spend a lot of time chasing around the system tables to find out what it belongs to and what might have caused a violation.

Apart from its benefits for self-documentation, this style is particularly useful for distinguishing the key definitions from the column definitions in scripts that will be kept as system documentation.

PRIMARY KEY and FOREIGN KEY names

Naming a constraint has special implications for PRIMARY KEY and FOREIGN KEY constraints, particularly from Firebird 1.5 forward. It is possible to override Firebird's native naming rules for keys:

In all versions, a supplied name will override the default name INTEG_nn and apply the supplied name to the constraint. The supporting indexes will, by default, have the same name as the named constraint, if the constraint is named; other wise they will be named RDB\$PRIMARYnn and RDB\$FOREIGNnn, respectively.

V.1.0.x In versions 1.0.x, the default index name (RDB\$PRIMARYnn or RDB\$FOREIGN_nn) is enforced

The constraint-naming behaviors are described in more detail below and in Chapter 17, *Referential Integrity*.

The NOT NULL Constraint

The optional NOT NULL constraint is a column-level constraint that can be applied to force the user to enter a value.

Because of its role in the formation of keys, you need to be aware of certain restrictions pertaining to the NOT NULL constraint:

- it must be applied to the definition of any column that will be involved in a PRIMARY KEY or UNIQUE constraint
- it cannot be removed from a domain or column by an ALTER DOMAIN or ALTER TABLE ALTER COLUMN statement nor by overriding a domain at column level—do not use a domain that is constrained NOT NULL to define a column that is allowed to be NULL

- it is recommended in the definition of any column that will be involved in a UNIQUE constraint or a unique index since one and only one NULL is allowed per key segment in such indexes and allowing NULLs in these structures is not good practice, normally.

V.1.0.x

Firebird 1.0 versions do not permit any NULLs in UNIQUE constraints or UNIQUE indexes.



Firebird does not support a NULLABLE attribute, as some non-standard DBMS do. In compliance with standards, all columns in Firebird are nullable unless explicitly constrained to be NOT NULL. Null is not a value, so any attempt to input null to the column or set it to null will cause an exception.

For more insight into NULL, refer to the topics [*Demystifying NULL*](#) in Chapter 6 and [*Considering NULL*](#) in Chapter 20.

The PRIMARY KEY constraint

PRIMARY KEY is a table-level integrity constraint—a set of enforceable rules—which formally earmarks a column or group of columns as the unique identifier of each row in the table. While the PRIMARY KEY constraint is not itself a *referential constraint*, it is usually a mandatory part of any referential constraint, being potentially the object of the REFERENCES clause of a FOREIGN KEY constraint.

A table can have only one primary key. When you define the constraint, Firebird automatically creates the required index, using a set of naming rules which cannot be overridden. The names of primary key indexes are discussed below.

Simply creating a unique index does not create a primary key. A primary key is not an index, but a constraint in its own right. Creating a PRIMARY KEY constraint does, however, create the required index using the columns enumerated in the constraint declaration.



Do not import an existing primary index from a file-based legacy system or create such an index in anticipation of declaring a primary key constraint. Firebird cannot piggy-back a primary key constraint onto an existing index—at least up to and including release 2.5—and the query optimizer does not perform properly if indexes are duplicated.

Choosing a primary key

Identifying candidate columns to be the primary key is a science in itself and beyond the scope of this Guide. Many worthy tomes have been written on the subject of normalization, the process of eliminating redundancy and repeating groups in sets of data and arriving at a correct identification of the element that uniquely represents one single row set in the table. If you are a newcomer to relational databases, the value of investing in a good book about data modeling cannot be stressed enough.

A primary key candidate—which may be one column or a group of columns—has two unbreakable requirements:

- The NOT NULL attribute must be declared for all columns in the group of one or more columns which will be used. The integrity of the key can be enforced only by comparing values and NULL is not a value.
- The column or group of columns has to be unique—that is, it cannot occur in more than one row in the table. A driver's licence or social security number might be considered, for example, because they are generated by systems that are presumed not to issue duplicate numbers.

To these theoretical “givens” must be added a third one:

- The total size (width) of the candidate key must be within the limit of one-quarter of the page size. In Firebird 1.X, the limit is even smaller: 253 bytes or less. This is not simply a matter of counting characters. The implementation limit will be reduced—in some cases, drastically—if there are multiple columns, non-binary collations or multi-byte character sets involved.

How real data can defeat you

Using the EMPLOYEE table from the `employee.fdb` database in the Firebird `root/examples/empbuild` directory (`employee.gdb` in the v.1.0.x kits), let's illustrate how real data can defeat your theoretical assumptions about uniqueness. Here is a declaration that shows, initially, the meaningful data stored in this table:

```
CREATE TABLE EMPLOYEE (
    FIRST_NAME VARCHAR(15) NOT NULL, /* assumption: an employee must have
                                         a first name */
    LAST_NAME VARCHAR(20) NOT NULL, /* assumption: an employee must have
                                         a last name */

    PHONE_EXT VARCHAR(4),
    HIRE_DATE DATE DEFAULT CURRENT_DATE NOT NULL,
    DEPT_NO CHAR(3) NOT NULL,
    JOB_CODE VARCHAR(5) NOT NULL,
    JOB_GRADE SMALLINT NOT NULL,
    JOB_COUNTRY VARCHAR(15) NOT NULL,
    SALARY NUMERIC (15, 2) DEFAULT 0 NOT NULL,
    FULL_NAME COMPUTED BY FIRST_NAME || ' ' || LAST_NAME
);
```

This structure in fact has no candidate key. It is not possible to identify a single employee row by using (FIRST_NAME, LAST_NAME) as the key, since the combination of both elements has a medium-to-high probability of being duplicated in the organization. We could not store records for two employees named “John Smith”.

In order to get a key, it is necessary to invent something. That “something” is the mechanism known as a *surrogate key*.

Surrogate keys

We have already visited the surrogate key in the introductory topic on keys in Chapter 14. A surrogate primary key is a value of guaranteed uniqueness and no semantic content, that substitutes for the key in a table structure that cannot provide a candidate key from within its own structure. The EMPLOYEE table therefore introduces EMP_NO (declared from a domain) to take this surrogate role for it:

```
CREATE DOMAIN EMPNO SMALLINT ;
COMMIT;
ALTER TABLE EMPLOYEE
    ADD EMP_NO EMPNO NOT NULL,
    ADD CONSTRAINT PK_EMPLOYEE
    PRIMARY KEY(EMP_NO) ;
```

This database also maintains a generator named EMP_NO_GEN and a Before Insert trigger named SET_EMP_NO on the EMPLOYEE table, to produce a value for this key whenever a new row is inserted. The topic [*Implementing Auto-Incrementing Keys*](#) in Chapter

30, describes this technique in detail. It is the recommended way to implement surrogate keys in Firebird.

You may wish to consider the benefits of using a surrogate primary key not just in cases where the table cannot supply candidates but also in cases where your candidate key is composite. Any columns containing data that are meaningful and/or editable are non-atomic. Using non-atomic columns for keys is poor practice in general. For more discussion, read on to the topic *Atomicity of PRIMARY KEY columns* later in this section.

Composite primary keys

During data analysis, it sometimes happens that no single unique column can be found in the data structure. Theory suggests that the next best thing is to look for two or more columns that, when grouped together as the key, will ensure a unique row. When multiple columns are conjoined to form a key, the key is called a *composite key* or, sometimes, a *compound key*.

If you come to Firebird with a cargo of background experience working with a DBMS such as Paradox, using composite keys to implement hierarchical relationships, it can be quite hard to part with the notion that you cannot live without them. Yet, in practice, composite keys should be considered with a high degree of restraint in a DBMS like Firebird, which does not track through disk-based physical index structures to implement relationships.

Firebird does not need composite indexes and, more to the point, they do impose some problems, both for development and, when large tables are involved, for performance:

- Composite keys are typically composed of non-atomic key elements—that is, the columns selected have semantic meaning (they are “significant as data”) and are almost certainly vulnerable to external changes, redundancy, duplication and typographical errors.
- Any foreign keys in other tables that reference this table will have to propagate every element of the composite key. Referential integrity is at risk from the use of non-atomic keys. A combination of non-atomic elements compounds the risk.
- Keys—foreign, as well as primary—have mandatory indexes. Composite indexes have stricter size limits than single-column indexes.
- Composite indexes tend to be large. Large indexes use more database pages, causing indexed operations (sorts, joins, comparisons) to be slower than is necessary.

Atomicity of PRIMARY KEY columns

It is recommended practice to avoid involving in your primary and foreign keys any column which is meaningful as data. It violates one of the primary principles of relational database design, that of *atomicity*. The atomicity principle requires that each item of data exist completely in its own right, with a single, internal rule governing its existence.

For a primary key to be atomic, it should be beyond the reach of human decision. If a human has to spell it or type it, it is not atomic. If it is subject to any rule except the non-nullable, unique requirements, it is not atomic. Using the earlier example, even a systematic number such as a driver’s licence or a social security number does not have the atomicity required for a primary key, because it is subject to an external system.

Syntaxes for declaring the primary key

Several syntaxes are available for assigning the PRIMARY KEY constraint to a column or group of columns. All columns that are elements in a PRIMARY KEY must be previously defined as NOT NULL. Since it is not possible to add a NOT NULL constraint to a

column after it has been created, it is essential to take care of this constraint before applying the additional constraint.

The PRIMARY KEY constraint can be applied in any of the following phases of metadata creation:

- in the column definition, during CREATE TABLE or ALTER TABLE, as part of the column's definition set
- in the table definition, during CREATE TABLE or ALTER TABLE, as a separately-defined table constraint

Defining PRIMARY KEY as part of a column definition

In the following sequence, a non-nullable domain is defined and committed ahead; then the primary key column is defined using that domain and, simultaneously, the PRIMARY KEY constraint is applied to the table immediately:

```
CREATE DOMAIN D_IDENTITY AS BIGINT NOT NULL;
COMMIT;
CREATE TABLE PERSON (
    PERSON_ID D_IDENTITY PRIMARY KEY,
    ...
);
```

Firebird creates a table constraint with a name like INTEG_nn and an index with a name like RDB\$PRIMARYnn. With this method, you cannot influence what these names will be, nor change them.



nn in each case is a number spun from a generator. The two numbers are unrelated.

The effect is similar if you use the same approach when adding a column using ALTER TABLE and make it the primary key:

```
ALTER TABLE BOOK
    ADD BOOK_ID D_IDENTITY PRIMARY KEY;
```

Defining PRIMARY KEY as a named constraint

Another way to define the primary key in the table definition is to add the constraint declaration at the end of the column definitions. The constraint declarations are placed last because they are dependent on the existence of the columns to which they apply. This method gives you the option of naming the constraint. The following declaration names the primary key constraint as PK_atable:

```
CREATE TABLE atable (
    ID BIGINT NOT NULL,
    ANOTHER_COLUMN VARCHAR(20),
    CONSTRAINT PK_atable PRIMARY KEY(ID) );
```

Now, instead of the system-generated name RDB\$PRIMARYnnn, Firebird stores PK_atable as the name of the constraint. Except in Firebird 1.0.x, it also applies the user-defined constraint name to the enforcing unique index: in this example, the index will be named PK_atable (but INTEG_nn in v.1.0.x).

Using a custom index

It is possible to ask Firebird to enforce the primary key with a descending index. For this, there is an optional syntax extension in the form of the USING clause, enabling constraint-supporting indexes to be defined as either ASC[ENDING] or DESC[ENDING] and to have a name that is different to that of the named constraint. It cannot be used as a way to utilise an index that already exists.

ASC and DESC determine the direction of the search order—lowest or highest first. The concept is discussed in more detail in the next chapter, *[Indexes](#)*.



If you specify a DESCENDING index for a primary or unique constraint, you must be sure to specify USING DESC INDEX when defining any foreign keys that reference it.

The following statement will create a primary key constraint named PK_ATEST and enforce it by creating a descending index named IDX_PK_ATEST:

```
CREATE TABLE ATEST (
    ID BIGINT NOT NULL,
    DATA VARCHAR(10));
COMMIT;

ALTER TABLE ATEST
    ADD CONSTRAINT PK_ATEST PRIMARY KEY(ID)
    USING DESC INDEX IDX_PK_ATEST;
COMMIT;
```

The create-time syntax will work, too:

```
CREATE TABLE ATEST (
    ID BIGINT NOT NULL,
    DATA VARCHAR(10),
    CONSTRAINT PK_ATEST PRIMARY KEY(ID)
    USING DESC INDEX IDX_PK_ATEST;
```

Adding a primary key to an existing table

The addition of table constraints can be deferred. It is a common practice for developers to define all of their tables without any table constraints and to add them subsequently, using a separate script. The rationale behind this practice is good: large scripts notoriously fail because the author overlooked some dependency. It simply causes fewer headaches to build databases in a sequence that eliminates the time and spleen spent on patching dependency errors and re-running scripts.

Typically, in the first script, we declare and commit the tables:

```
CREATE TABLE ATABLE (
    ID BIGINT NOT NULL,
    ANOTHER_COLUMN VARCHAR(20),
    < more columns > );
CREATE TABLE ANOTHERTABLE (
    ... );
...
COMMIT;
ALTER TABLE ATABLE
```

```
ADD CONSTRAINT PK_ATABLE
PRIMARY KEY(ID);
ALTER TABLE ANOTHERTABLE...
```

and so on.

In Chapter 17, when exploring FOREIGN KEY definitions, the benefits of building databases in a dependency-safe sequence will become obvious.

The UNIQUE constraint

A UNIQUE constraint, like a primary key, ensures that no two rows have the same value for a specified column or group of columns. You can have more than one UNIQUE constraint defined for a table, but it cannot be applied to the same set of columns that is used for either the PRIMARY KEY or another UNIQUE constraint.

A unique constraint, in fact, actually creates a unique key that has virtually the same powers as the primary key. It can be selected as the controlling key for a referential integrity constraint. This makes it useful for situations where you define a thin, surrogate primary key for atomicity and to improve performance of join and search operations, but you want to keep the option to form an alternative FOREIGN KEY link on the unique key for occasional use.



Although a nullable field is allowed in a UNIQUE constraint, no more than one NULL is allowed in each segment of the supporting index. Unless you have a specific design requirement for certain records with NULLs in the segments, simplify your life and make all the key fields non-nullable.

V.1.0.x In Firebird 1.0.x, the NOT NULL attribute must be applied to all of the columns on which the UNIQUE constraint will operate.

Like the primary key constraint, UNIQUE creates its own mandatory, unique index to enforce its rules. Naming of both the constraint and the index follows the same rules of behavior applicable to other keys: you can customise the name and direction of the supporting index by naming the constraint and/or by applying the optional USING <index-name> clause

. The following example in *isql* illustrates the naming behavior:

```
SQL> CREATE TABLE TEST_UQ (
CON> ID BIGINT NOT NULL,
CON> DATA VARCHAR(10),
CON> DATA_ID BIGINT NOT NULL);
SQL> COMMIT;
SQL> ALTER TABLE TEST_UQ
CON>ADD CONSTRAINT PK_TEST_UQ PRIMARY KEY(ID),
CON>ADD CONSTRAINT UQ1_DATA UNIQUE(DATA_ID) ;
SQL> COMMIT;
SQL> SHOW TABLE TEST_UQ;
ID          BIGINT          NOT NULL
DATA        VARCHAR(10)  NULLABLE
DATA_ID     BIGINT          NOT NULL
CONSTRAINT PK_TEST_UQ:
    Primary key (ID)
CONSTRAINT UQ1_DATA:
```

```

Unique key (DATA_ID)
SQL> SHOW INDICES TEST_UQ;
PK_TEST_UQ UNIQUE INDEX ON TEST_UQ(ID)
UQ1_DATA UNIQUE INDEX ON TEST_UQ(DATA_ID)
SQL> /* optional USING clause sets a custom name for the supporting index
CON> and, if a for the supporting index
SQL> CREATE TABLE TEST2_UQ (
CON> ID BIGINT NOT NULL,
CON> DATA VARCHAR(10),
CON> DATA_ID BIGINT NOT NULL,
CON> CONSTRAINT PK_TEST2_UQ PRIMARY KEY
CON> USING INDEX IDX_PK_TEST2_UQ);
SQL> COMMIT;

```



Remember the mantra: “an index is not a key”. You can define unique indexes—see the next chapter for details—but making a unique index does not create a unique key. If there is a chance that you might need to use a uniquely indexed column or structure as a key, consider defining the constraint instead.

CHECK constraints

A CHECK constraint is used for validating incoming data values—it enforces a match condition or requirement that a value must meet in order for an insert or update to succeed. It cannot change the incoming value—it will return a validation exception if the input fails the check.



CHECK constraints are applied after “Before” triggers have fired. Use a trigger when you need to perform a validation and conditionally change it to a valid one. “Before” and “After” triggers are discussed in detail in Chapter 30.

In a table definition, a CHECK constraint applies at table level. Unlike the CHECK constraints applied to domain definitions, its checked element is expressed as a column reference, not a generic value.

For example, on a domain, a CHECK clause might be

```
CHECK (VALUE > 10)
```

In a table definition, the same conditioning for a column named ACOLUMN would be expressed as

```
CHECK (ACOLUMN > 10)
```

CHECK activity

A CHECK constraint is active in both INSERT and UPDATE operations. Although it is a table-level constraint, its scope can range from column-level, through row-level and, although it is not recommended, to table-level and even beyond the boundaries of the table. It guarantees data integrity only when the values being verified are in the same row as the value being checked.



Beware of using expressions that compare the value with values in different rows of the same table or in different tables, since any row other than the current one is potentially in the process of being modified or deleted by another transaction. Especially, do not rely on a CHECK constraint to enforce a referential relationship!

.The search condition can

- verify that the value entered falls within a defined range
- match the value with a list of allowed values
- compare the value with a constant, an expression or with data values in other columns of the same row

Constraint restrictions

- A column can have only one CHECK constraint, although its logic can be expressed as a complex search condition—one constraint, many conditions.
- A CHECK constraint on a domain-based column cannot override the inherited domain-level check. The column definition can use a regular CHECK clause to add additional constraint logic to the inherited constraint. It will be ANDed to the inherited conditions.
- A CHECK constraint cannot refer to a domain.

Syntax of the CHECK constraint

A CHECK constraint can be written to support practically any validation rule—theoretically, almost any search condition will be accepted. It is important for the designer to choose conditions that are reasonable and safe, since they affect every insert and update operation on the table.

```
CHECK (<search condition>);
<search_condition> =
<val> <operator> {<val> |(<select_one>)}
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
| <val> IS [NOT] NULL
| <val> {[NOT]{=<|>}}|>=<=<
{ALL |SOME |ANY}(<select_list>)
| EXISTS (<select_expr>)
| SINGULAR (<select_expr>)
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
|(<search_condition>)
|NOT<search_condition>
| <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>
```

Consult Chapter 20, *Expressions and Predicates*, for syntaxes for setting the various styles of search condition.

Example This constraint tests the values of two columns to ensure that one is greater than the other. Although it also implies NOT NULL conditioning on both columns—the check will fail if either column is null—it does not confer NOT NULL constraint on the column:

```
CHECK (VALUE (COL_1 > COL_2));
```

The check will fail if the arithmetic test fails or if either COL_1 or COL_2 is null. This succeeds:

```
INSERT INTO TABLE_1 (COL_1, COL_2) VALUES (6,5);
```

Using External Files as Tables

In the current SQL argot, Firebird supports the external virtual table, or EVT. File-system files in text format can be read and manipulated by Firebird as if they were tables—albeit with considerable limitations arising from the fact that they are not internal database objects. Other applications can thus exchange data with a Firebird database, independently of any special transforming mechanism. External tables can be converted to internal tables.

The `EXTERNAL FILE` clause enables a table to be defined with its row structure mapping to fixed length “fields” in “records” (usually delimited by line-feeds) that reside in an external file. Firebird can select from and insert into such a file as if it were a regular table. It cannot, however, perform update or delete operations on external tables.

The text file containing the data must be created on or copied to a storage device that is physically under the control of the server—as usual, no NFS devices, shares or mapped drives.

Shared access by Firebird and other applications at file level is not possible. Firebird requires exclusive access during times when it has the file open in a transaction. At other times, the file can be modified by other applications.

Syntax The `CREATE TABLE` statement for activating or creating an external file defines both the external file specification—local location and file name—and the typed Firebird columns represented by the structure of the contained records.

```
CREATE TABLE EXT_TBL
    EXTERNAL FILE filespec
    (columndef [,columndef,...],
    [line_delimiter_1 CHAR(1)
    [, line_delimiter_2 CHAR(1)]]);
```

filespec is the fully qualified local path and file specification for the external data file. The file need not exist at the time the table is created. However, from Firebird 1.5 forward, the `CREATE` statement will fail if the **filespec** refers to an unconfigured external file location—see the topic *Securing external files*, below, and the `ExternalFileAccess` configuration topic in Chapter 3.

columndef is an ordinary Firebird column definition. Non-character data types can be specified, provided every string extracted from the column’s location in the external record is capable of being cast implicitly to that type

line_delimiter is an optional final column or pair of columns that can be defined to read the characters used by the file system to mark the end of a line of text. Although it makes reading the file easier for humans, it is not a requirement in a fixed format record unless programs that are going to read the data require it..

- on Linux/UNIX, this is the single character ASCII 10, the linefeed character
- on Windows, it is the ordered pair ASCII 13 (carriage return) followed by ASCII 10
- on Mac OS, it is ASCII 10 followed by ASCII 13
- other operating systems may use other variations or other characters

Restrictions and recommendations

Because the content of external files is beyond the control of the Firebird engine, certain restrictions and precautions are needed to ensure the integrity of both the data and the server when deploying them.

Securing external files

By default, access to external files is blocked by the **ExternalFileAccess** setting in `firebird.conf` being configured as `NONE`. A list of directories can be configured to restrict the locations where Firebird will search for or create external files. It is also possible to configure open access to anywhere in the file system. This is NOT recommended.

When deploying a database that uses this feature, make sure you set up a configuration for external files that cannot be compromised through either malicious or careless access.

For more information, refer to the parameter ExternalFileAccess in Chapter 34.

Format of external data

Firebird will create the external file itself if it does not find it in the location specified in the `CREATE EXTERNAL TABLE '<filespec>'` specification. If the file already exists, each record must be of fixed length, consisting of fixed-length fields that exactly match the declared maximum byte-length of the column specifications in the table definition. If the application that created the file uses hard line breaks, e.g. the two-byte carriage return and line break sequence in Windows text files, include a column to accommodate this sequence—see End-of-line characters, below.

BLOB and array data cannot be read from or written to an external file.

Most well-formed number data can be read directly from an external table and, in most cases, Firebird will be able to use its internal casting rules to interpret it correctly. However, it may be easier and more precise to read the numbers into character columns and, later, convert them using the `CAST(..)` function.



*Make sure you allow enough width to accommodate your data. For some guidelines on sizes, refer to the relevant chapter for the particular data type in Part Two, **Firebird Data Types and Domains**. In Chapter 6, the topic Valid Conversions contains a tabulated summary of the rules for data type conversions.*

CHAR vs VARCHAR

Using `VARCHAR` in the column definition of an external string field is not recommended because it is not a readily portable format:

`<2-byte unsigned short><string of character bytes>`

Varchar requires the initial 2-byte unsigned short to include the number of bytes in the actual string, and the string immediately follows. This is difficult or impossible for many external applications to achieve and it simply isn't worth the trouble. For this reason, favour `CHAR` over `VARCHAR` for string fields and ensure that the feeding application pads the strings to full length.

End-of-line characters

When you create the table that will be used to import the external data, you must define a column to contain the end-of-line (EOL) or new-line character if the application that created the file includes it. The size of this column must be exactly large enough to contain

a particular system's EOL symbol (usually one or two bytes). For most versions of UNIX, it is 1 byte. For Windows and Macintosh, it is 2 bytes.

Tips for inserting non-printable characters

When inserting to an external file, the function `ASCII_CHAR(decimal_ASCII_code)` (or, for versions prior to v.2.1, the external function of the same name from the `ib_udf` function library) can be used to pass the non-printable characters as an expression to the line delimiter columns in the SQL statement. For example, to insert a carriage return and line feed into a column:

```
INSERT INTO MY_EXT_TABLE (
    <COLUMNS...>,
    CRLF)
VALUES (
    <column_values...>,
    ASCII_CHAR(13) || ASCII_CHAR(10));
```

An alternative is to create a table to store any non-printable characters your applications might need to store. Create a table for the purpose and create a regular text file on the same platform as the server, using an editor that “displays” non-printable characters. Open your NPC table using an interactive tool and copy-paste the characters from the file directly to the table. For statements performing inserts to the external file, the character can be subqueried from this NPC table.

Operations

Only INSERT and SELECT operations can be performed on the rows of an external table. Attempts to update or delete rows will return errors.

Because the data are outside the database, operations on an external table are not under Firebird's record version control. Inserts therefore take effect immediately and cannot be rolled back.



If you want your table to be under transaction control, create another, internal Firebird table, and insert the data from the external table into the internal one.

Importing external files to Firebird tables

To import an external file into a Firebird table, begin by making sure that you have set up the appropriate access conditions—refer to Chapter 37 regarding the server parameter `ExternalFileAccess` in the topic `Server Configuration`.

1 Create a table to view raw data

Create a Firebird table that allows you to view the external data. Declare all columns as `CHAR`. The text file containing the data must be on the server. In the following example, the external file exists on a UNIX system, so the EOL character is 1 byte.

```
CREATE TABLE EXT_TBL EXTERNAL FILE 'file.txt' (
    FNAME CHAR(10),
    LNAME CHAR(20),
    HDATE CHAR(10),
    NEWLINE CHAR(1));
COMMIT;
```

2 Create the working table

Create another Firebird table that will eventually be your working table. Include a column for the EOL character if you expect to export data from the internal table back to an external file later:

```
CREATE TABLE PERSONNEL (  
    FIRST_NAME VARCHAR(10),  
    LAST_NAME VARCHAR(20),  
    HIRE_DATE DATE,  
    NEW_LINE CHAR(1));  
  
COMMIT;
```

3 Populate the file

Using a text editor, or an application that can output fixed-format text, create and populate the external file. Make each record the same length, pad the unused characters with blanks, and insert the EOL character(s) at the end of each record.

The number of characters in the EOL string is platform-specific—refer to the notes above.

The following example illustrates a fixed-length record with a length of 41 characters. “b” represents a blank space, and “n” represents the EOL:

```
12345678901234567890123456789012345678901  
fname.....lname.....hdate.....n  
CaitlinbbbCochranebbbbbbbbbbb2014-12-10n  
AlexanderbGalbraithbbbbbbbbbbb2013-10-01n  
NicholasbbMailaubbbbbbbbbbbb2012-10-05n  
RosebbbbbbGalbraithbbbbbbbbbbb2011-07-01n  
MogginzbLeChatbbbbbbbbbbb2011-09-21n
```

4 Test the file

A SELECT statement from table EXT_TBL returns the records from the external file:

```
SELECT FNAME, LNAME, HDATE FROM EXT_TBL;  
  
FNAME      LNAME      HDATE  
=====  =====  =====  
Caitlin    Cochrane    2014-12-10  
Alexander  Galbraith   2013-10-01  
Nicholas   Mailau      2012-10-05  
Rose       Galbraith   2011-07-01  
Mogginz    LeChat      2011-09-21
```

5 Complete the import

Insert the data into the destination table.

```
INSERT INTO PERSONNEL  
    SELECT FNAME, LNAME, CAST(HDATE AS DATE),  
    NEWLINE FROM EXT_TBL;  
  
COMMIT;
```



If you try to access the external file whilst it is still opened by another application, the attempt will fail. The reverse is also true. Furthermore, in versions prior to v.2.0, once your application has opened the file as a table, it will be unavailable to other applications until your application disconnects from the database.

Now, when you perform a SELECT from PERSONNEL, the data from your external table will appear in converted form:

```
SELECT FIRST_NAME, LAST_NAME, HIRE_DATE
FROM PERSONNEL;
FIRST_NAME LAST_NAME      HIRE_DATE
=====
Caitlin    Cochrane             10-DEC-2014
Alexander  Galbraith             01-OCT-2013
Nicholas   Mailau                05-OCT-2012
Rose       Galbraith             01-JUL-2011
Mogginz    LeChat               21-OCT-2011
```

Exporting Firebird tables to an external file

Carrying on with the example illustrated in the previous section, the steps for exporting data to our external table are similar:

1 Clear the external file

Open the external file in a text editor and remove everything from the file. Exit from the editor and again perform the SELECT query on EXT_TBL. It should be empty.

2 Pump out the data

Use an INSERT statement to copy the Firebird records from PERSONNEL into the external file, file.txt:

```
INSERT INTO EXT_TBL
SELECT
    FIRST_NAME, LAST_NAME,
    cast(HIRE_DATE AS VARCHAR(11),
    ASCII_CHAR(10)
FROM PERSONNEL
    WHERE FIRST_NAME LIKE 'Clau%';
```

3 Test the external table

Now, querying the external table:

```
SELECT FNAME, LNAME, HDATE FROM EXT_TBL;
FNAME      LNAME      HDATE
=====
Caitlin    Cochrane     10-DEC-2014
....
Mogginz    LeChat       21-OCT-2011
```

Converting external tables to internal tables

It is possible to convert the current data in external tables to an internal tables. The means to do this is to back up the database using the *gbak* utility with the **-convert** switch (abbreviation **-co**). All external tables defined in the database will be converted to internal tables by restoring the backup. Afterwards, the external table definition will be lost.

For more information, refer to the section [The *gbak* Utility](#) in Chapter 39, **Backing Up Databases**.

Dropped database

If you use DROP DATABASE to delete the database, you must also remove the external file—it will not be automatically deleted as a result of DROP DATABASE.

Altering Tables

The ALTER TABLE statement is used for changing the structure of a table: adding, changing or dropping columns or constraints. One statement can encompass several changes, if required. To submit an ALTER TABLE request, you need to be logged in as the table's creator (owner), SYSDBA or an equivalent superuser.



Alterations to each table, or to its triggers, are reference-counted. Any one table can be altered at most 255 times before you must back up and restore the database. However, the reference count is not affected by switching a trigger on and off using (and only this) syntax variant:

`ALTER TRIGGER triggername ACTIVE | INACTIVE`

For more information about ALTER TRIGGER, consult the topic [Changing Triggers](#) in Chapter 30.

Any data format conversions required by changes are not performed *in situ* by the engine. Firebird stores the new format description and delays the translation until the data are needed. It introduces a performance hit that could have an unanticipated impact on a user's work. Plan to perform a backup and restore after changes in table structures if the database contains any data.

Preparing to use ALTER TABLE

Before modifying or dropping columns or attributes in a table, you need to do three things:

- 1 Make sure you have the proper database privileges.
- 2 Save the existing data.
- 3 Drop any dependency constraints on the column.
- 4 Plan a backup and restore after changes in table structures if the database contains any data.

Altering Columns in a Table

Existing columns in tables can be modified in a few respects, viz.

- The name of the column can be changed to another name not already used in the table
- The column can be “moved” to a different position in the left-to-right column order
- Conversions from non-character to character data are allowed, with some restrictions

Syntax `ALTER TABLE table`
 `ALTER [COLUMN] simple_column_name alteration`

```

alteration = new_col_name | new_col_type | new_col_pos
new_col_name = TO simple_column_name
new_col_type = TYPE datatype_or_domain
new_col_pos = POSITION integer

```



If you attempt to rename a column, you will bump into dependency problems if the column is referred to by a constraint or is used in a view, trigger or stored procedure.

Examples

Here we change the name of a column from EMP_NO to EMP_NUM:

```
ALTER TABLE EMPLOYEE
```

```
  ALTER COLUMN EMP_NO TO EMP_NUM; /* the keyword COLUMN is optional */
```

Next, the left-to-right position of the column—known as its degree—is moved:

```
ALTER TABLE EMPLOYEE
```

```
  ALTER COLUMN EMP_NUM POSITION 4;
```

This time, the data type of EMP_NUM is changed from INTEGER to VARCHAR(20):

```
ALTER TABLE EMPLOYEE
```

```
  ALTER COLUMN EMP_NUM TYPE VARCHAR(20);
```

Restrictions on altering data type

Firebird does not let you alter the data type of a column or domain in a way that might result in data loss.

- The new column definition must be able to accommodate the existing data. If, for example, the new data type has too few bytes or the datatype conversion is not supported, an error is returned and the change cannot proceed.
- When number types are converted to character types, each number type is subject to a minimum length in bytes, according to type. These are tabulated at the end of Chapter 8, in Table 8-4.
- Conversions from character data to non-character data are not allowed.
- Columns of BLOB and ARRAY types cannot be converted.

Attention

Any changes to the field definitions may require the indexes to be rebuilt.

Dropping columns

The owner of a table can use ALTER TABLE to drop (remove) a column definition and its data from a table. Dropping a column causes all data stored in it to be lost. The drop takes effect immediately unless another transaction is accessing the table. In this event, the other transaction continues uninterrupted and Firebird postpones the drop until the table is no longer in use.

Before attempting to drop a column, be aware of the dependencies that could prevent the operation from succeeding. It will fail if the column

- is part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint.
- is involved in a CHECK constraint—there may be table-level CHECK constraints on the column in addition to any imposed by its domain.
- is used in a view, trigger, or stored procedure.

Dependencies must be removed before the column drop can proceed. Columns involved in PRIMARY KEY and UNIQUE constraints cannot be dropped if they are referenced by FOREIGN KEY constraints. In this event, drop the FOREIGN KEY constraint before

dropping the PRIMARY KEY or UNIQUE key constraint and column it references. Finally, you can drop the column.

Syntax `ALTER TABLE name DROP colname [, DROP colname ...];`

For example, the following statement drops the column JOB_GRADE from the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE DROP JOB_GRADE;
```

To drop several columns with a single statement:

```
ALTER TABLE EMPLOYEE
  DROP JOB_GRADE,
  DROP FULL_NAME;
```

Dropping constraints

A correct sequence must be followed when dropping constraints, since both primary key and CHECK constraints are likely to have dependencies.



The `sql` command `SHOW TABLE <table-name>` lists out all the constraints and triggers for <table-name>. Alternatively, to find the names of constraints, it may be helpful to define and commit the four system views defined in the script `system_views.sql` provided in Appendix V.

UNIQUE KEY and PRIMARY KEY constraints

When a primary key or unique constraint is to be dropped, it will be necessary first to find and drop any foreign key constraint (FK) that references it. If it is a unique key, the FK declaration actually names the columns of the unique constraint. For example,

```
...
FK_DATA_ID FOREIGN KEY DATA_ID
  REFERENCES TEST_UQ(DATA_ID);
```

If the referenced key is the primary key, the name of the primary key column is optional in FK declarations and is often omitted. For example, looking at the `./examples/empbuild/employee.fdb` database:

```
...TABLE PROJECT (
  ...,
  TEAM_CONSTRT FOREIGN KEY (TEAM_LEADER)
  REFERENCES EMPLOYEE );
```

Dropping a foreign key constraint is usually straightforward:

```
ALTER TABLE PROJECT
  DROP CONSTRAINT TEAM_CONSTRT;
COMMIT;
```

After that, it becomes possible to drop the primary key constraint (PK) on the EMP_NO column of the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE
  DROP CONSTRAINT EMP_NO_CONSTRT;
```

CHECK constraints

Any CHECK conditions that were added during table definition can be removed without complications. CHECK conditions inherited from a domain are more problematical. To be free of the domain's constraints, it will be necessary to perform an `ALTER TABLE ALTER COLUMN ...TYPE` operation to change the column to another data type or domain.

Adding a column

One or more columns can be added to a table in a single statement, using the ADD clause. Each ADD clause includes a full column definition, which follows the same syntax as column definitions in CREATE TABLE. Multiple ADD clauses are separated with commas.

Syntax

```
ALTER TABLE table ADD <col_def>
<col_def> = col {<datatype> | [COMPUTED [BY] (<expr>) | domain}
[DEFAULT {literal | NULL | USER}]
[NOT NULL] [<col_constraint>]
[COLLATE collation]
<col_constraint> = [CONSTRAINT constraint] <constraint_def>
[<col_constraint>]
<constraint_def>=
PRIMARY KEY
| UNIQUE
| CHECK (<search_condition>)
| REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

The following statement adds a column, ALT_EMP_NO, to the EMPLOYEE table using the EMPNO domain:

```
ALTER TABLE EMPLOYEE
ADD EMP_NO EMPNO NOT NULL;
```

Example Here we add two columns, EMAIL_ID and LEAVE_STATUS to the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE
ADD EMAIL_ID VARCHAR(10) NOT NULL,
ADD LEAVE_STATUS DEFAULT 10 INTEGER NOT NULL;
```

Including integrity constraints

Integrity constraints can be included for columns that you add to the table. For example, a UNIQUE constraint could have been included for the EMAIL_ID column in the previous statement:

```
ALTER TABLE EMPLOYEE
ADD EMAIL_ID VARCHAR(10) NOT NULL,
ADD LEAVE_STATUS DEFAULT 10 INTEGER NOT NULL,
ADD CONSTRAINT UQ_EMAIL_ID UNIQUE(EMAIL_ID);
```

or

```
ALTER TABLE EMPLOYEE
ADD EMAIL_ID VARCHAR(10) NOT NULL UNIQUE,
ADD LEAVE_STATUS DEFAULT 10 INTEGER NOT NULL;
```

Adding new table constraints

The ADD CONSTRAINT clause can be included to add table-level constraints relating to new or existing columns.

Syntax

```
ALTER TABLE name ADD [CONSTRAINT constraint] <tconstraint_opt>;
```

where `<tconstraint_opt>` can be a PRIMARY KEY, FOREIGN KEY, UNIQUE, or CHECK constraint. The CONSTRAINT constraint phrase is omitted if you don't care to name the constraint yourself.

Example

```
ALTER TABLE EMPLOYEE
    ADD CONSTRAINT UQ_PHONE_EXT UNIQUE(PHONE_EXT);
```

When ALTER TABLE is not enough

Sometimes, you need to make a change to a column that can not be achieved with ALTER TABLE. Examples might be where you need to change a column that is storing international language in character set NONE to another character set, to correct your design error; or to redefine a telephone number, originally defined by someone as an integer, as an 18-character column.

In the first case, it isn't possible to change the character set of a column—so you need a workaround that both preserves the data and makes it available in the correct character set. In the other case, simply changing the data type of the telephone number column won't work if we already have existing integer data in the column. We want to keep the actual numbers but we have to convert them to strings. That cannot be done in the current structure, because an integer column cannot store a string.

The workaround entails creating a temporary column in your table, with the correct attributes, and “parking” the data there whilst you drop and recreate the original column.

Steps

- 1 Add a temporary column to the table, having a definition with the new attributes you need.

```
ALTER TABLE PERSONNEL
    ADD TEMP_COL VARCHAR(18);
COMMIT;
```

- 2 Copy the data from the column to be changed to the temporary column, massaging it appropriately, e.g. applying a character set “introducer” to convert the text data to the correct character set or, in our example, casting it appropriately.

```
UPDATE PERSONNEL
    SET TEMP_COL = CAST(TEL_NUMBER AS VARCHAR(18))
    WHERE TEL_NUMBER IS NOT NULL;
COMMIT;
```

- 3 After verifying that the data in the temporary column have been changed as planned, drop the old column.

```
ALTER TABLE PERSONNEL
    DROP TEL_NUMBER;
COMMIT;
```

Rename the temporary column to be the name of the column just dropped:

```
ALTER TABLE PERSONNEL
    ALTER TEMP_COL TO TEL_NUMBER;
COMMIT;
```

Removing (Dropping) a Table

Use `DROP TABLE` to remove an entire table permanently from the database. This is permanent and, once committed, cannot be undone.

```
DROP TABLE name;
```

The following statement drops the table `PERSONNEL`:

```
DROP TABLE PERSONNEL;
```

The RECREATE TABLE statement

Sometimes, you may want to drop a table and create it again “from scratch”. For these occasions, Firebird has `RECREATE TABLE`, which does the following:

- drops the existing table and all of the objects belonging to it
- commits the change
- creates the new table as specified in the clauses and sub-clauses



Data and ancillary objects such as constraints, indexes and triggers are not preserved. Make sure that you save the source of any of these ancillary definitions that you want to keep before submitting a `RECREATE TABLE` request!

Syntax

The syntax is identical to that of `CREATE TABLE`. Simply substitute the `CREATE` keyword with `RECREATE` and proceed.

Restrictions and recommendations

If the table is in use when you submit the `DROP` or `RECREATE` statement, the request will be denied, with an “Object xxxxx is in use” message.

Always take a backup before any activity that changes metadata.

Although it is possible to make metadata changes when users are on-line, it is not recommended, especially for radical changes like dropping and recreating tables. If necessary, force users off and get exclusive access. Instructions for getting exclusive access are in Chapter 35, in the topic [The *gfx* Tool Set](#).

Temporary Tables

Firebird databases of on-disk structure (ODS) 11.1 and higher can store persistent definitions of tables that applications can use to store sets of temporary data during a run-time session. Instances of these temporary structures can be created by many session-users simultaneously—hence, they are known as *global temporary tables*, commonly abbreviated to “GTT”.

Server versions lower than v.2.1 do not support GTTs. It is also not possible to create and use GTTs under a newer server if the ODS of the database is 11.0 or lower. Later, in the topic [Temporary Storage for Older Versions](#), we look at the recommended way to emulate some of the features of GTTs for these lower versions.

Global Temporary Tables

The same structural features that you can apply to regular tables (indexes, triggers, field-level and table level constraints) can be applied to a global temporary table. However, a GTT cannot be linked in a referential relationship to a regular table nor to another GTT that has a different lifespan (see below), nor be referred to in a constraint or domain definition.

For obvious reasons, the EXTERNAL TABLE option is not possible for a GTT definition.

Each instance of a GTT gets its own “private” pages for data and indexes, laid out just like those for a regular table but inaccessible to any other instance. At the end of the lifetime of the GTT instance, those pages are released for immediate re-use, in much the same way as the data and index pages for regular tables are released when the table is dropped.



The private files containing the GTT instances are not subject to the Forced Writes setting of the database. They are always opened for asynchronous writes, i.e., Forced Writes is always OFF for GTT instances.

Lifespan of GTT Sets

An instance of a GTT—a set of data rows created by and visible within the given connection or transaction—is created when the GTT is referenced for the first time, usually at statement prepare time. The lifespan of a GTT “travels” with its definition and cannot be overridden, so make sure you define the right one for your requirements.

When defining a GTT, the options for specifying the lifespan of an instance are dictated by what you specify to happen ON COMMIT of the parent transaction:

- PRESERVE ROWS retains the set until the client session ends
- DELETE ROWS destroys the set when the parent transaction is committed or rolled back. Transaction lifespan is the default.



You can check the lifespan type of a GTT by querying the system table RDB\$RELATIONS and looking at the RDB\$RELATION_TYPE for the GTT. A value of 4 indicates a session-wide lifespan and 5 a transaction-level lifespan.

CREATE Syntax for a GTT

The pattern for the CREATE GLOBAL TEMPORARY TABLE statement is the same as for CREATE TABLE, with the addition of the lifespan clause and the exclusion of the EXTERNAL TABLE option, viz.,

```
CREATE GLOBAL TEMPORARY TABLE <table-identifier>(
    <column-definitions> [,
    <table-constraint-definitions>])
[ON COMMIT DELETE | PRESERVE ROWS]
```



Examples

Because transaction-level lifespan is the default, the ON COMMIT DELETE clause may be omitted if you want to specify this level.

The first example defines a GTT named GTT_S_01 whose instances will remain alive for the duration of the client session:

```
CREATE GLOBAL TEMPORARY TABLE GTT_S_01 (
    ID BIGINT NOT NULL,
    DESCRIPTION VARCHAR(100),
```



```

UPDATE_TIME TIMESTAMP DEFAULT current_timestamp,
CONSTRAINT PK_GTT_S_01 PRIMARY KEY (ID)
)
ON COMMIT PRESERVE ROWS;
/* Remember to commit the DDL */
COMMIT;

```

Another example defines a GTT whose instance will be destroyed when the transaction in which it was invoked completes. It has a foreign key referencing the primary key of the GTT in our first example:

```

CREATE GLOBAL TEMPORARY TABLE GTT_T_02 (
    ID BIGINT NOT NULL PRIMARY KEY,
    DETAILS VARCHAR(200),
    UPDATE_TIME TIMESTAMP DEFAULT current_timestamp,
    MASTER_ID BIGINT NOT NULL
    REFERENCES GTT_S_01 (ID),
    CONSTRAINT PK_GTT_S_01 PRIMARY KEY (ID)
); -- ON COMMIT DELETE omitted because it is the default lifespan
/* Remember to commit the DDL */
COMMIT;

```



When the GTT instances are destroyed, any DELETE triggers that are defined for the GTT will not fire.

Temporary Storage for Older Versions

While Firebird does not prevent end-user applications from executing DDL statements, such as CREATE TABLE and DROP TABLE, it is not recommended to write applications that do such things in the course of normal business. For versions prior to v.2.1 or databases of ODS 11.0 or lower, a popular model for storing temporary data is the “permanent temporary table”.

The idea is to define a regular table structure in the permanent metadata that end-user applications will write to and read from in the course of a session or transaction. A batch identifier is included, written from a generator (sequence) for a session-wide lifespan or, for a transaction-wide lifespan, the CURRENT_TRANSACTION value.

Applications can insert, reprocess and delete rows in such a table during the course of a task—remember, Firebird does not put locks on tables in the normal course of events.

According to the conditions and needs of the application, it can itself be responsible for deleting the temporary rows when it is finished with the batch of rows it created, using the batch ID for a searched delete. Alternatively, the application could post a row containing the batch ID to a housekeeping table signalling “clean-up required” to a later, deferred operation that runs after hours, before a backup.



If you have the option of using GTTs for temporary storage, then do so. The “permanent temporary table” solution has no way to truly isolate the temporary data or to effect the garbage-free cleanup that GTTs provide.

Tree Structures

Table structures designed to store multi-layered hierarchical lists of nodes for “trees”—such as family trees, menus, classification systems and many others—are at cross-purposes with the design ideals of SQL databases, specifically normalization. We solve the problem by rigorously abstracting the inter-nodal relationship so that we store each node with its own unique identifier, along with a foreign key referencing the unique identifier of its immediate parent node. The “flattening” for use by applications requiring hierarchical sets is performed at run-time by data manipulation language (DML).

At the DDL level, the style of table to create—where every node of the notional tree structure has one and only one row and exactly one or no parent—is known as a “self-referencing table”. The rationale and details for designing such tables can be found in Chapter 17, **Referential Integrity**, in the topic *Self-Referencing Relationship*.

The DML side of the issue, tracking recursively through the nodes and building the flattened output, can be implemented using common table expressions (v.2.1 and above) or recursive stored procedures that return sets (“selectable stored procedures”).

Refer to the topic *Common Table Expressions* in Chapter 23, **Views and Other Run-time Sets**, and to *Recursive Procedures* in Chapter 29, **Stored Procedures**.

CHAPTER 16

INDEXES

Indexes—sometimes pluralized as *indices*—are table attributes that can be placed on a column or a group of columns to accelerate the retrieval of rows.

An index serves as a logical pointer to the physical locations (addresses) of rows in a table, much as you search an index in a book to quickly locate the page numbers of topics that you want to read. In most cases, if the engine can fetch the requested rows directly by scanning an index instead of scanning all rows in the table, requests will complete faster.

A well-designed system of indexes plays a vital role in the tuning and optimization of your entire application environment. However, creating indexes should be considered largely as a tuning exercise that you will engage in once you have finalised the database structure and have a reasonable set of test data to play with.

Limits

Firebird allows up to 256 user-created indexes per table (although, in v.1.0.x, the limit is just 64). These are theoretical limits that are governed by both page size and the actual on-disk size of the index description data on the index root page. You could not store 256 indexes for a table in a database with a page size smaller than 16 Kb. On the index root page, each index needs 31 bytes for its identifier, space for descriptions of each segment (column) involved in the index and some bytes to store a pointer to the first page of the index. Even a 16 Kb page may not be able to accommodate 256 indexes if there are more than a few compound indexes in the table.

The total size (width) of an index cannot exceed one-quarter of the page size. In reality, the number of bytes may be significantly smaller than that figure. Factors that can reduce the number of actual “slots” available to store characters include:

- character sets that use multiple bytes per character
- international character sets with complex upper/lower case pairings and/or dictionary sorting rules

- use of non-binary collations
- multiple segments (composite indexes) which require the addition of padding bytes to retain the geometry of the index

In other words, using any character set except NONE will influence your decisions about index design—particularly whether to use composite indexes. That’s the bad news. The good news is that Firebird makes good use of single-column indexes in multi-column searches and sorts, reducing the need for multi-column indexes that you might have considered essential from your experience with another DBMS.

Automatic vs User-defined Indexes

Firebird creates indexes to enforce various integrity constraints automatically—for more information, refer to Chapter 15, **Tables** and Chapter 17, **Referential Integrity**. To delete these indexes, it is necessary to drop the constraints that use them.

Use of the constraint indexes is not restricted to their work supporting the integrity of keys and relationships. They are considered, along with all others, when queries are prepared.

When defining your own indexes, it is of utmost importance to avoid creating any index that duplicates an automatically-generated one. It puts the optimizer (see below) in the unhappy position of have to choose between equals. In many cases, it will solve the problem by not choosing either of them.

Importing legacy indexes

Do not import the “primary indexes” along with the tables you import from a migrating DBMS. There are two important reasons to abandon these indexes:

- Many legacy systems use hierarchical index structures to implement referential integrity. SQL databases do not use this logic to implement RI and these indexes usually interfere with Firebird's optimizer logic.
- Firebird creates its own indexes to support primary and foreign key constraints, regardless of any existing index. As noted above, duplicate indexes cause problems for the optimizer and should be avoided completely.

Directional indexes

The sort direction of indexes in Firebird is important. It is a mistake to assume that the same index can be used to sort or search “both ways”, that, is lowest-to-highest and highest-to-lowest. As a rule of thumb, ASC (ascending) indexes will help searches where relatively low values are sought, whereas DESC (descending) indexes will help for maximum or high values .

If an automatic index is ASC (the default) there will be no problems if you need to define a DESC index using the same column(s). The reverse is also true: you can choose to have the automatic indexes for keys created in descending order. The optimizer will not be upset if you also create an ascending one on the same columns.

Query plans

Before a query is executed, a set of preparation routines—known as the optimizer—begins analyzing the columns and operations in the request, to calculate the fastest way to respond. It starts by looking for indexes on the tables and columns involved. Working its way through a sequence of cost-based decision paths, it develops a plan—a kind of “roadmap” for the route it will follow when it actually executes the query. The final plan it chooses reflects the cheapest route it can predict according to the indexes it can use.

The optimizer's plan can be viewed in an *isql* shell session in two ways:

- 1 By default, *isql* does not display the plan. Use `SET PLAN ON` to have the plan displayed at the top of the output from a `SELECT` query.
- 2 Use `SET PLANONLY` to submit queries and view the plans without actually running the queries. This enables you to inspect the plan for any query, not just `SELECT` queries.

For details about using *isql*, refer to Chapter 24.

It is possible to override the optimizer's plan with one of your own, by including a `PLAN` clause in your query statement. Most third-party GUI tools provide the ability to view the plan, with or without running the query, and to override it.



Plans that seem to work faster on small, stable test sets will not necessarily satisfy on large, dynamic sets. Don't override the optimizer's plan unless you have tested your own and found it to be consistently faster on realistic data in well-simulated production conditions.

More about Query Plans

Query plans—particularly those generated by Firebird's optimizer engine—are discussed in greater detail in Chapter 19, *DML Queries*, in the topic [*Query Plans and the Optimizer*](#).

How Indexes Can Help

If the optimizer decides to use an index, it searches the pages of the index to find the key values required and follows the pointers to locate the indicated rows in the table data pages. Data retrieval is fast because the values in the index are ordered. This allows the system to locate wanted values directly, by pointer, and avoid visiting unwanted rows at all. Using an index typically requires fewer page fetches than “walking through” every row in the table. The index is small, relative to the row size of the table, and, provided the index design is good, it occupies relatively fewer database pages than the row data.

Sorting and grouping

When the columns specified in an `ORDER BY` or `GROUP BY` clause are indexed, the optimizer can sequence the output by navigating the index(es) and assemble the ordered sets more directly than it can with non-indexed reads.

A `DESCENDING` index on the aggregated column can speed up the query for the aggregating function `MAX(..)`, since the row with the maximum value will be the only one it visits.

For more information about using function expressions in queries, refer to Chapters 20, *Expressions and Predicates* and 23, *Ordered and Aggregated Sets*.

Joins

For joins, the optimizer goes through a process of merging streams of data by matching the values specified in the implicit or explicit “ON” criteria. If an index is available for the column or columns on one side of the join, the optimizer builds its initial stream by using the index to match the join key with its correspondent from the table on the other side of the join. Without an index on either side, it must generate a map of the locations of the eligible rows of one side first and navigate that, in order to make the selections from the table on the other side of the join.

Comparisons

When an indexed column is evaluated to determine whether it is greater than, equal to or less than a constant, the value of the index is used for the comparison and non-matching rows are not fetched. Without an index, all of the candidate rows have to be fetched and compared in turn.

What to Index

The length of time it takes to search a whole table is directly proportional to the number of rows in the table. An index on a column can mean the difference between an immediate response to a query and a long wait. So— why not index every column?

The main drawbacks are that indexes consume additional disk space, and inserting, deleting, and updating rows takes longer on indexed columns than on non-indexed columns. The index must be updated each time a data item in the indexed column changes and each time a row is added to or deleted from the table.

Nevertheless, the boost in performance for data retrieval usually outweighs the overhead of maintaining a conservative but useful collection of indexes. You should create an index on a column when:

- Search conditions frequently reference the column. An index will help in date and numeric searches when direct comparisons or BETWEEN evaluations are wanted. Search indexes for character columns are useful when strings are to be evaluated for exact matches or against STARTING WITH and CONTAINING predicates. They are not useful with the LIKE predicate.
- The column does not carry an integrity constraint but is referenced frequently as a JOIN condition
- ORDER BY clauses frequently use the column to sort data.

When sets must be ordered on multiple columns, composite indexes which reflect the output order specified in ORDER BY clauses can sometimes improve retrieval speed. Often, however, single-key indexes on each column that might be involved in sorting give better performance and thus are usually first choice on the test bed. Composite indexes should be tested thoroughly in a production simulation to verify whether they actually improve the sort.

- You need an index with special characteristics not provided by data or existing indexes, such as a non-binary collation or an ascending or descending direction
- Aggregations are to be performed on large sets. Single-column or suitably ordered complex indexes can improve the speed with which aggregations are formed in complex GROUP BY clauses

You should not create indexes for columns that

- are seldom referenced in search conditions
- are frequently updated non-key values, such as timestamps or user signatures
- have a small number of possible or actual values spread over a wide campus of rows
- are styled as two-phase (True/False) or three-phase (True/False/unknown) Boolean

When to index

Some indexes will suggest themselves to you during the initial design period—typically for requirements that you know will need particular sortings, groupings and evaluations. For the rest, it is very good practice to be conservative about creating indexes until it becomes clear that they might be beneficial. It is a rewarding strategy to defer the creation of doubtful indexes until after a point in development where you have good samplings of typical test data and an awareness of which operations are too slow.

The benefits of deferred index design include

- reduction of “performance fogging” that can interfere with the functional integrity testing
- faster identification of the real sources of bottlenecks
- avoidance of the overhead of maintaining unnecessary or inefficient indexes

Using CREATE INDEX

The DDL statement CREATE INDEX creates an index on one or more columns of a table. A single-column index enables indexed searching on only one column in response to a query, whereas multi-column index searches may facilitate searching one or more columns.

Syntax `CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX <index-name>
ON <table-name> (col [, col ...] | COMPUTED BY (<expr>);`

Mandatory elements

CREATE INDEX <index-name>

—names the index. The identifier must be distinct from all other object identifiers in the database, apart from constraint and column identifiers. It is a good idea to use a systematic naming scheme, however, as an aid to self-documentation.



When designing your naming scheme, keep in mind that the supporting indexes automatically created by Firebird for integrity constraints can be given custom names and thus be integrated with the overall naming scheme.

<table-name>

—the name of the table to which the index applies

col [, col...]

—column name, or comma-separated list naming the columns which are to be the index keys. Column order is significant in indexes. For more information, see the topic below, [*Multi-column Indexes*](#).

Example The following declaration, creates a non-unique, ascending index on a personal name column in a PERSON table. It may aid search conditions like WHERE LAST_NAME = 'Johnston' or WHERE LAST_NAME STARTING WITH 'Johns':

```
CREATE INDEX LAST_NAME_X ON PERSON(LAST_NAME);
```

Optional elements

UNIQUE

The UNIQUE keyword can be used on indexes for which you want to disallow duplicate entries. The column or group is checked for duplicate values when the index is created and for existing values each time a row is inserted or updated.

Unique indexes make sense only when you need to enforce uniqueness as an intrinsic characteristic of the data item or group. For example, you would not define a unique index on a column storing a person's name, because personal names are not intrinsically unique. Conversely, a unique index is a good idea on a column containing a social security number, since a unique key violation on it would alert the user to an error that needed attention.

Example In this example, a unique index is created on three columns of an inventory table to ensure that the system stores at most one row for each size and color of an item:

```
CREATE UNIQUE INDEX STK_SIZE_COLOR_UQX
ON STOCK_ITEM (PRODUCT_ID, SIZE, COLOR);
```

Note that a unique index is not a key. If you require a unique key for referential purposes, apply a UNIQUE constraint to the column(s) instead. Refer to the topic [*The UNIQUE constraint*](#) in the previous chapter.

Finding duplicates

Of course, it won't be possible to create a unique index on a column that already contains duplicate values. Before defining a unique index, use a SELECT statement to find duplicate items in the table. For example, before putting a unique index on PRODUCT_NAME in this PRODUCT table, this check would reveal any duplicates in the column:

```
SELECT
    PRODUCT_ID,
    UPPER(PRODUCT_NAME)
FROM PRODUCT
GROUP BY PRODUCT_ID, UPPER(PRODUCT_NAME)
HAVING COUNT(*) > 1;
```




Upper-casing the column to make the search case-insensitive is not necessary from the point of view of data uniqueness. Still, if uniqueness has been “broken” by faulty data entry, we would want to find the offending records.

How you deal with duplicates depends on what they signify, according to your business rules, and the number of duplicates needing to be eliminated. Usually, a stored procedure will be the most efficient way to handle it. Stored procedures are discussed in detail in Chapters 28 and 29.

ASC[ENDING] or DESC[ENDING]

The keywords ASC[ENDING] and DESC[ENDING] determine the vertical sort order of the index. ASC specifies an index that sorts the values from lowest to highest. It is the default and can be omitted. DESC sorts the values from highest to lowest and must be specified if a descending index is wanted. A descending index may prove useful for queries that are likely to search for high values (oldest age, most recent, biggest, etc.) and for any ordered searches or outputs that will specify a sort order that is “highest first”, ranked or in “newest-to-oldest” date order.

Example

The following definition creates a descending index on a table in the employee database:

```
CREATE DESCENDING INDEX DESC_X
ON SALARY_HISTORY (CHANGE_DATE);
```

The optimizer will use this index in a query such as the following, which returns the employee numbers and salaries of the 10 employees who most recently had a raise:

```
SELECT FIRST 10 EMP_NO, NEW_SALARY
FROM SALARY_HISTORY
ORDER BY CHANGE_DATE DESCENDING;
```

If you intend to use both ascending and descending sort orders on a particular column, define both an ascending and a descending index for the same column. For example, it will be fine to create the following index in addition to the one in the previous example:

```
CREATE INDEX ASCEND_X
ON SALARY_HISTORY (CHANGE_DATE);
```

COMPUTED BY <expr>

From the “2” series forward, an index can be created using arbitrary expressions applied to columns in the table structure. The keywords COMPUTED BY preface the expression that is to define the index nodes. The index created is known as an *expression index*. Expression indexes have exactly the same features and limitations as regular indexes, except that, by definition, they cannot be composed of multiple segments.



In satisfying DML requests, the engine will use the expression index if the expression used in the search predicate matches exactly the expression used in the index declaration. The expression index will never be chosen otherwise.

Examples

This index provides the upper case value of a column, that would typically be useful for a case-insensitive search:

```
CREATE INDEX IDX1 ON T1
COMPUTED BY ( UPPER(COL1 COLLATE PT_BR) );
COMMIT;
/**/

SELECT * FROM T1
WHERE UPPER(COL1 COLLATE PT_BR) = 'ÔÛÀÀ'
-- PLAN (T1 INDEX (IDX1))
```

```
Indexing on a more complex expression:
CREATE INDEX IDX2 ON T2
  COMPUTED BY ( EXTRACT(YEAR FROM COL2) || EXTRACT(MONTH FROM COL2) );
COMMIT;
/**/
SELECT * FROM T2
  ORDER BY EXTRACT(YEAR FROM COL2) || EXTRACT(MONTH FROM COL2)
-- PLAN (T2 ORDER IDX2)
```

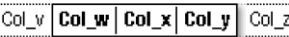
Multi-column Indexes

If your applications frequently need to search, order or group on the same group of multiple columns in a particular table, it might be of benefit to create a multi-column index (also known as a composite or complex index).

The optimizer is likely to use a subset of the segments of a multi-column index to optimize a query if the left-to-right order in which the query accesses the columns in an ORDER BY clause matches the left-to-right order of the column list defined in the index. However, queries do not need to be constructed with the exact column list that is defined in the index in order for it to be available to the optimizer. The index can also be used if the subset of columns used in the ORDER BY clause begins with the first columns in the multi-column index, *in the same order*.

Firebird can use a single element (or *segment*) of composite index to optimize a search if all of the elements to the left of that element are also used in the matching order. Consider a segmented index on three columns, Col_w, Col_x and Col_y, in that order:

Figure 16.1 Using a segmented index



The index would be picked by the optimizer for this query:

```
SELECT <list of columns> FROM ATABLE
ORDER BY COL_w, COL_x;
```

It would not be picked for either of these queries:

```
SELECT <list of columns> FROM ATABLE
ORDER BY COL_x, COL_y;
```

```
/**/
SELECT <list of columns> FROM ATABLE
ORDER BY COL_x, COL_w;
```

OR predicates in queries

If you expect to issue frequent queries against a table where the queries use the OR operator, it is better to create a single-column index for each condition. Since multi-column indexes are sorted hierarchically, a query that is looking for any one of two or more conditions must search the whole table, losing the advantage of an index.

For example, suppose the search requested

...

```
WHERE A > 10000 OR B < 300 OR C BETWEEN 40 AND 80
...
```

an index on (A,B,C) would be used to find rows containing eligible values of A but it could not be used for searching for B or C values because the OR operator works by testing each condition, beginning at the leftmost predication. The operation stops as soon as a condition is found to be true. By contrast, individual indexes on A, B and C would all be used. The direction of an index could be important here, too. For A, a descending index would be more useful than an ascending one if the search value is at the high end of range of stored values.

Search criteria

The same rules that apply to the ORDER BY clause also apply to queries containing a WHERE clause. The next example creates a multi-column index for the PROJECT table in employee.fdb:

```
CREATE UNIQUE INDEX PRODTYPEX
ON PROJECT (PRODUCT, PROJ_NAME);
```

The optimizer will pick the PRODTYPEX index for this query, because the WHERE clause refers to the first segment of the index:

```
SELECT * FROM PROJECT
WHERE PRODUCT = 'software';
```

Conversely, it will ignore that index for the next query, because PROJ_NAME is not the first segment:

```
SELECT * FROM PROJECT
WHERE PROJ_NAME STARTING WITH 'Firebird 1';
```

Inspecting Indexes

To inspect indexes defined in the current database, use the *isql* command SHOW INDEX:

- To see all indexes defined for a specific table, use the command
SHOW INDEX tablename
- To view information about a specific index, use
SHOW INDEX indexname

Tip

You can examine a great deal of useful information about populated indexes using the *gstat* tool. For details, refer to the section [Collecting Database Statistics—gstat](#) in Chapter 38, *Monitoring and Logging Features*.

Making an Index Inactive

The ALTER INDEX statement is used to switch the state of an index from active to inactive and vice versa. It can be used to switch off indexing before inserting or updating a large batch of rows, to avoid the overhead of maintaining the indexes during the long operation. After the operation, indexing can be reactivated and the indexes will be rebuilt.

Housekeeping

The other use for ALTER INDEX {ACTIVE | INACTIVE} is a housekeeping one. The distribution of values changes, gradually under normal conditions and, under some operating conditions, more frequently. The binary tree structures in which indexes are maintained are stored on index pages. As indexes grow and change, the nodes become progressively more spread across many linked pages, reducing the overall effectiveness of the index. Switching an index from active to inactive and back to active rebuilds it, equalising the distribution of nodes from page to page and, often, reducing the level of indirection, i.e., the depth of the “tree” of pages that have to be traversed when reading the nodes into the database cache. Every little bit helps when troubleshooting index performance problems.

Syntax

ALTER INDEX index-name INACTIVE | ACTIVE ;



ALTER INDEX cannot be used on any index that was auto-created to support a constraint.

For more information about ways to optimize the benefits of index use by Firebird’s optimizer and to keep your indexes in good shape for continued efficiency, refer to the topic [Optimal Indexing](#) in Chapter 19.

‘Index is in use’ error

An index that is being used in a transaction cannot be altered or dropped until the transaction has finished using it. Attempts will have different results, according to the lock setting of the active transaction—

- In a WAIT transaction, the ALTER INDEX operation waits until the transaction completes
- In a NO WAIT transaction, Firebird returns an error.

For more information about transaction lock settings, refer to the topic [Locking Policy](#) in Chapter 26, *Configuring Transactions*.

Altering the structure of an index

Unlike many ALTER statements, the ALTER INDEX syntax cannot be used to alter the structure or attributes of the object. You can either drop the index and define it afresh, using CREATE INDEX or you can create another index to use as an alternative.

Dropping an Index

The DROP INDEX statement removes a user-defined index from the database. No user can drop an index except the user who created it, or SYSDBA or another user with root/Administrator privileges.

Use DROP INDEX also when an index needs to have a change of structure: segments added, removed or reordered, sort order altered. First use a DROP INDEX statement to delete the index, then use a CREATE INDEX statement to recreate it, using the same name and the new characteristics.

Syntax `DROP INDEX name;`

The following statement deletes an index from the JOB table in `employee.fdb`:

`DROP INDEX MINSALX;`

CHAPTER

17

REFERENTIAL INTEGRITY

The term referential integrity refers to the capability of a database to protect itself from receiving input that would result in an inconsistent relationship. Specifically, the referential integrity of a database exists according to its ability to enforce and protect a relationship between two tables.

Implementing formal referential constraints adds some extra work to the database designer's task, so—what is the payback? If you are new to the concept, then you are sure to find many reasons of your own to justify the additional time and attention, including the following:

BOMB-PROOFING Formal referential constraints—especially when used intelligently with other types of constraint—will bomb-proof the business rules of your databases against application bugs, regardless of their source. This becomes especially important when you deploy your systems to sites where unskilled or partly skilled staff have access to the database through third-party utilities.

QUERY SPEED Indexes automatically created for referential integrity constraints speed up join operations.

QUALITY CONTROL During development and testing, potential bugs tend to show up early because the database rejects operations that break the rules. Effectively, they eliminate the grief from proceeding with application development under false assumptions about data consistency.

SELF-DOCUMENTATION Integrity rules enforced by your database provide “free” documentation that eliminates the need for any descriptive documents apart from your schema scripts. The rules defined in the metadata correctly become the authoritative reference to the data model for new staff and future development.

Terminology

When a RDBMS provides the ability to declare the relationship between two tables, it is sometimes termed declarative referential integrity, a fuzzy term which seems to have been propagated by writers of magazine articles. Referential integrity is a design objective, a

quality; the author prefers the term formal referential constraints when referring to the mechanisms for implementing the rules.

In a relational database management system—RDBMS—relationships between two tables are created by means of the foreign key constraint. The foreign key constraint enforces the rules of existence for the rows it represents, protecting the table against attempts to store rows that are inconsistent with the data model. However, this constraint does not need to work alone. Other integrity constraints—described in detail in the Chapter 15, **Tables**—can work in combination with the referential constraint to protect the consistency of relationships.

The FOREIGN KEY Constraint

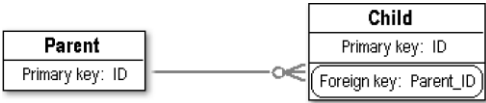
A foreign key is a column or set of columns in one table that corresponds in exact order to a column or set of columns defined as a PRIMARY KEY or as a UNIQUE constraint in another table. In its simplest form, it implements an optional *one-to-many* relationship.



An optional relationship exists when the relationship is made possible by the formal structure but is not required. That is to say, a parent instance may exist without any referencing child but, if both exist, both are constrained. The other side of the coin is a mandatory relationship. Mandatory relationships are discussed later in this chapter.

The standard entity-relationship model depicts a simple one-to-many relationship between two entities like this:

Figure 17.1 Entity-relationship model



If we implement this model as two tables—PARENT and CHILD—then rows in the CHILD table are dependent on the existence of a linking row in PARENT. Firebird's FOREIGN KEY (FK) constraint enforces this relationship in the following ways:

- requires that the value presented in the FK column of the referencing table, CHILD (CHILD.PARENT_ID) must be able to be linked to a matching value present in the referenced unique key—in this case, the primary key—of PARENT (PARENT.ID)
- by default, disallows a row in PARENT to be deleted, or to have its linking unique key value changed to a different value, if dependent CHILD rows exist.
- must implement the relationship that was intended at the time the reference was created, or the last time it was updated
- by default, allows the FK column to be null. Since it is impossible to link null to anything, such child rows are orphans—they have no parent.

Implementing the constraint

In order to implement the referential constraint, certain prerequisites must be attended to. In this topic, we follow through a very simple example. If you are developing in an existing, complex environment, where SQL privileges are in effect, then you may need to

be concerned about the REFERENCE privilege. It is introduced in a separate topic later in this chapter.

The parent structure

It is necessary to start with the parent table and implement a controlling unique key to which the dependent table will link. This is commonly the primary key of the parent table, although it need not be. A foreign key can link to a column or group which has been constrained using the UNIQUE constraint. For present purposes, we will use the primary key:

```
CREATE TABLE PARENT (
    ID BIGINT NOT NULL,
    DATA VARCHAR(20),
    CONSTRAINT PK_PARENT PRIMARY KEY(ID));
COMMIT;
```

The child structure

In the child structure, we need to include a column—PARENT_ID—that exactly matches the primary key of the parent in type and size (and also column order, if the linkage involves multiple columns):

```
CREATE TABLE CHILD (
    ID BIGINT NOT NULL,
    CHILD_DATA VARCHAR(20),
    PARENT_ID BIGINT,
    CONSTRAINT PK_CHILD PRIMARY KEY(ID));
COMMIT;
```

The next thing to do is declare the relationship between the child and the parent by means of a FOREIGN KEY constraint.

Syntax for defining a FOREIGN KEY

```
...
FOREIGN KEY (column [, col ...])
REFERENCES (parent-table [, col ...])
[USING [ASC | DESC] INDEX index-name] /* v.1.5 and above */
[ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
[ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
```

Defining our foreign key:

```
ALTER TABLE CHILD
ADD CONSTRAINT FK_CHILD_PARENT
    FOREIGN KEY(PARENT_ID)
    REFERENCES PARENT(ID); /* REFERENCES PARENT is also valid,
    because ID is the primary key of PARENT */
```

Firebird stores the constraint FK_CHILD_PARENT and creates a non-unique index on the column(s) named in the FOREIGN KEY argument. The index will be named FK_CHILD_PARENT as well, unless you used the optional USING clause to assign a different name to the index.

V.1.0.x In Firebird 1.0.x, the index name will be INTEG_nn (where nn is a number).



If you specified a DESCENDING index for the referenced primary or unique constraint, you must be sure to specify USING DESC INDEX for any foreign keys that reference it.

The two tables are now engaged in a formal *referential integrity constraint*. We can add new rows to PARENT without restriction:

```
INSERT INTO PARENT (ID, DATA)
VALUES (1, 'Parent No. 1');
```

However, there are restrictions on CHILD. We can do this:

```
INSERT INTO CHILD (ID, CHILD_DATA)
VALUES (1, 'Child No. 1');
```

Because the nullable column PARENT_ID was omitted from the column list, NULL is stored there. This is allowed under the default integrity rules. The row is an *orphan*.

However, we get a constraint error if we try to do this:

```
INSERT INTO CHILD (ID, CHILD_DATA, PARENT_ID)
VALUES (2, 'Child No. 2', 2);
```

```
ISC ERROR CODE:335544466
```

```
ISC ERROR MESSAGE:
```

```
violation of FOREIGN KEY constraint "FK_CHILD_PARENT" on table "CHILD"
```

There is no row in PARENT having a PK value of 2, so the constraint disallows the insert.

Both of the following are allowed:

```
UPDATE CHILD
SET PARENT_ID = 1
WHERE ID = 1;
COMMIT;
/* */
INSERT INTO CHILD (ID, CHILD_DATA, PARENT_ID)
VALUES (2, 'Child No.2', 1);
COMMIT;
```

Now, the PARENT row with ID=1 has two child rows. This is the classic master-detail structure—an uncomplicated implementation of the one-to-many relationship. To protect the integrity of the relationship, the default rules will disallow this because the parent has children:

```
DELETE FROM PARENT WHERE ID = 1;
```

Action triggers to vary integrity rules

Obviously, integrity rules take effect whenever some change in data occurs that affects the relationship. However, the default rules don't suit every requirement. We may want to override the rule that permits child rows to be created as orphan, or to be made orphans by having their foreign key set to null in an operation. If it is a problem for our business rules that a parent row cannot be deleted if it has dependent child rows, we may want Firebird to take care of the problem automatically.

Firebird's SQL language can oblige, through its optional automatic action triggers:

[ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
[ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]

Automatic action triggers

Firebird provides the optional standard DML events ON UPDATE and ON DELETE, along with a range of action options to vary the referential integrity rules. Together, the DML event and the automatic behavior specified by the action option form the action trigger—the action to be taken in this dependent table when modifying or deleting the referenced key in the parent. The actions defined include *cascading* the change to associated foreign table(s).

Action trigger semantics

NO ACTION

Because this is the default action trigger, the keyword can be—and usually is—omitted . The DML operation on the parent's PK leaves the foreign key unchanged and potentially blocks the operation on the parent.

ON UPDATE CASCADE

The cascade action directs that any changes in the parent key be passed down—cascaded—to the children. In the dependent table, the foreign key corresponding to the old value of the primary is updated to the new value of the primary key.

ON DELETE CASCADE

In the dependent table, the row with the corresponding key is deleted.

SET NULL

The foreign key corresponding to the old parent PK is set to NULL—the dependent rows become orphans. Clearly, this action trigger cannot be applied if the foreign key column is non-nullable.

SET DEFAULT

The foreign key corresponding to the old parent PK is set to its default value. There are some “gotchas” about the SET DEFAULT action that it is important to know about:

- the default value is the one that was in effect at the time the foreign key constraint was defined. If the column's default changes later, the original default for the FK's SET DEFAULT action does not follow the new default—it remains as the original.
- if no default was ever declared explicitly for the column, then its default is implicitly NULL. In this case, the SET DEFAULT behavior will be the same as SET NULL.
- if the default value for the foreign key column is a value that has no corresponding PK value in the parent, then the action trigger will cause a constraint violation.

Interaction of Constraints

By combining the formal referential constraint with other integrity constraints (see Chapter 15, *Tables*) it is possible to implement most, if not all, of your business rules with a high degree of precision. For example, a NOT NULL column constraint will restrict the action options and prevent orphan rows, if required; whereas a foreign key column that is nullable can be used to implement special data structures such as trees (see below).

If you need to make a column of your foreign key non-nullable, then create a “dummy” parent row with an unused key value, such as 0 or -1. Use the SET DEFAULT action to emulate the SET NULL behavior by making the column default to the dummy key value.

Referential constraints can be assisted by CHECK constraints. In some cases, a CHECK constraint inherited from a domain could interfere or conflict with a referential constraint, too. It is worth spending a few minutes sketching out the effects of each constraint on paper or a whiteboard, to identify and eliminate any potential problems.

Custom Action Triggers

It is perfectly possible to write your own action triggers to customize or extend referential behavior. Although the automatic triggers are flexible enough to cover most requirements, there is one special case where custom triggers are generally called for. This is the case where creation of the mandatory enforcing index on the foreign key column is undesirable because the index would be of very low selectivity.

Broadly, indexes of low selectivity occur where a small number of possible values is spread over a large table, or where only a few possible values are ever used in the actual table data. The resulting massive duplication of values in the index—described as “long chains”—can impact query performance severely as the table grows.



Index selectivity is discussed in some detail in Chapters 19 and 21. If the topic is new to you, you are urged to digest it thoroughly before deciding to use formal integrity constraints to implement every single one-to-many relationship in your data model “just because you can”.

When writing custom referential triggers, you must make sure that your own triggers will preserve referential integrity when data in any key change. You might consider using your application code to achieve it but it is far from ideal. Triggers are much safer, since they centralise the data integrity rules in the database and enforce them for all types of access to the data, be it by program, utility tool, script or server application layer..

Without formal cascading update and delete actions, your custom solution must take care of rows in child tables that will be affected by changes to or deletions of parent keys. For example, if a row is to be deleted from the referenced table, your solution must first delete all rows in all tables that refer to it through foreign keys.

Refer to the topic [Referential Integrity Support](#) in Chapter 30, *Triggers*, for some discussion about implementing custom RI triggers.

Lookup Tables and Your Data Model

We often use lookup tables—also known as *control tables* or *definition tables*—to store static rows that can supply expanded text, conversion factors and the like to output sets and, often, directly to applications as selector lists. Examples are “type” tables that identify entities like account types or document types, “factor” tables used for currency conversion or tax calculation, and “code lookup” tables storing such items as color-matching codes. Dynamic tables are linked to these static tables by matching a foreign key with the primary key of the static table.

Data modeling tools cannot distinguish a lookup relationship from a master-detail relationship since, simplistically, one lookup row can supply values to many “user” rows.

Tools represent it as a parent-child dependency and may erroneously recommend a foreign key on the “child” side.

Yet, in an implemented database, this relationship is not master-detail or parent-child, because the primary key value of the lookup set commands one and only one column. It has no effect on other relationships that this “pseudo-child” participates in.

It is tempting to apply a formal foreign key constraint to columns that reference lookup tables, with the argument that a cascading referential constraint will protect data consistency. The flaw here is that properly designed lookup tables will never change their keys—so there is no potential inconsistency to protect against.

Take a look at this example of a lookup relationship, inherited by converting a very poorly designed Access camping goods application to Firebird. Access client applications can do cute things with entire tables that allow amateurs to build “RAD” applications. This table was used in a visual control that could display a table and transplant a value into another table at the click of a button.

```
CREATE TABLE COLORS (COLOR CHARACTER(20) NOT NULL PRIMARY KEY);
```

Here is a DDL fragment from one of the tables that used COLORS for a lookup:

```
CREATE TABLE STOCK_ITEM (
...
    COLOR CHARACTER(20) DEFAULT 'NEUTRAL',
...,
    CONSTRAINT FK_COLOR FOREIGN KEY (COLOR)
        REFERENCES COLORS(COLOR)
        ON UPDATE CASCADE
        ON DELETE SET DEFAULT;
```

There were a lot of problems with this key. First, the COLORS table was available to the inventory buyers to edit as they saw fit. Updates cascaded all over the system whenever new items came into stock. Deletions frequently stripped the color information from the relatively few items where it mattered. Worst of all, the bulk of items in this system were one color—'NEUTRAL'—with the result that the foreign key's index was a real showstopper on inventory queries.

The “relational way” to avoid the unplanned breakage of consistency would have been to use a lookup key with no meaning as data (i.e. an atomic key):

```
CREATE TABLE COLORS (
    ID INTEGER NOT NULL PRIMARY KEY, /* or UNIQUE */
    COLOR CHARACTER(20));
COMMIT;
INSERT INTO COLORS (ID, COLOR)
    VALUES (0, 'NEUTRAL');
COMMIT;
CREATE TABLE STOCK_ITEM (
...
    COLOR INTEGER DEFAULT 0,
...);
```

Such a key need never change and it can (and should) be hidden from users entirely. Tables that use the lookup table store the stable key. Changes in available values are implemented as new lookup rows with new keys. Values already associated with keys do not change—they are preserved to ensure that history data are not compromised by subsequent changes.

In the event that, even with the higher distribution of key values, the foreign key would produce an index that was still poorly selective over a large table, the improvement to the stability of the table justifies avoiding a formal referential constraint. Existence can be easily enforced using custom triggers.

The REFERENCES Privilege

Firebird implements SQL security on all objects in the database. Every user—except the owner of the database and users with SYSDBA or system superuser privileges—must be GRANTED the necessary privileges to access an object. SQL privileges are discussed in great detail in Chapter 37, **Database Security**.

However, one privilege may be of special significance in the design of your referential integrity infrastructure: the REFERENCES privilege. If the parent and child tables have different owners, a GRANT REFERENCES privilege may be needed to give users sufficient permission to enable referential constraint actions.

The REFERENCES privilege is granted on the referenced table in the relationship—that is, the table referenced by the foreign key—or, at least, on every column of the reference primary or unique key. The privilege needs to be granted to the owner of the referencing table (the child table) and also to any user who needs to write to the referencing table.

At run-time, REFERENCES kicks in whenever the database engine verifies that a value input to a foreign key is contained in the referenced table.

Because this privilege is also checked when a foreign key constraint is defined, it will be necessary for the appropriate permissions to be granted and committed beforehand. If you need to create a foreign key that refers to a table owned by someone else, that owner must first grant you REFERENCES privileges on that table. Alternatively, the owner can grant REFERENCES privileges to a role and then grant that role to you.



Don't make this harder than it needs to be. If there is no requirement to deny read privileges on the referenced table, then have the owner grant the REFERENCES privilege on it to your "general user" role or, if you are using it, to PUBLIC.

If you have these restrictions amongst your requirements, it may be necessary to maintain two separate permissions scripts—one for developers, that is run following table creation, and another for end-users, that is run on an otherwise completed schema.

Handling Other Forms of Relationship

Referential constraints can be applied to other forms of relationship apart from the optional one-to-many form described so far:

- One-to-one
- Many-to-many
- Self-referencing one-to-many (nested or tree relationships)
- Mandatory variants of any form of relationship

One-to-one relationship

Optional 1:1 structures can be valuable where an entity in your data model has a large number of distinct attributes, only some of which are accessed frequently. It can save storage and page reads dramatically to store occasional data in optional “peer” relations that share matching primary keys.

A 1:1 relationship is similar to a 1:many relationship, insofar as it links a foreign key to a unique key. The difference here is that the linking key needs to be unique to enforce the 1:1 relationship—to allow, at most, one dependent row per parent row.

It is usual to double up the use the primary key column(s) of the “peer” table as the foreign key to the parent.

```
CREATE TABLE PARENT_PEER (
    ID INTEGER NOT NULL,
    MORE_DATA VARCHAR(10),
    CONSTRAINT PK_PARENT_PEER PRIMARY KEY(ID),
    CONSTRAINT FK_PARENT_PEER_PARENT
    FOREIGN KEY (ID) REFERENCES PARENT);
```

The effect of this double usage is to cause two mandatory indexes to be created on the column that is the primary key of the peer table: one for the primary key and one for the foreign key. The FK index is stored as if it were non-unique but the PK index will enforce the uniqueness.

Sometimes, especially in older Firebird versions, the optimizer is quirky about this doubling up and could ignore the peer table's primary index. For example,

```
SELECT PARENT.ID, PARENT_PEER.ID,
       PARENT.DATA, PARENT_PEER.MORE_DATA
FROM PARENT JOIN PARENT_PEER
ON PARENT.ID = PARENT_PEER.ID;
```

ignores the primary key index of the peer and produces this plan:

```
PLAN JOIN (PARENT_PEER NATURAL, PARENT INDEX (PK_PARENT))
```

With a “thin” key, such as the one used in the example, the impact on performance may not be severe. With a composite key, the effect may be serious, especially if there will be multiple joins involving several parent-to-peer 1:1 relations. It should at least make you want to test the plans and consider surrogating the primary keys of 1:1 structures.

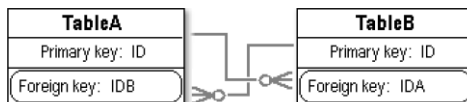


It is not a bad thing if you choose to add a special column to a peer relation in order to implement distinct primary and foreign keys. It can be a useful aid to self-documentation.

Many-to-many relationship

In this interesting case, our data model shows us that each row in TableA may have relationships with multiple rows in TableB, while each row in TableB may have multiple relationships with rows in TableA:

Figure 17.2 Many-to-many relationship



As modelled, this relationship gives rise to a condition known as a *circular reference*. The proposed foreign key in TableB references the primary key of TableA, which means that the TableB row cannot be created if there is no row in TableA with a matching primary key. However, for the same reason, the required row cannot be inserted into TableA if there is no matching primary key value in TableB.

Dealing with a circular reference

If your structural requirements dictate that such a circular reference must exist, it can be worked around. Firebird allows a foreign key value to be NULL—provided the column is not made non-nullable by another constraint—because NULL, being a “non-value”, does not violate the rule that the foreign key column must have a matching value in the referenced parent column. By making the FK on one table nullable, you can insert into that table and create the primary key that the other table requires:

```
CREATE TABLE TABLEA (
    ID INTEGER NOT NULL,
    ...,
    CONSTRAINT PK_TABLEA PRIMARY KEY (ID));
COMMIT;
--
CREATE TABLE TABLEB (
    ID INTEGER NOT NULL,
    ...,
    CONSTRAINT PK_TABLEB PRIMARY KEY (ID));
COMMIT;
--
ALTER TABLE TABLEA
    ADD CONSTRAINT FK_TABLEA_TABLEB
    FOREIGN KEY(IDB) REFERENCES TABLEB(ID);
COMMIT;
--
ALTER TABLE TABLEB
    ADD CONSTRAINT FK_TABLEB_TABLEA
    FOREIGN KEY(IDA) REFERENCES TABLEA(ID);
COMMIT;
```

The workaround:

```
INSERT INTO TABLEB(ID)
    VALUES(1); /* creates a row with NULL in IDB */
COMMIT;
INSERT INTO TABLEA(ID, IDB)
    VALUES(22, 1); /* links to the TABLEB row just created */
COMMIT;
UPDATE TABLEB
    SET IDA = 22 WHERE ID = 1;
COMMIT;
```

Clearly, this model is not without potential problems. In most systems, keys are generated, not supplied by applications. To ensure consistency, it becomes a job for all client applications inserting to these tables to know the value of both keys in both tables within a

single transaction context. Performing the entire operation with a stored procedure would reduce the dependence of the relationship on application code.



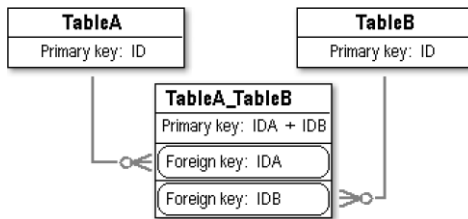
In practice, tables with many:many relationships implemented circularly are very difficult to represent in GUI applications.

Using an intersection table

Generally, it is better practice to resolve many:many relationships by adding an *intersection table*. This special structure carries one foreign key for each table in the many:many relationship. Its own primary key (or a unique constraint) is a composite of the two foreign keys. The two related tables that are intersected by it do not have foreign keys relating to one another at all.

This implementation is easy to represent in applications. Before Insert and Before Update triggers on both tables take care of adding intersection rows when required.

Figure 17.3 Resolution of many:many relationship



Implementing it:

```

CREATE TABLE TABLEA (
    ID INTEGER NOT NULL,
    ...,
    CONSTRAINT PK_TABLEA PRIMARY KEY (ID));
COMMIT;
--
CREATE TABLE TABLEB (
    ID INTEGER NOT NULL,
    ...,
    CONSTRAINT PK_TABLEB PRIMARY KEY (ID));
COMMIT;
--
CREATE TABLE TABLEA_TABLEB (
    IDA INTEGER NOT NULL,
    IDB INTEGER NOT NULL,
    CONSTRAINT PK_TABLEA_TABLEB
    PRIMARY KEY (IDA, IDB));
COMMIT;
--
ALTER TABLE TABLEA_TABLEB
    ADD CONSTRAINT FK_TABLEA FOREIGN KEY (IDA)
        REFERENCES TABLEA,
    ADD CONSTRAINT FK_TABLEB FOREIGN KEY (IDB)

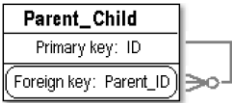
```

```
REFERENCES TABLE;  
COMMIT;
```

Self-Referencing Relationship

If your model has an entity whose primary key refers to a foreign key located in the same entity, you have a self-referencing relationship:

Figure 17.4 Self-referencing relationship



This is the classic *tree* hierarchy, where any row (member, node) can be both parent and child: that is, it can have children (or *branch*) rows dependent on it and, at the same time, it can be a child (branch) of another row (member, node). It needs a CHECK constraint or Before Insert and Before Update triggers to ensure that a PARENT_ID never points to itself.

If your business rules require that parents must exist before children can be added, you will want to use a value, e.g. -1, as the root node of the tree structure. The PARENT_ID should be made NOT NULL and defaulted to your chosen root value. The alternative is leave PARENT_ID as nullable, as in the example below, and use NULL as the root.

In general, custom triggers for Before Insert and Before Update will be required for trees that will be nested to more than two levels. For consistency in trees with a NULL root node, it is important to ensure that constraint actions do not create orphan children unintentionally.

```
CREATE TABLE PARENT_CHILD (  
    ID INTEGER NOT NULL,  
    PARENT_ID INTEGER  
    CHECK (PARENT_ID <> ID));  
COMMIT;  
--  
ALTER TABLE PARENT_CHILD  
    ADD CONSTRAINT PK_PARENT  
    PRIMARY KEY(ID);  
COMMIT;  
--  
ALTER TABLE PARENT_CHILD  
    ADD CONSTRAINT FK_CHILD_PARENT  
    FOREIGN KEY(PARENT_ID)  
    REFERENCES PARENT_CHILD(ID);  
COMMIT;
```

About tree structures

Much more can be said about designing tree structures. It is a challenging topic in relational database design that stretches standard SQL to its boundaries. Unfortunately, it is beyond the scope of this Guide. For some interesting solutions, try **SQL for Smarties** by Joe Celko (Morgan Kaufmann, ISBN 1-55860-323-9).

Mandatory relationships

A mandatory—or obligatory—relationship is one which requires that a minimum of one referencing (child) row exist for each referenced (parent) row. For example, a delivery note structure—a header with customer and delivery address information—would be illogical if it were permitted to have header row without any referencing item lines.

It is a common beginner mistake to assume that a NOT NULL constraint on the child will make a one-to-many relationship mandatory. It does not, because the FOREIGN KEY constraint operates only in the context of an instantiated dependency. With no referencing row, the nullability of the foreign key is irrelevant to the issue.

A mandatory relationship is one place where user-defined trigger constraints must be added to extend referential integrity. Firebird SQL does not provide a “mandatoriness” constraint. It can take some fancy logic at both the client and the server to ensure that events will occur in the right sequence to meet both the referential constraint and the mandatoriness requirements. It will involve both insert and delete triggers, since the logic must enforce the ‘minimum of one child’ rule not only at creation time but when child rows are deleted.

For details about writing triggers and an example of a trigger solution for enforcing a mandatory relationship, refer to Chapter 30, *Triggers*.

“Object is in Use” Error

It is worth mentioning this exception in the context of applying referential integrity constraints, since it is a regular source of frustration for new users. Firebird will not allow a referential constraint to be added or dropped if a transaction is using either of the participating tables. In versions 1.X, you will require exclusive access to avoid this error. In the later versions, you will be allowed to proceed but you can still expect an *Object in Use* exception if the tables affected by creating a FOREIGN KEY constraint are involved in an “interesting” transaction.



If you are yet to start getting your head around transactions (Part V of this book) then, for now, suffice it to say that everything you do potentially affects what someone else is doing and vice-versa. Everything is done in a transaction. Transactions isolate one piece of pending work from other pieces of pending work. At some point you can expect what you want to do to bump into what someone else is doing. When you want to alter the structures of objects, it's always best to operate in an exclusive bubble!

Sometimes it may be less than obvious to you just in what way the object is in use. Other dependencies—such as stored procedures or triggers that refer to your tables, or other referential constraints affecting one or both—can cause this exception to occur if they are in use by an uncommitted transaction. Metadata caches—blocks of server memory that hold metadata used to compile recent client requests and any recently-used stored procedure and trigger code—keep locks on the objects in their possession. Each connection has its own metadata cache, even on a Superserver, so the server could be holding locks on objects that no connection is actually using.

What this is all leading to recommend strongly that you get exclusive access to the database for any metadata changes, particularly those involving dependencies.



If you have exclusive access and the exception still occurs, don't overlook the possibility that you are using it yourself. If you are working in an admin utility with a data browser focused on one of your tables, then the object is in use!

SECOND EDITION

PART IV



Working With Data

CHAPTER

18

DATA MANIPULATION LANGUAGE—DML

The parts of the SQL language lexicon that are used to store, retrieve, update and delete data are known as *data manipulation language*, abbreviated to DML.

As an introduction to DML, in this chapter we take a look at the fundamentals of *sets*, since all DML requests involve them. The statements, syntaxes and expressions available in DML form the content of the next five chapters.

Chapter 19, ***DML Queries***, introduces the structure and syntax of the DML queries that create data, modify it and retrieve it into your applications.

Chapter 20, ***Expressions and Predicates***, describes functions, operations and expressions and how to use them

Chapter 21, ***Querying Multiple Tables***, looks at queries that operate on multiple tables using joins, subqueries and unions. It ends with some tips for optimizing your DML.

Chapter 22, ***Ordered and Aggregated Sets***, examines the syntax and issues for specifying sets that require sorting or grouping and those that retrieve rows from multiple tables without joining

Chapter 23 covers the definition and use of ***views*** and other table-like objects that can be either pre-defined via DDL or derived at run-time from other objects that store or create physical data.

Chapter 24 covers the ***isql*** tool that is distributed with Firebird, providing both an interactive, non-graphical console interface with databases and a means to pass SQL requests to databases from your system's command shell.

The Concept of Sets

All DML queries ultimately target data that are persistently stored in databases as columns in tables. A query defines a logical collection of data items arranged in a specified left-to-right order, in one or more columns, known as a *set*. A query may confine a set's specification to being a single row or just a single column or it may

consist of multiple rows and multiple columns made from the data in several tables. In a multi-row set, the column arrangement of all rows in the set is identical. The query also specifies the operations to be performed upon the set or on selected columns or rows in it. It can use expressions to transform data for retrieval or storage or to provide special search conditions for manipulating sets of data.

Cardinality and degree

A term you will sometimes meet with respect to sets—including tables—is *cardinality*. Cardinality describes the number of **rows** in a set, which might be a table or an output set. More loosely, one may encounter cardinality of a row or cardinality of a key value, referring to the position of a row in an ordered output set.

The term used to refer to the number of columns in a set is **degree**. By extension, one may encounter a phrase like “the degree of a column”, meaning its position in the left-to-right order of the columns in the set.

A table is a set

A table is a set whose complete specification can be accessed in the system tables by the database server when the table is referred to by a query from a client. The Firebird server performs its own internal queries on the system tables to extract the metadata it needs in order to execute client queries.

Output sets

A common use of a query statement is to output a set to the client application, for the purpose of populating the application’s local data structures, often referred to as *datasets* or *recordsets*. The verb used for obtaining an output set is SELECT, following by a list defining the structure of the output rows.

The terms *dataset* and *recordset* are synonymous with output set. Users often see these sets in a visual display. The output set may be in no particular row order, or it can be delivered as a sorted set, as specified in an ORDER BY clause.



“No particular order” means just that. Rows for tables are stored with no top-to-bottom sequencing attributes whatsoever. The order in which unordered sets arrive at the client is not predictable from one query to another.

For example, the following query will output a set of three columns from TABLEA, containing every row that matches the conditions specified in the WHERE clause. The rows will be sorted so that the row with the lowest value in COL1 will appear first:

```
SELECT
  COL1,
  COL2,
  COL3
FROM TABLEA
WHERE COL3 = 'Mozart'
ORDER BY COL1;
```


If no WHERE clause is provided, the set will contain every row from TABLEA, not just those have the value 'Mozart' in COL3.

If all columns of TABLEA are wanted, then the symbol '*' can optionally be used instead of itemizing the columns of the set. For example, the following statement will retrieve all of the columns in TABLEA and, because there is no WHERE clause limiting the set, it will consist of all of the rows in the table:

```
SELECT * FROM TABLEA;
```

Input sets

Just as the SELECT ... FROM part of a retrieval query specifies a set for an output or cursor operation, so the other DML statements (INSERT, UPDATE and DELETE) specify input sets identifying the data upon which the operation is to be performed.

An INSERT query's input set is specified by identifying a single table and a left-to-right ordered list of identifiers for columns which are to receive the input values defined in the subsequent VALUES() clause. The VALUES() clause must supply a list of values that corresponds exactly to the input set's order and the data types of its members.

The following example defines an input set consisting of three columns from TABLEB. The constants in the VALUES() clause will be checked to ensure that there are exactly three values and that they are of the correct (or compatible) data types:

```
INSERT INTO TABLEB(
    COLA,
    COLB,
    COLC)
VALUES(
    99,
    'Christmas 2014',
    '2014-12-25');
```



INSERT has an alternative syntax where the VALUES() clause is replaced by selecting a set from one or more other tables, real or virtual. INSERT statements are discussed in more detail below.

An UPDATE query defines its input set by identifying a single table and listing one or more input columns, together with their new values, in a SET clause. It identifies the target rows using a WHERE clause:

```
UPDATE TABLEB
    SET COLB = 'Marine Day (Japan)',
        COLC = '2014-07-21'
WHERE ...;
```

UPDATE statements are discussed in more detail below.

A DELETE query cannot specify columns in its input set—it is implicitly always '*' (all columns). It identifies the target rows using a WHERE clause. For example, the following query will delete every row where COLC (a DATE type) is earlier than 13th December, 2009:

```
DELETE FROM TABLEB
    WHERE COLC < '2009-12-13';
```

DELETE statements are discussed in more detail below.

Output sets as input sets

Several kinds of queries, more complex than the simple SELECTs touched on so far, involve the creation of internal output sets behind the scenes that are, in turn, used as input to a successive stage in the retrieval of the ultimate set that has been requested by the client application.

Grouped or aggregated sets

SQL has an important feature that uses an input set formed from an output set generated within the same SELECT query—the GROUP BY clause. Instead of the output set from column list being passed to the client, it is passed instead to a further stage of processing, where the GROUP BY clause causes the data to be partitioned into one or more nested groups. Each partition typically extracts summaries by way of expressions that summarise (“aggregate”) numeric values at one or more levels.

For more information about aggregated queries, see Chapter 22.

Streams and rivers for JOINS

When columns in one table are linked to columns in another table to form joined sets, the input column sets from each table are known as *streams*. The output of two joined streams is referred to as a *river*. When joins involve more than two tables, streams and rivers become the input for further joining of streams and rivers, until the final river becomes the output of the query.

The Optimizer

A group of Firebird engine routines—known as *the optimizer*—subjects a join query specification and the indexes available to a cost-evaluation process referred to as *optimization*. The optimizer generates a plan, which is a “best-cost” map of the course the engine will take when generating that final “output river” when the statement is subsequently executed. on subsequent executions of the statement.

Refer to the topic [Query Plans and the Optimizer](#) at the end of the next chapter, **DML Queries**, for a more detailed discussion.

Cursor sets

A SELECT statement can define a set that is not output to the client at all, but remains on the server to be operated on by a server side cursor. The cursor itself is a pointer that the application instructs to fetch rows one-by-one, on demand, by progressing forward through a prepared SELECT statement.

The objective of a cursor operation is to use data from the cursor set and “do something”, consistently for each cursor set row processed. The rows of the specified set are operated on, one-by-one, in a “forward” (top-to-bottom) direction. It can be an efficient way to batch-process sets, where the process performs one or more tasks on the current cursor row, inside a single iteration of a loop through the cursor set.

For example, at each iteration, data items could be taken from the current cursor set row and used for inserting new rows into a specified input set.

Firebird’s procedural SQL language—PSQL—has a commonly used DML language extension for opening a cursor, that takes the form

```
FOR SELECT <any valid select specification>
```

```

INTO <list of pre-declared local variables> DO
BEGIN
    < optionally operate on variables >;
    SUSPEND;
END

```

This is an *unnamed cursor* syntax. PSQL also supports *named cursors*, which a statement or a block of PSQL code can engage, using **WHERE CURRENT OF** <cursor-name> instead of a search condition, to perform positioned updates or deletes on the underlying rows targeted by the cursor set.

Implementing cursors

- To use a cursor, a dynamic application must pre-declare it using the function *isc_dsql_set_cursor_name*. The client application is responsible for ensuring that the prepared statement is available for assignment at the appropriate point in execution of the program. While some data access components realize this interface, many do not.
- Embedded SQL (ESQL) provides the **DECLARE CURSOR** statement for implementing a pre-compiled named cursor declaration.
- The PSQL language provides syntax to operate on both named and unnamed cursors locally inside a stored procedure, trigger or dynamically executable block. For more information, refer to Chapter 29, **Stored Procedures and Executable Blocks**.

Except for the techniques available in PSQL, cursors are beyond the scope of this book.

Nested Sets

Nested sets are formed using a special kind of SQL expression syntax known as a subquery. This kind of expression takes the form of a bracketed **SELECT** statement that can be used in several ways:

- A subquery embedded into the column specification list of the main query can be used as a “look-up” to retrieve a single, run-time output value.
- In a **WHERE** clause, a nested set of zero or more rows consisting of a single field can be defined in a search condition, as the argument to an **IN()** expression.
- A set of rows and columns known as a *derived table* can be specified by a nested **SELECT** and used in joins within a complex **SELECT** statement.

The allowed **SELECT** syntaxes vary according to the context in which the nested set is used. They are described in detail as they occur in the proceeding chapters.

For a simple example, a query on **TABLEA** uses a nested subquery to include a run-time output column named **DESCRIPTION** derived from **COLX** in **TABLEB**:

```

SELECT
    COL1,
    COL2,
    COL3,
    (SELECT COLA FROM TABLEB WHERE COLX='Espagno1') AS DESCRIPTION
FROM TABLEA
WHERE ... ;

```

Notice the use of the keyword `AS` to mark the identifier ('DESCRIPTION') of the derived column. It is optional, but recommended for code clarity.

Correlated subqueries

It is very common for an embedded query to be *correlated* with the main query. A correlated subquery is one whose `WHERE` clause is linked to one or more values in the output list of the outer query or to values in other tables involved in a more complex query. A correlated subquery is similar in effect to an inner join and can sometimes be used to good effect in place of a join.

The topic of subqueries is discussed in detail in the following chapters, especially, in Chapter 21, **Querying Multiple Tables**. Derived tables appear in Chapter 23, **Views and Other Run-time Set Objects**.

SQL Privileges and DML

Data manipulation—reading from and writing to tables—require database privileges, controlled by declarations made with `GRANT` and `REVOKE`. For information, refer to Chapter 37, **Database-level Security**.

CHAPTER

19

DML QUERIES

Data manipulation operations fall into two broad categories: those that retrieve data from the abstract structures that store them and those that add, modify or remove data to or from those structures.

A query uses an SQL statement from the DML lexicon to specify the source or target data for the operation being requested and, as required, how to retrieve or modify it.

Commonly, the statement is stored in the code of the application, ready to be submitted at an appropriate point in the application flow. Strictly speaking, a statement doesn't become a query until it is submitted to the server. However, most developers use the terms *request*, *statement* and *query* interchangeably.

The SELECT Statement

The SELECT statement is the fundamental method for *retrieving* sets of data from the database. It has the following general form:

```
SELECT
    [FIRST (m)] [SKIP (n)] [[ALL] | DISTINCT]
    <list of columns> [, [column-name] | expression | constant ] [AS] alias-name]
FROM <table-or-procedure-or-view>
    [{[[INNER] | [{LEFT | RIGHT | FULL} [OUTER]] JOIN}] <table-or-procedure-or-view>
    ON <join-conditions> [{JOIN..}]
[WHERE <search-conditions>]
[GROUP BY <grouped-column-list>]
[HAVING <search-condition>]
[UNION <select-expression> [ALL]]
[PLAN <plan-expression>]
[ORDER BY <column-list>]
[ROWS <m> [TO <n>]]
[FOR UPDATE [OF col1 [,col2..]] [WITH LOCK]]
```

Clauses in a SELECT statement

In the following topics, we take an introductory look at each allowable clause of the SELECT statement. Most clauses are optional but it is important to present those you do use in the correct order.

Optional set quantifiers

Following the SELECT keyword, a set quantifier can be included to govern the inclusion or suppression of rows in the output set once they have met all other conditions.

ALL

This is the default quantifier for output lists and is usually omitted. It returns all rows that meet the conditions of the specification.

See also the existential quantifiers *ANY() and SOME() Predicates* and *The SINGULAR() Predicate* in the next chapter.

DISTINCT

This row quantifier suppresses all duplicate rows in the sets that are output.

For example, the table EMPLOYEE_PROJECT stores an intersection record for each employee (EMP_NO) and project (PROJ_ID) combination, resolving a Many:Many relationship. EMP_NO + PROJ_ID forms the primary key.

```
SELECT DISTINCT EMP_NO, PROJ_ID FROM EMPLOYEE_PROJECT;
```

returns 28 rows, i.e. all of them, the same as SELECT [ALL], because every occurrence of (EMP_NO + PROJ_ID) is, by nature, distinct..

```
SELECT DISTINCT EMP_NO FROM EMPLOYEE_PROJECT;
```

and

```
SELECT DISTINCT PROJ_ID FROM EMPLOYEE_PROJECT;
```

return 22 and 5 rows, respectively.

Evaluation of distinctness is applied to all of the output columns, making it useful in some queries that use joins to produce a denormalized set. Test it well to ensure that it produces the result you expect.

FIRST (m) SKIP (n)

The optional keywords FIRST (m) and/or SKIP (n), if used, precede all other specifications. They provide the option to limit the output from an ordered set to a range of *m* rows, optionally starting the set after skipping the first *n* rows. It makes no sense to use this construct with an unordered set. The ORDER BY clause obviously needs to use an ordering condition that actually makes sense with regard to the selection of the candidate rows.

The two keywords can be used together or individually. The arguments *m* and *n* are integers, or expressions that resolve to integer. The brackets around the *m* and *n* values are required for expression arguments and optional for simple integer arguments.

Since FIRST and SKIP operate on the set that is output by the rest of the specification, they can not be expected to make the query execute faster than the full specification would otherwise do. Any performance benefit comes from the reduction of traffic across the wire.

Example `SELECT FIRST 5 SKIP 100 MEMBER_ID, MEMBERSHIP_TYPE, JOIN_DATE`

```
FROM MEMBERS
ORDER BY JOIN_DATE;
```

The example above will return five rows, starting at the 101st row in the ordered set. No exception occurs if the ordered set does not have enough rows to satisfy the `FIRST n` specification.



*To return the *n* highest-order rows according to the ordering conditions, order the set in `DESC[ENDING]` order.*

ROWS <*m*> [TO <*n*>]

Introduced in the “2” series, this standard SQL clause achieves the same as the non-standard `FIRST (m) SKIP (n)` construct, limiting the range of output from an ordered set. Although the surfaced syntax for each method is different, the underlying processing of the set is the same.

If `ROWS <m>` is used alone, it is the same as using `FIRST <m>` alone: the first *m* rows are returned. However, if both the <*m*> and <*n*> arguments are used, the set returned will be *n* rows long (or less, if the intermediate output contains fewer than *n* rows), skipping all rows preceding the *m*th row.

In the following example, the same set is returned as in the example above for the `FIRST...SKIP` syntax:

Example

```
SELECT MEMBER_ID, MEMBERSHIP_TYPE, JOIN_DATE
FROM MEMBERS
ORDER BY JOIN_DATE
ROWS 6 TO 106 ;
```

ROWS or FIRST...SELECT?

The two syntaxes are almost interchangeable but an attempt to mix parts of the `ROWS` syntax with parts of the `FIRST...SKIP` syntax within the same `SELECT` statement or expression will cause an exception. However, a `SELECT FIRST...[SKIP]` statement or expression can embed a `SELECT...ROWS`, and vice versa.

If you are using Firebird 2 or higher, the standards-compliant `ROWS` syntax is preferred. It is also safe to use a `SELECT` expression with `ROWS` as the target set specifier in a `DELETE` statement, whereas using a `SELECT FIRST...[SKIP]` in the same manner will have the undesired result of zapping all of the rows from the table. You can read more about `SELECT` (subquery) expressions in the next chapter, *Expressions and Predicates*.

SELECT <list of columns>

The `SELECT` clause defines the list of columns that are to be returned in the output set. It must contain at least one column, but it doesn't have to be a column that exists in a table. That statement is not as strange as it sounds. The column list is really an output specification and this is data manipulation language (DML). The output specifications can include any of the following:

- the identifier of a column that is stored in a table, specified in a view or declared as an output argument to a stored procedure. Under some conditions, the column identifier must be qualified with the name or alias of the table it belongs to.
- a simple or complex expression, accompanied by a run-time identifier
- a constant value, accompanied by a run-time identifier

- a server context variable, accompanied by a run-time identifier
- the '*' symbol, popularly known as “select star”, which specifies every column. Although SELECT * does not preclude selecting one or more columns from the same table individually, it does not make sense to do so in general. To include a duplicated column for a special purpose, apply the AS keyword and an alias to it and return it as a computed (read-only) field.

The following SELECT specifications are all valid:

Simple list of columns:

```
SELECT COLUMN1, COLUMN2 ...
```

Qualified column names, required in multi-table specifications

```
SELECT
    TABLEA.ID,
    TABLEA.BOOK_TITLE,
    TABLEB.CHAPTER_TITLE,
    CURRENT_TIMESTAMP AS RETRIEVE_DATE ...
```

Expression (aggregating)

```
SELECT MAX(COST * QUANTITY) AS BEST_SALE ...
```

Expression (transforming)

```
SELECT 'EASTER' || CAST(EXTRACT(YEAR FROM CURRENT_DATE) AS CHAR(4)) AS SEASON ...
```

Variables and constants

```
SELECT
    ACOLUMN,
    BCOLUMN,
    CURRENT_USER, /* context variable */
    'Jaybird' AS NICKNAME ... /* run-time constant */
```

All columns in a table

```
SELECT * ...
```

Quantifiers—either syntax needs an ORDER BY clause to make sense:

```
SELECT FIRST 5 SKIP 100 ACOLUMN, BCOLUMN ...
...
ORDER BY 1
--
SELECT ACOLUMN, BCOLUMN ...
..
ORDER BY 1
ROWS 6 TO 106
```

Expressions and constants as output fields

A constant or an expression—which may or may not include a column name—can be returned as a read-only, run-time output field. The field should be given a column name of its own, unique in all sets involved. Names for run-time columns are known as *column aliases*. For clarity, the column alias can be marked optionally with the keyword AS.

Taking the example above:

```
SELECT 'EASTER' || CAST(EXTRACT(YEAR FROM CURRENT_DATE) AS CHAR(4)) AS SEASON ...
```

In 2014, this column alias will be returned in every row of the set as


```
... SEASON ...
=====
EASTER2014
```

Constants, many different kinds of expressions, involving functions and calculations, and scalar subqueries—including correlated subqueries—can be massaged into read-only output fields.



Constants of types BLOB and ARRAY can not be output as run-time fields.

For details about expressions and functions, refer to the next chapter..

FROM <table-or-procedure-or-view>

The FROM clause specifies the source of data, which may be a table, a view or a stored procedure that has output arguments. If the statement involves joining two or more structures, the FROM clause specifies the leftmost structure. Other tables are added to the specification by way of succeeding ON clauses (see JOIN <specification>, below).

In the following examples, some FROM clauses are added to the SELECT specifications in the previous set of examples:

```
SELECT
    COLUMN1,
    COLUMN2
FROM ATABLE ...

--
SELECT
    TABLEA.ID,
    TABLEA.BOOK_TITLE,
    TABLEB.CHAPTER_TITLE,
    CURRENT_TIMESTAMP AS RETRIEVE_DATE
FROM TABLEA ...

--
SELECT
    MAX(COST * QUANTITY) AS BEST_SALE
FROM SALES ...

--
SELECT
    ACOLUMN,
    BCOLUMN,
    CURRENT_USER,
    'Jaybird' AS NICKNAME
FROM MYTABLE ...

--
SELECT
    'EASTER' || CAST(EXTRACT(YEAR FROM CURRENT_DATE) AS CHAR(2))
    AS SEASON
FROM RDB$DATABASE ;
```

What is this table RDB\$DATABASE?

RDB\$DATABASE is a system table that stores one and only one row, consisting of pieces of header information about the database. What's stored in it is immaterial to the widespread use Firebird developers make of it. It is the fact that it always has one row, no more, no less, that makes it handy when we want to retrieve a constant or calculated value from the server that isn't derived from a column defined in a table, view or stored procedure.

For example, to get a new generator value into an application—something we may need for creating rows on the detail side of a new master-detail structure—we can create a function in our application that calls this query:

```
SELECT GEN_ID(MyGenerator, 1) FROM RDB$DATABASE;
```

If you've had anything to do with Oracle® databases, you'll recognize it as Firebird's answer to DUAL.

JOIN <specification>

Use this clause to add the names and joining conditions of the second and each subsequent data stream (table, view or selectable stored procedure) that contributes to a multi-table SELECT statement—one JOIN ... ON clause for each source set. JOIN syntax and issues are discussed in detail in Chapter 21, *Querying Multiple Tables*.

The following statement illustrates a simple inner join between the two tables from the previous example:

```
SELECT
    TABLEA.ID,
    TABLEA.BOOK_TITLE,
    TABLEB.CHAPTER_TITLE,
    CURRENT_TIMESTAMP AS RETRIEVE_DATE
FROM TABLEA
    JOIN TABLEB
        ON TABLEA.ID = TABLEB.ID_B ...
```

Table aliases

In the same statement fragment, table identifiers can be optionally substituted with table aliases. For example:

```
SELECT
    T1.ID,
    T1.BOOK_TITLE,
    T2.CHAPTER_TITLE,
    CURRENT_TIMESTAMP AS RETRIEVE_DATE
FROM TABLEA T1
    JOIN TABLEB T2 ON T1.ID = T2.ID_B
...
```

For more information about why and how table aliases are used in multi-table queries, refer to *Using Relation Aliases* in the next chapter.

SQL-89 inner join syntax

Firebird provides backward support for the deprecated SQL-89 implicit inner join syntax, e.g.

```
SELECT
```

```

TABLEA.ID,
TABLEA.BOOK_TITLE,
TABLEB.CHAPTER_TITLE,
CURRENT_TIMESTAMP AS RETRIEVE_DATE
FROM TABLEA, TABLEB ...

```

For several reasons, you should avoid this old syntax in new applications.

WHERE <search-conditions>

Search conditions limiting the rows for output are located in the WHERE clause. Search conditions can vary from a simple match condition for a single column to complex conditions involving expressions, AND, OR and NOT operators, type-casting, character set and collation conditions and more.

The WHERE clause is the filtering clause that determines which rows are candidates for inclusion in the output set. Those rows that are not eliminated by the search conditions of the WHERE clause may be ready for output to the requestor or they may “go forward” for further processing: ordering by an ORDER BY clause, with or without consolidation by a GROUP BY clause.

The following simple examples illustrate some WHERE clauses using a sampling of conditions to limit the rows selected:

```

SELECT
  COLUMN1,
  COLUMN2
FROM ATABLE
  WHERE ADATE BETWEEN '2012-12-25' AND '2014-12-24'...
-- limits rows returned to a specific period of time
--
SELECT
  T1.ID,
  T2.TITLE,
  CURRENT_TIMESTAMP AS RETRIEVE_DATE
FROM TABLEA T1
  JOIN TABLEB T2
    ON T1.ID = T2.ID_B
  WHERE T1.ID = 99 ;
/* limits rows returned to those where the matching ID columns
have the value 99 */
--
SELECT MAX(COST * QUANTITY) AS BEST_SALE
FROM SALES
  WHERE SALES_DATE > '31.12.2011'...
-- only rows with sales dates in 2012 or later will be returned

```

The next chapter is devoted to the topic of expressions and predicates for defining search conditions.

Parameters in WHERE clauses

Data access interfaces that implement the Firebird API have the capability to process the constants in search conditions as replaceable parameters. Thus, an application can set up a

statement with dynamic search conditions in the WHERE clause, for which values are assigned just before each execution of the query. This capability is sometimes referred to as *late binding*.

For more details, see the topic below, *Using Parameters*.

GROUP BY <grouped-column-list>

The output from the SELECT statement can optionally be partitioned into one or more nested groups that aggregate (summarize) the sets of data returned at each nesting level. These groupings often include aggregating expressions, that is, expressions containing functions that work on multiple values, such as totals, averages, row counts, minimum/maximum values.

The following simple example illustrates a grouping query. The SQL aggregating function SUM() is used to calculate the total items sold for each product type:

```
SELECT
    PRODUCT_TYPE,
    SUM(NUMBER_SOLD) AS SUM_SALES
FROM TABLEA
    WHERE SERIAL_NO BETWEEN 'A' AND 'K'
    GROUP BY PRODUCT_TYPE;
```

The output might be similar to the following:

PRODUCT_TYPE	SUM_SALES
-----	-----
Gadgets	174
Whatsits	25
Widgets	117

Firebird provides an extensive range of grouping capability, with very strict rules governing the logic. Aggregating expressions are discussed in the next chapter, while detailed information about GROUP BY can be followed up in Chapter 22, **Ordered and Aggregated Sets**.

HAVING <grouping-column predicate>

The optional HAVING clause may be used in conjunction with a grouping specification to include or exclude rows or groups, similarly to the way the WHERE clause limits row sets.

In this example, we modify the output of the previous example by using a HAVING clause to limit the output to those groups whose SUM_SALES is more than 100:

```
SELECT
    PRODUCT_TYPE,
    SUM(NUMBER_SOLD) AS SUM_SALES
FROM TABLEA
    WHERE SERIAL_NO BETWEEN 'A' AND 'K'
    AND PRODUCT_TYPE = 'WIDGETS'
    GROUP BY PRODUCT_TYPE
    HAVING SUM(NUMBER_SOLD) > 100;
```

The output:

PRODUCT_TYPE	SUM_SALES
-----	-----
Gadgets	174

UNION <select-specification>

UNION sets are formed by combining two or more separate query specifications—which may involve different tables—into one output set. The only restriction is that the output columns in each separate output specification must match by degree, type and size. That means they must each output the same number of columns in the same left-to-right order and that each column must be consistent throughout in data type and size.

By default, a UNION set suppresses duplicates in the final output set. To retain all duplicates, include the keyword ALL.

UNION sets are discussed in more detail in Chapter 21, *Querying Multiple Tables*.

PLAN <plan-expression>

The PLAN clause allows a query plan to be optionally included in the query specification. It is an instruction to the optimizer to use particular indexes, stream order and access methods for the query. The optimizer creates its own plan when it prepares a query statement: you can view the plan in isql and many of the available GUI utilities. Usually, the optimizer knows best but, sometimes, it can be worthwhile to experiment with variations to the optimizer's plan when you think a query is slower than it could be.

Query plans and the syntax of plan expressions are discussed at the end of this chapter, in the topic [c *Query Plans and the Optimizer*](#).

ORDER BY <column-list>

Use this clause when your output set needs to be sorted in a specific order. For example, to get a list of names in alphabetical order by last name and first name:

```
SELECT
  EMP_NO,
  LAST_NAME,
  FIRST_NAME
FROM EMPLOYEE
ORDER BY LAST_NAME, FIRST_NAME;
```

Unlike GROUP BY columns, ORDER BY columns do not have to present in the output specification (SELECT clause). The identifier of any ordering column that also appears in the output spec can be replaced by its position number in the output spec, counting from left to right:

```
SELECT
  EMP_NO,
  LAST_NAME,
  FIRST_NAME
FROM EMPLOYEE
ORDER BY 2, 3;
```

Pay close attention to the indexes on columns that are going to be used for sorting. Refer to Chapter 16, *Indexes*, for guidance. For more about the syntaxes and issues, see Chapter 22, *Ordered and Aggregated Sets*.

The FOR UPDATE clause

The optional FOR UPDATE clause instructs the engine to wait for a FETCH call, fetch one row into the cursor for a “current row” operation and then wait for the next FETCH call. As each row is fetched, it becomes earmarked for an update operation. It will not work if the database or the transaction is read-only.

Syntax `SELECT...
 [FOR UPDATE [OF col1 [,col2...]]] [WITH LOCK]]`

The optional sub-clause OF <column-list> can be used in embedded SQL (ESQL) to restrict the fields present in a *named cursor* that are allowed to be updated. In DSQL and PSQL it is a “no-op” and should just be ignored.

Named cursors

A named cursor is not achievable in DSQL except indirectly, by way of an executable block of PSQL invoked by an EXECUTE BLOCK statement. It can be achieved in dynamic applications by way of the API function *isc_dsql_set_cursor_name* and in static SQL (ESQL) by way of a DECLARE CURSOR statement.

For more information about PSQL cursors, including their use in EXECUTE BLOCK statements, refer to Chapter 29, **Stored Procedures and Executable Blocks**.

ESQL and direct use of the Firebird API functions are beyond the scope of this book. For the curious, more information can be found in the original beta documentation set for InterBase 6.0, published by the defunct Inprise Corporation and now archived by IBPhoenix, particularly the titles *Embedded SQL* and *API Guide*.

WITH LOCK

The optional WITH LOCK sub-clause enables a certain degree of explicit record locking (“pessimistic locking”). Explicit record locking is contrary to the architectural design of Firebird and will almost certainly cause problems if used without expert knowledge of how transactions work in Firebird.

Use with extreme caution!

Without the FOR UPDATE sub-clause, the pessimistic lock will succeed only if all of the rows specified are free of pending updates and deletes. If the locking of the whole set succeeds, then those rows and any rows in other tables that depend on them will be inaccessible to other transactions until the transaction in hand is committed or rolled back.

With the FOR UPDATE sub-clause, the lock is attempted on each row of the set specified as it is retrieved for buffering. While the lock might succeed on one or more initial rows, it is likely to fail as the retrieval proceeds because another transaction has entailed a row in a pending write operation. The result will be that the task is interrupted with an exception that will require skilful handling by the client application.

If your requirements leave you with no choice but to resort to pessimistic locking, minimise the risk by making the affected set as small as possible, preferably just a single row, and by controlling the operation tightly in the application code.



WITH LOCK can only be used with a top-level SELECT statement over a single, natural table. It is not available in a subquery expression, including derived tables and common table expressions, nor with any joined, grouped or aggregated set, nor with views, selectable stored procedure output or external tables.

INTO <target-list>

Available only for a [FOR] SELECT statement that is called inside a PSQL module. The <target-list> argument is a comma-separated list of local variables prefixed with colons, context variables or constants.

For more information about PSQL variables and the INTO syntax, refer to Chapter 28, *Procedural SQL—PSQL*.

INSERT and UPDATE Statements

When new data are to be added to tables, the basic SQL verb for the request is *INSERT INTO*. When existing rows are to be modified, the verb is *UPDATE*. Firebird also supports two syntaxes for testing whether a row with a certain unique value exists, each resulting in an UPDATE operation if it does exist and an INSERT if it does not. One form is *UPDATE OR INSERT*; the other is *MERGE*.

INSERT INTO

The INSERT INTO statement is used for adding new rows to a single table. SQL does not permit rows to be inserted to more than one table in a single INSERT statement.

Insert statements can operate on views, under some conditions. Refer to Chapter 23 for discussion of views that can accept inserts on behalf of an underlying table.

The INSERT statement has two general forms for passing the values for the input column list:

Inserting a list of constants

```
INSERT INTO table-name | view-name (<list of columns>)
VALUES (<matching list of values>)
```

Inserting from an in-line query

```
INSERT INTO <table> (<list of columns>)
SELECT [[FIRST m] [SKIP n]] <matching list of values from another set>
[ORDER BY <in-line column[s]> [DESC]]
```

In the following example, an INSERT INTO clause defines an input set for TABLEB and a SELECT clause defines a corresponding in-line query from TABLEA to supply values to the input set:

```
INSERT INTO TABLEB
(COLA, COLB, COLC)
SELECT COL1, COL2, COL3 FROM TABLEA;
```



It is not possible to insert into an in-line query.

The RETURNING clause

If the INSERT statement inserts a single row, it can include the optional RETURNING clause specifying a set of one or more columns, constants or expressions to be returned to the caller, on completion, as a one-row set. The fields return can comprise anything useful,

including the NEW context value as it stood after execution of any BEFORE INSERT triggers.

A set of NULL is returned if the statement does not find a row to update.



The RETURNING clause is not available in versions older than v.2.1.

Example

```
INSERT INTO BIRDS (
    GENUS, SPECIES, COMMON_NAME)
VALUES ('Psittaciformes ', 'Strigops habroptilus', 'Kakapo')
RETURNING ID, NEW.GENUS, NEW.SPECIES
```

INTO <target-list>

Available only for an INSERT statement that is called inside a PSQL module. The <target-list> argument is a comma-separated list of local variables prefixed with colons, context variables or constants designed to accept the returned values in the correct order and data type.

For more information about PSQL variables and the INTO syntax, refer to Chapter 28, **Procedural SQL—PSQL**.

Inserting into BLOB columns

As a rule, if you need to insert BLOBs as part of a VALUES() list, they must be constructed in the client application using API functions and structures. In DSQL applications, they are usually passed as streams. The segment size can simply be disregarded except in ESQL applications.

However, if you are passing a VALUES() entry to a BLOB as text, Firebird will accept character strings as input.

For example,

```
INSERT INTO ATABLE (BLOBMEMO)
VALUES ('Now is the time for all good men to come to the aid of the party');
```

This capability will suit conditions where the text to be stored will never exceed 32767 bytes (or 32766, if you are reading a VARCHAR field into a BLOB). For many programming interfaces, the problem may seem a non-issue, because they cannot handle such large strings, anyway. However, since Firebird will accept concatenated SQL string expressions, such as MYVARCHAR1 || MYVARCHAR2, it will be necessary to protect string inputs from overflows.

The INSERT INTO ...SELECT technique passes BLOBs to BLOBs directly.

Inserting into ARRAY columns

In dynamic SQL, it is not possible to insert data into ARRAY columns at all. It is necessary to implement a custom method in application or component code, that calls the API function `isc_array_put_slice`.

In embedded applications (ESQL) it is possible to construct a precompiled SQL statement to insert entire arrays into an array column. Errors can occur if the data does not exactly fill the array.

Using INSERT with automatic fields

Table definitions may have defined columns whose field values are populated with data automatically when a new row is inserted. This may happen in several ways:

- a column is defined by a **COMPUTED BY** clause
- a column, or a domain under which it was defined, includes a **DEFAULT** clause
- a **BEFORE INSERT** trigger has been added to the table to populate the column automatically

Automatic fields may affect the way you formulate INSERT statements for the table.

COMPUTED BY columns

Including a computed column in the input column list is not valid and will cause an exception—as illustrated by the following example:

```
CREATE TABLE EMPLOYEE (
    EMP_NO INTEGER NOT NULL PRIMARY KEY,
    FIRST_NAME VARCHAR(15),
    LAST_NAME VARCHAR(20),
    BIRTH_COUNTRY VARCHAR(30) DEFAULT 'Taiwan',
    FULL_NAME COMPUTED BY FIRST_NAME || ' ' || LAST_NAME);
COMMIT;
INSERT INTO EMPLOYEE (
    EMP_NO,
    FIRST_NAME,
    LAST_NAME,
    FULL_NAME)
VALUES (99, 'Jiminy', 'Cricket', 'Jiminy Cricket');
```

Columns with defaults

If a column has a default defined for it, the default will work

- only on inserts and
- only if the defaulted column is *omitted* from the input column list.

If the statement in the previous example is corrected, it will cause ‘Taiwan’ to be written to the **BIRTH_COUNTRY** column:

```
INSERT INTO EMPLOYEE (EMP_NO, FIRST_NAME, LAST_NAME)
VALUES (99, 'Jiminy', 'Cricket');
COMMIT;
-- (continued)
SELECT * FROM EMPLOYEE WHERE EMP_NO = 99;

EMP_NO  FIRST_NAME  LAST_NAME  BIRTH_COUNTRY  FULL_NAME
=====
99      Jiminy      Cricket    Taiwan         Jiminy Cricket
```

Defaults do not kick in to replace NULLs:

```
INSERT INTO EMPLOYEE (EMP_NO, FIRST_NAME, LAST_NAME, BIRTH_COUNTRY)
VALUES (100, 'Maria', 'Callas', NULL);
COMMIT;
```

```
--
SELECT * FROM EMPLOYEE WHERE EMP_NO = 100;

EMP_NO  FIRST_NAME  LAST_NAME  BIRTH_COUNTRY  FULL_NAME
=====
100     Maria       Callas                Maria Callas
```

If you are developing applications using components that generate INSERT statements from the column specifications of the datasets' SELECT statements—for example, Delphi and JBuilder—be especially aware of this behavior. If your component does not support a method to get the server defaults, it may be necessary to customize the update statement that it uses to perform an insert from a dataset.

BEFORE INSERT triggers

When inserting into a table that uses triggers to populate certain columns automatically, make sure you omit those columns from your input list if you want to ensure that the trigger will do its work.

This is not to say that you must always omit triggered fields from input lists. A trigger that is designed to provide a value in case no value is supplied by INSERT statements is highly recommended, to bomb-proof data integrity—especially where multiple applications and tools are used to access your databases. The trigger should test the input for certain conditions—NULL, for example—and do its stuff according to the conditions.

In the following example, the primary key, OID, is populated by a trigger:

```
CREATE TABLE AIRCRAFT (
    OID INTEGER NOT NULL,
    REGISTRATION VARCHAR(8) NOT NULL,
    CONSTRAINT PK_AIRCRAFT PRIMARY KEY (OID));
COMMIT;

--
SET TERM ^;
CREATE TRIGGER BI_AIRCRAFT FOR AIRCRAFT
    ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    IF (NEW.OID IS NULL) THEN
        NEW.OID = GEN_ID(ANYGEN, 1);
    END ^
SET TERM ;^
COMMIT;

--
INSERT INTO AIRCRAFT (REGISTRATION)
    SELECT REGISTRATION FROM AIRCRAFT_OLD
        ORDER BY REGISTRATION
    ROWS 3 TO 6;
COMMIT;
```

In this case, the trigger fetches the generator value for the primary key, because no value was passed in the input list for it. However, because the trigger was set up to do this only when it detects NULL, the following INSERT statement will work just fine, too—

provided, of course, that the value supplied for OID does not violate the unique constraint on the PK:

```
INSERT IN AIRCRAFT (OID, REGISTRATION)
VALUES(1033, 'ECHIDNA');
```

When and why would you do this? Surprisingly often, in Firebird. When implementing master-detail structures, you can secure the primary key of the master row from its generator before that row is posted to the database, by a simple DSQL call:

```
SELECT GEN_ID (YOURGENERATOR, 1) FROM RDB$DATABASE;
```

Alternatively:

```
SELECT NEXT VALUE FOR YOURGENERATOR
FROM RDB$DATABASE;
```

Generators operate outside transaction control and, once you have that number, it is yours. You can apply it to the foreign key column of the detail rows in client buffers, as you create them, without posting a master record to the database. If the user decides to cancel the work, there is nothing to “undo” on the server. If you have ever struggled to achieve this kind of capability with a DBMS that uses an autoincrement, or “identity” type, you will learn to love this feature.

For more angles on this technique, see Chapter 30, *Triggers*.

UPDATE

The UPDATE statement is used for changing the values in columns in existing rows of a table. It can also operate on a table through a cursor sets or an “updatable” view. SQL does not allow an UPDATE statement to target rows in multiple tables.

Refer to Chapter 23 for discussion of views that can accept updates on behalf of an underlying table.

An UPDATE query that modifies only the current row of a cursor is known as a *positioned* update. One which may update multiple rows is known as a *searched* update.

Positioned vs searched operations

UPDATE and DELETE statements may be positioned—targeted at one and only one row—or searched—targeted at zero or more rows. Strictly speaking, a positioned update can occur only in the context of the current row of a cursor operation (**WHERE CURRENT OF <cursor-name>**), while a searched update, optionally limited by the search conditions in a WHERE clause, occurs in all other contexts.

Most dataset component interfaces emulate the positioned update or delete by using a searched update with a uniquely-targeted WHERE clause. These unidirectional or scrolling dataset classes maintain a “current row buffer” that stores or links to the column and key values for the row that the user task has selected for an operation. When the user is ready to post an UPDATE or DELETE request, the component constructs a searched UPDATE or DELETE statement that targets one database row uniquely by using the primary key (or some other unique column list) in the WHERE clause.



Not all component interfaces have the “smarts” to detect duplicate rows in supposedly “live” buffers. In these products, it is up to the developer to ensure uniqueness, or to find some other way to prevent the application from unexpectedly updating multiple rows.

Using the UPDATE statement

The UPDATE statement has the following general form:

```
UPDATE table-name | view-name
SET column-name = value [,column-name = value ...]
[WHERE <search conditions> | WHERE CURRENT OF cursor-name]
[RETURNING <list-of-values> [INTO <target-list>]]
```



For searched updates, if a WHERE clause is not specified, the update will be performed on every row in the table.

The SET clause

```
SET column-name = value [,column-name = value ...]
```

The SET clause is a comma-separated list that identifies each column for modification, along with the new value to be assigned to it. The new value must be of the correct type and size for the column's definition. If a column is nullable, it can also be set to NULL instead of a value.

A value can be:

- a constant value of the correct type, e.g. `SET COLUMNB = '99'`. If the column is a character type of a specified character set that is different to the character set of the connection, an update value can be forced to that special character set by including the appropriate character set introducer to the left of the constant string. A character set introducer is the character set name prefixed with an underscore symbol.

For example: `SET COLUMNY = _ISO8859_1 'fricassée'`

- the identifier of another column in the same table—provided it is of the correct type. For example, `SET COLUMNB = COLUMNX` will replace the current value of COLUMNB with the current value of COLUMNX. A character set introducer (see previous item) can be used if appropriate.
- an expression, e.g. `SET REVIEW_DATE = REVIEW_DATE + 14` updates a date column to two weeks later than its current value. For details about expressions, refer to the next chapter.
- a server context variable, e.g. `SET DATE_CHANGED = CURRENT_DATE`. Context variables are discussed and listed in Chapter 6, **About Firebird Data Types**.
- a parameter placeholder symbol, appropriate to the syntax implemented in the application code, e.g., from Delphi, `SET LAST_NAME = :LAST_NAME` or from another application interface, `SET LAST_NAME = ?`
- an internal or user-defined (UDF) function call, e.g., `SET BCOLUMN = UPPER (ACOLUMN)` uses the internal SQL function UPPER to transform the value of ACOLUMN to upper-case and store the result in BCOLUMN. For information about using functions, refer to the next chapter, **Expressions and Predicates**.

A COLLATE clause can be included when modifying character columns, if appropriate. For most character sets, it would be necessary to use one in the example `SET BCOLUMN = UPPER (ACOLUMN)`, since the default (binary) collation sequence does not usually support the UPPER() function.

For example, if ACOLUMN and BCOLUMN are in character set ISO8859_1, and the language is Spanish, the SET clause should be:

```
SET BCOLUMN = UPPER(ACOLUMN) COLLATION ES_ES
```

For details about character sets and collations, refer to Chapter 9, **Character Types**.

Value-switching in pre-2.5 versions

Take care when “switching” values between columns. Until Firebird 2.5, a clause like

```
...
SET COLUMNA = COLUMNB
```

causes the current value in COLUMNA to be overwritten by the current value in COLUMNB immediately. If you then do

```
SET COLUMNB = COLUMNA
```

the old value of COLUMNA is gone and the value of COLUMNB and COLUMNA will stay the same as they were after the first switch. To switch the values, you would need a third column in which to “park” the value of COLUMNA until you were ready to assign it to COLUMNB:

```
...
SET
  CUMMNC = COLUMNA,
  COLUMNA = COLUMNB,
  CUMMNB = CUMMNC
```

You can use expressions with SET so, if switching two integer values it is possible to “work” the switch by performing arithmetic on the two values. For example, suppose COLUMNA is 10 and COLUMNB is 9:

```
...
SET
  CUMMNB = CUMMNB + CUMMNA, /* CUMMNB is now 19 */
  CUMMNA = CUMMNB - CUMMNA, /* CUMMNA is now 9 */
  CUMMNB = CUMMNB - CUMMNA /* CUMMNB is now 10 */
...
```

Always test your assumptions when performing switch assignments! This behaviour is not compliant with the standard but it is the way it works in all Firebird versions prior to v.2.5.

Corrected value-switching

If you have application code that was written for any Firebird version up to and including v.2.1.x and you are migrating it to use with v.2.5 or higher, you will need to be aware of how the value switching works in the corrected versions:

```
...
SET COLUMNA = COLUMNB
```

causes the current value in COLUMNA to be assigned the current value in COLUMNB. The original value of COLUMNA is retained. If you then do

```
SET COLUMNB = COLUMNA
```

it will be the original value (OLD.COLUMNA) that is assigned to COLUMNB.



*For compatibility with legacy code, the 2.5+ server can be temporarily configured to “misbehave” like older versions by setting the configuration parameter **OldSetClauseSemantics** to 1 in firebird.conf. It will affect all application and PSQL code that is run on that server, until that parameter disappears in a future release.*

Updating BLOB columns

Updating a BLOB column actually replaces the old BLOB with a completely new one. The old BLOB_ID that was the reference key for the BLOB that was stored previously does not survive the update. Also:

- It is not possible to update a blob by concatenating another blob or a string to the existing blob.
- A text (sub_type 1) blob can be set using a string as input. For example, MEMO in the next example is a text blob:

UPDATE ATABLE

SET MEMO = 'Friends, Romans, countrymen, lend me your ears: I come not to bury Caesar, but to praise him.';



Remember that, although blobs are unlimited in size, string types are limited to 32,765 bytes (VARCHAR) or 32767 bytes (CHAR)—that is bytes, not characters. Special care is needed with concatenations and multi-byte character sets.

Updating ARRAY columns

It is not possible to update ARRAY columns in dynamic SQL at all. To update arrays, it is necessary to implement a custom method in application or component code, that calls the API function *isc_array_put_slice*.

In embedded applications (ESQL) it is possible to construct a precompiled SQL statement to pass slices of arrays to update (replace) corresponding slices in stored arrays.

Column defaults

Column DEFAULT constraints are never considered when UPDATE statements are processed.

The WHERE clause

An UPDATE statement can have one WHERE clause, to target the rows that are to be modified by the request. It can comprise any number of search conditions, separated by AND, OR or existence operators.

To target a single row, the WHERE clause must include the primary key of the row, or another unique key column or combination. If a unique key combination is not present, the request will update all of the rows where the conditions are met. An UPDATE statement with no WHERE clause is valid and will modify every row in the table.

The RETURNING clause

If the UPDATE statement updates a single row, it can include the optional RETURNING clause specifying a set of one or more columns, constants or expressions to be returned to the caller, on completion, as a one-row set. The fields return can comprise anything useful, including the NEW and OLD context values as they stood after execution of any BEFORE UPDATE triggers.

Any columns named will return the NEW value, i.e., only OLD.<column-name> can return the old value. A set of NULL is returned if the statement does not find a row to update.



The RETURNING clause is not available in versions older than v.2.1.

Example

```
UPDATE BIRDS
SET
  GENUS = 'Psittaciformes ',
  SPECIES = 'Nestor notabilis'
WHERE COMMON_NAME = 'Kea'
RETURNING ID, OLD.GENUS, SPECIES
```

INTO <target-list>

Available only for an UPDATE statement that is called inside a PSQL module. The <target-list> argument is a comma-separated list of local variables prefixed with colons, context variables or constants.

For more information about PSQL variables and the INTO syntax, refer to Chapter 28, **Procedural SQL—PSQL**.

UPDATE OR INSERT

UPDATE OR INSERT provides one way to use a single statement to apply a set of data that will update a row, if it exists, or insert a row if it does not. By default, the existence test is performed against the primary key but it need not be. The optional keyword MATCHING can be used to supply test conditions for another column or set of columns.

If the MATCHING sub-clause is not used, or is used with a unique column or set, a RETURNING clause can be applied to return a set of values to the requestor. Using a RETURNING clause with MATCHING conditions that find multiple rows will cause an exception.



This syntax is not available in versions 2.0.x or lower.

Syntax

```
UPDATE OR INSERT INTO
{table-name | view-name} [( <list-of-columns> )]
VALUES ( <list-of-values> )
[MATCHING ( <list-of-columns> )]

[RETURNING <values> [INTO <list-of-variables>]]
```

Syntax elements

The required and optional elements are:

table-name | view-name

As with the INSERT and UPDATE statements, the target object can be only a single persistent table or an updatable view.

<list-of-columns>

A comma-separated list of columns, as for the input clause an INSERT statement. Any read-write column in the target object can be included, once and only once. If the request results in an update, rather than an insert, each member of the input set has the same effect as left-side part of the assignment sub-clause of a SET clause in an UPDATE statement.



If applying this syntax to a view, make sure that all of the columns in the list are updatable

VALUES (<list-of-values>

A comma-separated list of columns, as for the destination clause of an INSERT statement. If the request results in an update, each member of the input set has the same effect as right-side part of the assignment sub-clause of a SET clause in an UPDATE statement.

MATCHING (<list-of-columns>

This is the explicit search clause for the row that is to be updated, if it exists. It is optional for a target that is a table with a primary key; for an unkeyed table or a view, it is mandatory. The comma-separated search keys are ANDed, i.e., to be found, a row must have matches in all of the columns listed in the MATCHING clause.

NULL

Matches are determined with IS NOT DISTINCT, not with the equality operator. Matching NULL with NULL thus returns True, not False (unknown) as occurs with an equality test. For more information, see the topic IS [NOT] DISTINCT FROM in the next chapter.

RETURNING <values> []

Refer to the topic The RETURNING clause in the previous section. It will cause an exception if the UPDATE OR INSERT statement finds more than one row to update.

Example

```
UPDATE OR INSERT INTO BIRDS (COMMON_NAME, GENUS, SPECIES)
VALUES ('Kea', 'Psittaciformes', 'Nestor notabilis')
MATCHING (COMMON_NAME)
RETURNING ID, OLD.GENUS, OLD.SPECIES;
```

INTO <target-list>

As with the RETURNING clause used elsewhere, the INTO sub-clause is available in procedural language definitions as an option for returning the list of specified values into PSQL variables.

MERGE

The MERGE verb, available in v.2.1 and higher versions, merges data into a table or updatable view from a variety of data sources—not just persistent tables, views and stored procedures but also derived tables and global temporary tables (GTTs). How the source data are written to the target table—even whether they are written at all—is determined by conditional WHEN clauses.



*Views, derived tables and global temporary tables are discussed in Chapter 23, **Views and Other Run-time Set Objects**.*

Syntax

```
MERGE INTO {target-table-name | target-view-name} [[AS] target-alias]
USING {source-table-name | source-view-name | (<runtime-object>)} [[AS] source-alias]
ON <search-criteria>
WHEN MATCHED THEN UPDATE SET column-name = value [, column-name = value ...] (Cont.)
```


WHEN NOT MATCHED THEN INSERT [(*<list-of-columns>*)] VALUES (*<list-of-values>*)

Syntax elements

The required and optional elements are:

Target

A target table or view name is a required element, the name of the table or updatable view with which the data are to be merged.

[AS] *<target-alias>*

An alias for the named target. The AS keyword may be omitted. Use of an alias for the source is optional unless it is necessitated by complexity, such as a self-referencing or recursive operation.

Source

The source for the data can be

- a persistent or global temporary table
- a view
- a derived table

[AS] *<source-alias>*

An alias for the source, required if the source is a derived table, available if needed otherwise. The AS keyword may be omitted.

Search criteria

The search criteria are a matter of supplying a predicate for matching the specified data elements of the source data with the columns specified as corresponding in the target set.

WHEN [NOT] MATCHED clause

The WHEN clause sets the action to be taken according to the search results. At the time of this writing, you can set an action for one or both of WHEN MATCHED and WHEN NOT MATCHED, but not more than one of either:

- WHEN MATCHED results in the UPDATE action being executed on all rows in the target that meet the criteria.



Firebird's implementation as at v.2.5 is not standards-compliant with respect to the outcome if multiple source records match the same record in the target table. The standard specifies that such a case should fail with an exception. Firebird allows the action clause to be executed for each match, with the successive changes overwriting the preceding ones.

- WHEN NOT MATCHED results in the specified action being executed as many times as there are rows in the source table that do not match the criteria.

Examples: In the first example, a unique value in the source table is used to search for the same unique value in the target. If it is found, the WHEN MATCHED action is executed (a column is updated). If it is not found, a row is inserted into the target table using values from columns in the source set.

```
MERGE INTO PRODUCTS p
  USING NEW_PRODUCTS np
```

```

ON np.BARCODE = p.BARCODE /* a UNIQUE column in both tables */
AND np.PRODUCT_STATUS = 'Available'
WHEN MATCHED THEN
    UPDATE SET PRODUCT_STATUS = np.PRODUCT_STATUS
WHEN NOT MATCHED THEN
    INSERT (PRODUCT_DESCRIPTION, BARCODE, COLOUR, PRODUCT_STATUS, FIRST_STOCK_DATE)
    VALUES (np.PRODUCT_DESCRIPTION, np.BARCODE, np.COLOUR, np.PRODUCT_STATUS,
    CURRENT_TIMESTAMP);

```

The second example shows how MERGE could be used to concoct a “conditional insert” in dynamic DML by using just the WHEN NOT MATCHED clause:

```

MERGE INTO BIRDS b
USING RDB$DATABASE
    ON b.COMMON_NAME = 'Kea'
WHEN NOT MATCHED THEN
    INSERT (COMMON_NAME, GENUS, SPECIES)
    VALUES ('Kea', 'Psittaciformes', 'Nestor notabilis');

```

It is a trick that will not work if the source set has more than one row. Before MERGE entered the lexicon, the only way to do a conditional insert was to write a parameterised stored procedure. It remains the better approach, since the trick demonstrated here does not allow *parameters*. Parameters in a fully-fledged DML implementation form one of its of its most powerful features. They are introduced later in this chapter.

The DELETE Statement

The DELETE query is used for deleting whole rows from a table. SQL does not allow a single DELETE statement to remove rows in more than one table.

A DELETE query that modifies only the current row of a cursor is known as a positioned delete. One which may delete multiple rows is known as a searched delete.

The DELETE statement has the following general form:

```

DELETE FROM table-name
[WHERE <search predicates> | WHERE CURRENT OF cursor-name]

```



If a WHERE clause is not specified, every row in the table will be deleted.

The following statement deletes all rows in a table that have a certain flag set:

```

DELETE FROM NEW_PRODUCTS
    WHERE PRODUCT_STATUS = 'SCRATCH';

```

EXECUTE Statements

Although the EXECUTE verb belongs in the DML lexicon, it is not supported in dynamic DML. In the ESQL language set, it is used in embedded applications to execute a previously prepared dynamic SQL statement—in contrast with regular DML statements in ESQL, which are precompiled and hence are not prepared at run-time.

Don't confuse this EXECUTE statement with other syntaxes that include the EXECUTE verb:

- The EXECUTE PROCEDURE statement (introduced below in the topic *Queries That Execute Server-side Code*) is used for invoking executable stored procedures.
- EXECUTE BLOCK is a construct that evolved in dynamic DML from the “2” series onward. It encapsulates and executes a block of procedural SQL (PSQL) code “on-the-fly” within a dynamic statement. For details and guidance for its usage, refer to the topic *Run-time PSQL* in Chapter 29, *Stored Procedures and Executable Blocks*.
- EXECUTE STATEMENT is a statement syntax that is provided in PSQL extensions (see *Execute Statement* in Chapter 28, *Procedural SQL—PSQL*). It is not available in DML.

Using Parameters

The queries we have looked at so far in this chapter are “non-parameterised”: the values for the search criteria are supplied as literal values. For example, in this query

```
SELECT MAX(COST * QUANTITY) AS BEST_SALE
FROM SALES
WHERE SALES_DATE > '31.12.2011' ;
```

the query is asked to operate on all of the rows in SALES that have a SALES_DATE later than the last day of 2011, which is offered as a date literal.

The Firebird API allows a statement to be submitted to the server as a kind of template with placeholders for the values. The client request asks for the statement to be prepared—by obtaining syntax and metadata validation—without actually executing it. The placeholders are called *parameters*.

Once the request is prepared, the application can proceed to assign values to the parameters one or many times, just before each execution. This capability is sometimes referred to as *late binding*. Application interfaces vary in the way they surface late parameter binding to the host language. Depending on the interface you use for application development, a parameterized version of the last example may look something like the following:

```
SELECT MAX(COST * QUANTITY) AS BEST_SALE
FROM SALES
WHERE SALES_DATE > ? ;
```

The replaceable parameter in this example—marked by the ‘?’ symbol—allows the application to prepare the statement and capture from the user (or some other part of the application) the actual date on which the query is to be filtered. The same query can be run repeatedly, within the same transaction or in successive transactions, with a variety of dates, without needing to be prepared again.

The API “knows” the order and format of multiple parameters because the application interface passes descriptive structures—XSQLDAs—that contain an array of SQLVARs, the variable descriptors that describe each parameter and other data describing the array as a whole.

Delphi and some other object-oriented API implementations use the methods and properties of classes to hide the machinery used for creating and managing the raw statement and the descriptors internally. Other interfaces code the structures closer to the surface. The statement is passed to the server with a format like the following:

```
INSERT INTO DATATABLE(DATA1, DATA2, DATA3, DATA4, DATA5,...more columns)
VALUES (?, '?', '?', ?, ?, ...more values);
```

If parameters are implemented in your application programming language, or can be, then it is highly recommended to make use of them.

Note for Delphi® and FreePascal users

The ObjectPascal language for Delphi®, originally developed by Borland/Inprise in parallel with a phase in the evolution of Firebird's InterBase® ancestors, surfaces parameters as named objects that can be incorporated in the SQL request, in lieu of the '?' placeholders. The convention used mimics that used in PSQL to refer to the values of local variables in SQL statements and in ESQL to pass host variables, prefixing the name with the colon symbol. FreePascal, an open source, cross-platform emulation of ObjectPascal, adheres to the same convention.

In ObjectPascal/FreePascal, the simple example above would be expressed in the SQL property of a data access object as follows:

```
SELECT MAX(COST * QUANTITY) AS BEST_SALE
FROM SALES
WHERE SALES_DATE > :SALES_DATE ;
```

Once the Prepare call has validated the statement and passed the metadata description back to the client application, the data access object then lets the latest value be assigned to the parameter using a local method that converts the value to the format required by Firebird:

```
aQuery.ParamByName ('SALES_DATE').AsDate := ALocalDateVariable;
```

Batch operations

It is a very common requirement to perform a task that inserts, updates or deletes many rows in a single operation. For example, an application reads a data file from a device and massages it into INSERT statements for posting to a database table, or a replication service processes batches of changes between satellite and parent databases. Many similar statements are posted within a single transaction, usually by means of a prepared statement and late-bound parameters.

Batch inserts

Our simple example above could be the statement for a batch insert where an application is reading data in from some external structure:

```
INSERT INTO DATATABLE(DATA1, DATA2, DATA3, DATA4, DATA5,...more columns)
VALUES (?, '?', '?', ?, ?, ...more values);
```

When prepared, the statement is ready to accept an unspecified number of sets of value assignments in a procedure that loops at the client side.

Often, a stored procedure will be a more elegant way to set up a repeatable batch operation, especially if there is a requirement to transform data en route to the database table or to insert rows into multiple tables. The application can pass each set of values for the parameters directly to the prepared procedure, calling EXECUTE PROCEDURE each time it iterates; or it can pass parameters once to the procedure and have the procedure itself iterate through a source set or an external table.

Preventing slowdown on batches

Two behaviors can cause huge batches to get progressively slower during execution.

- One is the maintenance of indexes, which adds work to the task and also tends to expand the distribution of the index pages.
- In batch updates or deletes, another performance inhibitor is the accumulation of back versions, the deltas of the old rows that are marked for obsolescence once the update or delete operations are committed. Back versions do not become available for garbage collection until the transaction completes. In transactions involving back versions numbering in tens to hundreds of thousands, or more, garbage collection may never win its battle to clean out obsolete versions. Under these conditions, only a backup and restore will sanitize the database completely.

Deactivating indexes

A strategy for mitigating batch slowdown when other users are not accessing the input table is to deactivate the secondary indexes. This is simple to do:

```
ALTER INDEX index-name INACTIVE;
```

When the large batch operation is finished, reactivate and rebuild the indexes:

```
ALTER INDEX index-name ACTIVE;
```

Partitioning large batches

Unfortunately, you cannot disable the indexes that enforce primary and foreign keys without dropping the actual constraints. Often, it will be impracticable to drop constraints because of chained dependencies. For this reason and others, judicious partitioning of the task is recommended to help offset both the deterioration of index geometry and the generation of an unnecessarily large numbers of new database pages.

When posting large numbers of inserts or updates in batch, it is preferable to partition them into groups and commit work about every 5000-10,000 rows. The actual size of an optimal partition will vary according to the sizes of rows and the database page size. Optimal groups may be smaller or larger than this range.



When you need to partition huge batches, a stored procedure is often the way to go. You can make use of local counter variables, event messages and return values to keep completing procedure calls in synch with the calling client.

Queries That Execute Server-side Code

Firebird supports two styles of stored procedures: *executable* and *selectable*. For details about the differences in technique for writing and using these procedure styles, refer to Chapters 28 and 29.

Executable procedures

In DSQL, EXECUTE PROCEDURE executes (calls) an executable stored procedure—that is, a stored procedure designed to perform some operations on the server, optionally returning a single line of one or more return values. The statement for executing such a procedure has the following general format:

```
EXECUTE PROCEDURE procedure-name [<list of input values>]
```

The following simple example illustrates calling an executable procedure that accepts two input arguments, performs some operations on the server and exits:

```
EXECUTE PROCEDURE DO_IT(49, '25-DEC-2011');
```

In applications, it is more powerful to use parameters (see below) in query statements that execute stored procedures. For example:

```
EXECUTE PROCEDURE DO_IT(?, ?);
```

or, in ObjectPascal and some other language environments, with named parameters:

```
EXECUTE PROCEDURE DO_IT(:IKEY, :REPORT_DATE);
```



The API provides the means for retrieving the structure containing the row of return values. Most of the language-specific interpretation layers provide the means to store it.

Selectable procedures

A selectable stored procedure is capable of returning a multiple-row set of data in response to a specialized SELECT statement form, as follows

```
SELECT <list of output columns>
  FROM procedure-name [( <list of input values> )]
[WHERE <search predicates>]
[ORDER BY <list drawn from output columns>]
```

In the following PSQL fragment, a stored procedure is defined to accept a single key as an input argument and return a set of rows. The RETURNS clause defines the output set:

```
CREATE PROCEDURE GET_COFFEE_TABLE (IKEY INTEGER)
RETURNS (
  BRAND_ID INTEGER,
  VARIETY_NAME VARCHAR(40),
  COUNTRY_OF_ORIGIN VARCHAR(30))
AS .....
```

The client application requests the output set from the stored procedure as follows:

```
SELECT BRAND_ID, VARIETY_NAME, COUNTRY_OF_ORIGIN
  FROM GET_COFFEE_TABLE(5002);
```

The same example, with the input argument parameterized:

```
SELECT BRAND_ID, VARIETY_NAME, COUNTRY_OF_ORIGIN
  FROM GET_COFFEE_TABLE(?);
```

or, with a named parameter, if the language interface allows it:

```
SELECT BRAND_ID, VARIETY_NAME, COUNTRY_OF_ORIGIN
  FROM GET_COFFEE_TABLE(:IKEY);
```

DML operations and state-changing events

Firebird incorporates some features that can be implemented in database design to respond to DML operations that change the state of data, viz., the posting of INSERT, UPDATE and DELETE statements.

Referential integrity action clauses

Referential integrity triggers are modules of compiled code created by the engine when you define referential integrity constraints for your tables. By including `ON UPDATE` and `ON DELETE` action clauses in `FOREIGN KEY` constraint declarations, you can specify one of a selection of predefined response actions that will occur when these DML events execute. For details, refer to Chapter 17, *Referential Integrity*.

Custom triggers

With custom triggers—those you write yourself using the PSQL language—you have the capability to specify exactly what is to happen when the server receives a request to insert, change or delete rows in tables. Custom triggers can be applied not just to the update and delete events, but also to inserts. Triggers can include exception handling, feedback and custom query plans.

DML event phases

Trigger syntax splits the custom DML actions into two phases: the first phase before the event, the second after it.

- The `BEFORE` phase makes it possible to manipulate and transform the values that are input by the DML statement and to define defaults with much more flexibility than is permitted by the standard `SQL DEFAULT` constraint. The `BEFORE` phase completes before any column, table or integrity constraints are tested.
- In the `AFTER` phase, response actions can be performed on other tables. Usually, such actions involve inserting to, updating or deleting from these other tables, using the `NEW` and `OLD` variables (see below) to provide the context of the current row and operation. The `AFTER` phase begins after all of the owning table's constraints have been applied. `AFTER` triggers cannot change values in the current row of the owning table.

Trigger events and phases

Table 19.1 The six phase/events of custom triggers

BEFORE INSERT	BEFORE UPDATE	BEFORE DELETE
AFTER INSERT	AFTER UPDATE	AFTER DELETE

Multi-action triggers

You can optionally write triggers with conditioned logic to roll all of the events (insert, update and delete) for one phase—`BEFORE` or `AFTER`—into one trigger module. Briefly, you can write a `BEFORE` or `AFTER` trigger that comprehends any two or all or `INSERT OR UPDATE OR DELETE`.

NEW and OLD context variables

The server makes two sets of context variables available to triggers. One consists of all the field values of the current row, as they were in the current row just before the row was last posted. The identifiers for this set consist of the word “`OLD`.” prefixed to any column identifier. In the same manner, all of the new values are “`NEW`.” prefixed to any column

identifier. Of course, “OLD.” is meaningless in an insert trigger and “NEW.” is meaningless in a delete trigger.

Multiple triggers per event

Another useful option is the ability to have multiple triggers for each phase/event combination. CREATE TRIGGER syntax includes the keyword POSITION, taking an integer argument which can be used to set the zero-based firing order of multiple triggers within a phase.

For detailed instructions, syntax and language extensions for creating triggers, refer to Chapter 30.

Query Plans and the Optimizer

This topic looks at the Firebird optimizer subsystem and the strategies it uses to devise the query plans that the engine will use for SELECTs and subqueries at execution time. We take a look at query plan syntax and some possibilities for presenting your own custom plan to the engine.

Plans and the Firebird query optimizer

To process a SELECT statement or a search condition, the Firebird engine subjects the statement to a set of internal algorithms known as the “query optimizer”. Each time the statement is prepared for execution, the optimizer computes a retrieval plan.

The plan

The query plan provides a kind of road map that tells the engine the least costly route through the required process of searching, sorting and matching to arrive at the requested output. The more efficient a plan the optimizer can construct, the faster the statement will begin returning results.

A plan is built according to the indexes available, the way indexes or streams are joined or merged and a method for searching (“access method”).

When calculating a plan, the optimizer will consider every index available, choosing or rejecting indexes according to their cost. It takes other factors into account besides the existence of an index, including the size of the table and, to a degree, the distribution of distinct values throughout the index. If the optimizer can determine that an index would incur more overhead than stepping row-by-row through a stream, it may choose to ignore the index in favor of forming an intermediate sorted stream itself or processing the stream in natural order.

Plan expressions

Firebird provides the same plan expression syntax elements for setting plans in SQL as it provides to the engine. Understanding the plans generated by the optimizer can be very useful, both for anticipating how the optimizer will resolve the task to be done and for using as a basis for custom plans.

Plan expression syntax

Should you need to “roll your own” plan, you can add it to the end of the query specification, using the optional `PLAN` clause:

```
<query-specification>
  PLAN <plan_expression>
```

This syntax allows specification of a single relation or a join of two or more relations in one pass. Nested parentheses can be used to specify any combination of joins. The operations pass their results from left to right through the expression.

Symbols

In the notation used here, the round brackets and commas are elements of the syntax. Curly braces, square brackets and pipe symbols are not part of the syntax—as with earlier syntax layouts, they indicate, respectively, mandatory and optional phrases and mutually exclusive variations.

```
plan-expression := [join-type] (plan-item-list)
join-type      := JOIN | MERGE
plan-item-list := plan-item | plan-item, plan-item-list
plan-item      := table-identifier access-type | plan_expression
table-identifier := { table-identifier | alias-name } [ table-identifier ]
access-type     := { NATURAL | INDEX (index-list) | ORDER index-name }
index-list      := index-name | index-name, index-list
```

The elements

join-type can be `JOIN` or `MERGE`.

- The default join type is `JOIN`, i.e. to join two streams using an index on the right-side stream to search for matching keys in the left-side stream.
- `MERGE` is chosen if there is no useable index. It causes the two streams to be sorted into matching orders and then merged. In custom plans, it will improve retrieval speed to specify this join type for joins where no indexes are available

table-identifier identifies a stream. It must be the name of a table in the database or an alias. If the same table will be used more than once, it must be aliased for each usage and followed by the name of the table being aliased. To enable the base tables of a view to be specified, the syntax allows multiple table identifiers. Plans for views are discussed in Chapter 23.

access-type must be one of

- `NATURAL` order, which means rows are accessed sequentially in no particular order. It is the default access type and can be omitted, although, again, it is wise to include it in custom plans to aid documentation.
- `INDEX` allows one or more indexes to be specified for testing predicates and satisfying join conditions in the query.
- `ORDER` specifies that the result of the query is to be sorted (ordered) on the leftmost stream, using the index.

A *plan_item* comprises an access type and the table identifier or alias.

Understanding the optimizer

If you are unfamiliar with query plans, you are probably quite confused about how all of this syntax translates to a plan. A little later, the syntax will make more sense when we take a look at some plans generated by the optimizer. However, at this point, it will be helpful to examine how the optimizer evaluates the “materials” for its operations: the join and search conditions requested in the statement, the streams underlying the query specification and the indexes that are available.

Factors in cost evaluation

The objective of the optimizer is to establish a plan that reflects the strategy which, according to several factors, is likely to start returning the output stream fastest. The evaluation may be quite imprecise, using several variables that are no more than rough estimates. Factors considered include

- availability of an index and the selectivity of that index. The selectivity factor used in estimates is that which was read from the system tables when the database was opened. Even at startup, it may not be accurate and it may have altered due to distribution changes in operations since the last selectivity was computed.
- the cardinality (number of rows) in the table streams
- whether there are selection criteria and if so, whether an index is available or is suitable
- the need to perform sorts, both intermediate (for merges) and final (for ordering and grouping)

Streams

The term *stream*, that crops up in discussions about the optimizer, is merely a generic way to refer to a set of rows. The set might be all of the rows and columns in a table, or it might be a set from a table that is restricted in some way by column specifications and search conditions. During the evolution of a query, the engine may create new streams from contributing stream, such as an internal, sorted set or a set of subqueried values for an IN() comparison. See also *Rivers*, below.

Use of indexes

In queries, the Firebird engine can use an index for three kinds of task:

- Equality comparisons—to perform an equals, greater than, less than or STARTING WITH comparison between the value in a column and a constant. If the column is indexed, the index key is used to eliminate ineligible rows, making it unnecessary to fetch and scan unwanted rows of table data just to eliminate them.
- Matching streams for joins—if an index is available for the columns named as the join condition for the stream on one side of the join, the stream on the other side is retrieved first and its join columns are matched with the index.
- Sorting and grouping—if an ORDER BY or GROUP BY condition specifies a column that has an index, the index is used to retrieve the rows in order and a sort operation is not required.

Bitmapping of streams

When join criteria can be matched to an indexed column, the engine generates a bitmap of each stream selected. Subsequently, using AND and OR logic in accordance with the search criteria, the individual bitmapped row streams are combined into a single bitmap

that describes all of the eligible rows. When the query needs to scan the same stream in multiple passes, scanning the bitmap is much faster than revisiting the indexes.

BLR – Binary Language Representation

Underlying Firebird's SQL engine is its native BLR language and a bare-bones interpreter consisting of a lexer, a parser, a symbol table and code generator. The DSQL engine and embedded applications pass BLR to the optimizer and it is BLR, not strings, that the optimizer analyzes. It can happen that two different SQL statements are interpreted into identical BLR.

If you are curious about what BLR looks like, you can use the *qli* tool (in your Firebird bin directory) to inspect statements. Start *qli* and submit the command SET BLR to toggle on BLR display.



A language manual for qli, in PDF format, can be downloaded from some Firebird resource websites, including www.ibphoenix.com.

If you have any SELECT statements in a stored procedure or trigger, you can view them in the more familiar *isql* by submitting a SET BLOB 2 command and then querying the column RDB\$PROCEDURE_BLR in the system table RDB\$PROCEDURES.

Joins without indexes

If no index is available for a JOIN term the optimizer specifies a SORT MERGE between the two streams. A sort merge involves sorting both streams and then scanning one time through both to merge the matching streams into a river. Sorting both streams avoids the need to scan the right stream repeatedly to find matches for the left stream's keys.

Rivers

A river is a stream that is formed when two streams are merged by a join. When joins are nested, a river can become a stream in a subsequent operation. Various strategies for merging streams into rivers are compared for cost. The resulting plan captures the best strategy that the optimizer can determine for joining pairs of streams in order from left to right.

In calculating the cost of a joining strategy, the optimizer gives a weighting to the order in which streams are joined. Alternative ordering evaluations are more likely with inner joins than with outer; full joins require special processing and extra weighting.

The length of a particular river comes into account after the longest path is determined in the course of join order evaluation. The optimizer initially favors the longest join path—the maximum number of stream-pairs that can be joined directly. The favoured access method is to scan through the longest stream—ideally, the leftmost stream—and loop through shorter streams.

However, more important than the length of the join path is how rows from the right stream are read. The access type is evaluated according to the indexes available and their attributes (selectivity) compared with natural order and any sorting specifications that are in the picture.

If a better index is available for a shorter stream, choosing the longest stream as the controlling stream may be less important than the cost saving from choosing a shorter stream with a highly selective (ideally, unique) index. In this case, the eligible rows in the searched stream are retrieved in a single visit and ineligible rows are ignored.

Examples of plans

Many graphical database admin and SQL monitor tools provide the capability to inspect the optimizer’s plan when a statement is prepared. Firebird’s own *isql* utility provides two interactive commands for viewing plans.

Inspecting plans with isql

SET PLAN—In an interactive *isql* session, you can use SET PLAN, with the optional keywords ON or OFF, to toggle the option to display the plan at the head of the console output:

```
SQL>SET PLAN;  
SQL>SELECT FIRST_NAMES, SURNAME FROM PERSON  
CON>ORDER BY SURNAME;
```

```
PLAN (PERSON ORDER XA_SURNAME)  
  
FIRST_NAMES          SURNAME  
===== =====  
George              Abraham  
Stephanie           Allen
```

SET STATS—With another toggle command, you can have *isql* display some statistics about queries at the end of the data output which can be very useful for testing alternative query structures and plans:

```
SQL>SET STATS ON;  
SQL> < run your query >
```

```
PLAN..  
< output >  
Current memory = 728316  
Delta memory = 61928  
Max memory = 773416  
Elapsed time = 0.17 sec  
Buffers = 2048  
Reads = 15  
Writes = 0  
Fetches = 539
```

SET PLANONLY—An alternative toggle command, SET PLANONLY [ON | OFF], prepares the statement and displays the plan, without executing the query:

```
SQL>SET PLAN OFF;  
SQL>SET PLANONLY ON;  
SQL>SELECT FIRST_NAMES, SURNAME FROM PERSON  
CON>ORDER BY SURNAME;
```

```
PLAN (PERSON ORDER XA_SURNAME)
```

If you intend to use the optimizer’s plan as a starting point for constructing a custom PLAN clause, the expression syntax that is output is identical to that required in the

statement's plan expression. Inconveniently, there is no provision in *isql* to capture the optimizer's plan into a text file.¹



Graphical tools that display plans usually provide a copy/paste facility.

The following examples use queries that you can test yourself, using the `employee.fdb` test database that is installed in your `$firebird/examples/empbuild` directory.²

Simplest query

This query merely retrieves every row from a lookup table, in no specific order. The optimizer determines that, although an index is available (the primary key index), it has no use for it:

```
SQL>SET PLANONLY ON;
SQL> SELECT * FROM COUNTRY;
```

```
PLAN (COUNTRY NATURAL)
```

Ordered query

```
SQL>SELECT * FROM COUNTRY ORDER BY COUNTRY;
```

```
PLAN (COUNTRY ORDER RDB$PRIMARY1)
```

This is still a simple query, without joins. However, the optimizer chooses the primary key index because it provides the requested ordering without the need to form an intermediate stream for sorting.

Now, let's see what happens when we decide to order the same plain query on a non-indexed column:

```
SELECT * FROM COUNTRY ORDER BY CURRENCY;
```

```
PLAN SORT ((COUNTRY NATURAL))
```

The optimizer chooses to perform the sort by walking through COUNTRY in natural order.

Cross join

```
SQL> SELECT E.*, P.* FROM EMPLOYEE E CROSS JOIN PROJECT P;
```

```
PLAN JOIN (P NATURAL,E NATURAL)
```

The **CROSS JOIN**, which produces a generally useless set, does not use join criteria at all. It simply merges each stream on the left with each stream on the right. The optimizer has no use for an index. However, this example is useful for introducing the connection between table aliases in the join specification and those in the plan.

1. Many aspects of the plan that is constructed internally by the optimizer are neither visible in the plan shown by *isql* nor accessible via the PLAN clause syntax for custom plans. The syntax surfaces only a subset of the real plan that the query engine will follow.
2. Because the sample database has no tables without indexes, some of the examples in this topic use specially scripted non-indexed versions of the Employee, Project and Department tables, actually named Employee1, Project1 and Department1, respectively. A script to create and populate these tables for your tests would be a simple exercise. Scripting in *isql* is discussed in Chapter 24.

Aliases in the Plan

The aliases specified in the query are used in the plan printed by the optimizer. When specifying a custom plan, it is a good idea to use the same mode of table identification as is used in the query, for consistency. However, whereas the SQL parser disallows mixing table identifiers and aliases in queries, the PLAN clause accepts any mixture.³ For example, the following is accepted. Note that the optimizer keeps the table identifiers consistent with those used in the query specification:

```
SQL> SQL> SELECT E.*, P.* FROM EMPLOYEE E CROSS JOIN PROJECT P
CON> PLAN JOIN (PROJECT NATURAL,EMPLOYEE NATURAL);
```

```
PLAN JOIN (P NATURAL,E NATURAL)
```

Join with indexed equality keys

```
SELECT E.*, S.OLD_SALARY, S.NEW_SALARY
FROM EMPLOYEE E
JOIN SALARY_HISTORY S
ON S.EMP_NO = E.EMP_NO;
```

```
PLAN JOIN (S NATURAL, E INDEX (RDB$PRIMARY7))
```

This join denormalizes a one-to-many relationship—each employee has one or more salary history records. The optimizer chooses to loop over the (potentially) longer detail stream in order to search for relevant rows by the unique primary key index of the EMPLOYEE table. In this instance, either the number of rows in each table is roughly equal or the number of rows in SALARY_HISTORY does not exceed the number of rows in EMPLOYEE enough to outweigh the benefit of using the unique index to as lookup key. This is an inner join and the optimizer reasonably guesses that the right stream will determine the length of the river.

Let's look at its treatment of the same streams, this time making the join left outer:

```
SELECT E.*, S.OLD_SALARY, S.NEW_SALARY
FROM EMPLOYEE E
LEFT JOIN SALARY_HISTORY S
ON S.EMP_NO = E.EMP_NO;
```

```
PLAN JOIN (E NATURAL, S INDEX (RDB$FOREIGN21))
```

This time, one row will be returned for each row in the right stream, regardless of whether there is a matching key in the controlling stream. The length of the river has no relevance here, since outer joins are unequivocal about which table must be on the left side, to be looped over. It is the algorithm for the outer join that determines the access method, not the sizes of the streams. With EMPLOYEE on the left side, it would be impossible to form the outer join by looping over the SALARY_HISTORY table to look up in EMPLOYEE.

Because the optimizer has no choice about the order in which the streams could be joined, it simply picks the most usable index on SALARY_HISTORY.

When size matters

In the next example, table size comes into account and is not overlooked by using the unique primary key index. DEPARTMENT has 21 rows, PROJECT has 6 and the

3. Another reason it would be a good idea to keep aliasing in the query specification and the custom plan consistent is that consistency is likely to be enforced in a future version.

optimizer chooses the smaller table's foreign key index to optimize the lookup on the larger table:

```
SELECT * FROM DEPARTMENT D
JOIN PROJECT P
ON D.MNGR_NO = P.TEAM_LEADER ;

PLAN JOIN (D NATURAL,P INDEX (RDB$FOREIGN13))
```

Join with an indexed ORDER BY clause

```
SQL> SELECT P.*, E.FULL_NAME FROM PROJECT P
JOIN EMPLOYEE E
ON E.EMP_NO = P.TEAM_LEADER
ORDER BY P.PROJ_NAME ;

PLAN JOIN (P ORDER RDB$11, E INDEX (RDB$PRIMARY7))
```

The unique index on EMPLOYEE is chosen because of the filter condition implicit in the join criteria—the query will eliminate employees who are not team leaders and the unique index permits that to happen without needing to search the EMPLOYEE table. The choice of the filter index may also be influenced by the need to use the navigational index on PROJ_NAME for the sort.⁴

The optimizer chooses the index on the right side because the right stream will be as long as the left or (potentially) longer. Again, the optimizer cannot tell that the relationship is, in fact, 1:1. The PROJ_NAME column specified to order the set has a unique index, RDB\$11, created for a UNIQUE constraint, to use for the sort and the optimizer chooses it. The sort index appears first, instructing the engine to sort the left stream before it begins matching the join key from the right-hand stream.

Equality join with no available indexes

The tables in the following query are non-indexed copies of the Project and Employee tables:

```
SQL> SELECT P1.*, E1.FULL_NAME FROM PROJECT1 P1
JOIN EMPLOYEE1 E1
ON E1.EMP_NO = P1.TEAM_LEADER
ORDER BY P1.PROJ_NAME;

PLAN SORT (MERGE (SORT (E1 NATURAL),SORT (P1 NATURAL)))
```

The streams from both sides will be sorted and then merged by a single pass through both sorted streams and, finally, the river is sorted again because neither of the contributing streams has the requested sort order.

Three-way join with indexed equalities

```
SQL> SELECT P.PROJ_NAME, D.DEPARTMENT, PDB.PROJECTED_BUDGET
FROM PROJECT P
```

4. “Sorting by index” represents a misconception. ORDER in the plan instructs the engine to read the stream in out-of-storage order, i.e., to use the navigational index to retrieve the rows. The method can only operate on the looping control stream and produces a pre-ordered result. Because ORDER can be used only on the leftmost stream in the join path, any rule that forced or influenced the join order—such as an outer join that precluded the stream from being the leftmost—would take precedence and the navigational index could not be used.

```
JOIN PROJ_DEPT_BUDGET PDB ON P.PROJ_ID = PDB.PROJ_ID
JOIN DEPARTMENT D ON PDB.DEPT_NO = D.DEPT_NO;
```

```
PLAN JOIN (D NATURAL, PDB INDEX (RDB$FOREIGN18), P INDEX (RDB$PRIMARY12))
```

Because plenty of usable indexes are available, the optimizer chooses the JOIN access method. The index linking the PDB stream with Department is used to select the Department stream. When forming between the resulting river and the Project stream, the equijoin between the primary key of Project and the (potentially) longer river is able to be formed by using Project's primary key index to select from the river.

Three-way join with only one indexed equality

For this example, we use the non-indexed copies of Project and Department to demonstrate how the optimizer will use an available index where it can and will do its best to short-circuit the non-indexed equijoin conditions:

```
SQL> SELECT P1.PROJ_NAME, D1.DEPARTMENT, PDB.PROJECTED_BUDGET
FROM PROJECT1 P1
JOIN PROJ_DEPT_BUDGET PDB ON P1.PROJ_ID = PDB.PROJ_ID
JOIN DEPARTMENT1 D1 ON PDB.DEPT_NO = D1.DEPT_NO;
```

```
PLAN MERGE (SORT
(P1 NATURAL), SORT (JOIN (D1 NATURAL, PDB INDEX (RDB$FOREIGN18))))
```

In the innermost loop, the foreign key index for the equijoin between the PDB stream is chosen to select the eligible rows from the Department stream. Notice that its selection of this index has nothing to do with the foreign key which the index was constructed to support, since the referenced table is not even present in the statement.

Next, the resulting river and the Project stream are both sorted. Finally (in the outermost loop) the two sorted streams are merged by a single read through the two streams.

Queries with multiple plans

When subqueries and unions are specified in a query, multiple SELECT statements are involved. The optimizer constructs an independent plan for each SELECT statement. Take the following example:

```
SELECT
P.PROJ_NAME,
(SELECT E.FULL_NAME FROM EMPLOYEE E
WHERE P.TEAM_LEADER = E.EMP_NO) AS LEADER_NAME
FROM PROJECT P
WHERE P.PRODUCT = 'software';
```

```
PLAN (E INDEX (RDB$PRIMARY7))
PLAN (P INDEX (PRODTYPEX))
```

The first plan selects the primary key index of EMPLOYEE to look up the TEAM_LEADER code in the primary table for the subquery. The second uses the index PRODTYPEX on the PROJECT table for the PRODUCT filter, because the first key of the index is the PRODUCT column.

Interestingly, if the same query is modified to include an ordering clause, the optimizer overrules its choice to use an index for the filter and chooses instead to use a unique index on PROJ_NAME to navigate the ordering column:


```

SELECT
  P.PROJ_NAME,
  (SELECT E.FULL_NAME FROM EMPLOYEE E
   WHERE P.TEAM_LEADER = E.EMP_NO) AS LEADER_NAME
FROM PROJECT P
WHERE P.PRODUCT = 'software'
ORDER BY 1;

```

```
PLAN (E INDEX (RDB$PRIMARY7))
```

```
PLAN (P ORDER RDB$11)
```

Specifying your own plan

The expression syntax that the optimizer provides in the plan it passes to the Firebird engine is available in SQL, via the PLAN clause. It enables you to define your own plan and restrict the optimizer to your choices.

A PLAN clause can be specified for almost any SELECT statement, including those used in creating views, in stored procedures and in subqueries. PLAN clauses are also accepted in triggers, except in Firebird 1.0.x. Multiple plans can be specified independently for the query and any subqueries. However, there is no “all or none” requirement—any plan clause is optional.

A PLAN clause is generated for a SELECT from a selectable stored procedure. Since the output of a selectable stored procedure is a virtual set, any conditions will be based on NATURAL access. However, any SELECT statements within the stored procedure itself will be optimized and can be subjected to a custom plan.



Constructing a custom plan for a SELECT on a view presents its own special problems for the developer. For more information refer to the topic Custom plans in Chapter 24, Views and Other Derived Tables.

You must specify the names and usages of all indexes that are to be used.

The optimizer always creates a plan, even if a custom plan is specified. While the optimizer does not interfere with a user-supplied plan, it does check that the indexes are valid for the context. Alternative paths are discarded but, otherwise, it is “business as usual”. A custom plan does not cause the optimizer to short-circuit the aspects of plan evaluation and generation that are not surfaced in the PLAN clause syntax.

Deciding on a join order

When two or more streams have to be joined, the order in which the streams are retrieved is often more important than all of the other factors combined. Streams are retrieved in left-to-right order, the leftmost stream of a particular join pair being the “prime controller” determining the search logic for the remaining joins and merges that are connected to it.

Ideally, this controlling stream is searched once. Streams that it joins to are searched iteratively until all matching rows are found. It follows, therefore, that the stream which is costlier to retrieve should be placed to the left of the “cheaper” stream, in the controlling position. The last stream in the join list will be fetched the most times, so make sure it will be the cheapest to retrieve.

Getting the instinct

Getting the “shape” of a join specification right isn’t rocket science and, with experience, you’ll develop an instinct for it. However, it is a technique that some people struggle with. With all query specifications, but especially with multi-table queries, the secret is to design them conscientiously. If you are inexperienced with SQL, don’t rely on a CASE tool or query builder utility to teach you. It’s a classic Catch-22 situation: until you acquire the experience, you can’t judge how mindless these tools are. If you always rely on the dumb tool, you’ll never develop the instinct.

Get acquainted with the performance characteristics inherent in both the structures and content of your data. Understand the normalizations in your database and recognize the shortest routes between your tables and your output specifications. Use pencil and paper to map and mould join lists and search conditions to fit output specifications with precision and without waste.

Badly-performing queries in Firebird are often caused by poor indexes and sub-optimal query specifications. In the next part of the topic, we take a look at indexes and some of the misadventures that can befall the optimizer and the designer when indexing interferes with the efficiency of retrieval.

Optimal Indexing

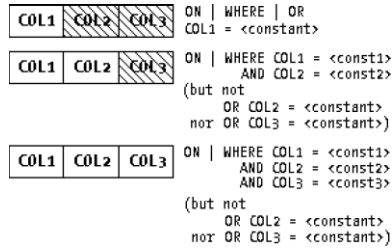
It is not a “given” that using an index on a join or a search will make the query faster. There are actually aspects of metadata structure and indexing that cause selection of some indexes to kill performance by comparison with a natural scan.

Duplicated or overlapping indexes may interfere with the optimizer’s ability to use other indexes. Drop any indexes that duplicate the index created automatically for a primary or foreign key or for a unique constraint. Composite primary and foreign key pairs, especially those that are long or carry semantic content, can make queries vulnerable to poor performance, wrong index choices and, not least, human error. When designing tables, consider using surrogate keys to separate the “meaningfulness” of columns for searching and ordering from the functional purpose of the formal key for joins.

Compound indexes

Compound (composite, complex) indexes are those that comprise more than one key column. A well-considered compound index may help speed queries and searches on complex ANDed conditions, especially when the table is large or a searched column on its own has low selectivity.

With compound indexes, *degree* (the number of elements and their relative left-to-right positions) is important. The optimizer can make use of an individual column of a compound index—or a subgroup of columns—in search, join or ordering operations, provided other columns not involved in the operation do not “obscure” it to the left or interrupt the left-to-right sequence. Figure 19.1 illustrates the possibilities available to the optimizer when evaluating a compound index on (COL1, COL2, COL3):

Figure 19.1 Availability of partial index keys

If a compound index can be used to advantage in place of one or more single column indexes, it makes sense to create and test it. There is no benefit in creating compound indexes for groups of OR conditions—they will not be used. Don't be tempted to create a compound index with the expectation that "one index will fit all cases". Create indexes for identified needs and be prepared to drop any that don't work well.

In practice, it is essential to consider the usefulness of any compound indexes in terms of the most likely output requirements, on a case-by-case basis. The more you allow redundant compound indexes to proliferate, the higher the likelihood that you will defeat the optimizer and get sub-optimal plans. The key elements of compound indexes often overlap with those of other indexes, forcing the optimizer to choose one from a range of two or more competing indexes. The optimizer can not be guaranteed to choose the best one in every case and it may even decide to use no index in cases where it cannot determine a winner.

Confirm any assumptions you make about the effect of adding a compound index, by testing not just the query in question but all regular or large query tasks that involve one or more of the same columns.



It's not a bad idea to keep a log of the index structures you have tested, along with a record of their effects—it helps to keep the temperature down in the test lab!

Single-column indexes

Single-column indexes are much more flexible, because they can be used in combination. They are preferred for any conditions that have no special need for a compound one and they are needed for each searched column that is part an ORed condition.

Natural ordering

Don't be upset to see NATURAL in a query plan and don't be afraid to use it when testing custom plans. Natural order specifies a top-to-bottom scan through the table, page by page. It is frequently faster for some data, especially matches and searches on columns for which an index would be very unselective and on tables where the number of rows is small and fairly static.

Fooling the optimizer

In situations where the optimizer chooses an index that you want it to ignore, you can trick it into ignoring it by adding a dummy OR condition. For a simple example, suppose you have a WHERE clause specifying a column that has a poorly selective index that you really want to keep, perhaps to enforce a referential integrity constraint:

```
SELECT ...
WHERE PARENT_COUNTRY = 'AU'
```

...

If this database is deployed in Australia (country code 'AU') the selectivity of PARENT_COUNTRY in this database may be so low as to kill performance but you are stuck with a mandatory index. To have the optimizer ignore the index, change the search predicate to

```
SELECT ...
WHERE PARENT_COUNTRY = 'AU' OR 1=1
...
```

Housekeeping indexes

Unlike many other relational database systems, Firebird never has need of a full-time DBA with an armory of algorithms for keeping databases running smoothly. For the most part, well-kept Firebird databases just “keep on keeping on”.

Index health does play an important part in the performance of a database. It is important to recognize that indexes are dynamic structures that, like moving parts in an engine, need to be “cleaned and lubed” from time to time.

Indexes are binary structures which may become distributed over many pages after major operations on big tables, especially if general database housekeeping is neglected. Indexes can be tuned in a number of ways to restore performance to optimal levels.

- Rebuilding an index will restore the balance of its tree structure by removing entries made obsolete by deletions and redistributing branches created by successive insertions. The tool for switching the state of an index between active and inactive is the ALTER INDEX statement—refer to the topic *Making an Index Inactive* in Chapter 16.
- A complete rebuild of an index from scratch, by dropping and recreating it in a pristine state, may improve the performance of an old index on a very large or dynamic table.
- Restoring a database from a *gbak* backup also recreates pristine indexes.

Index selectivity

Broadly, the selectivity of an index is an evaluation of the number of rows that would be selected by each index value in a search. A unique index has the highest possible selectivity, because it can never select more than one row per value; while an index on a Boolean has almost the lowest.

Indexing a column that, in production conditions, will store a single value predominantly is worse than not indexing the column at all. Firebird is quite efficient at building bitmaps for non-indexed sorts and searches.

Measuring selectivity

The selectivity of a unique index is 1. All non-unique indexes have a value lower than 1. Taking n as the number of distinct occurrences of the index value in the table and c as the number of rows in the table, you can calculate selectivity (s) as

$$s = n/c$$

The smaller the number of distinct occurrences, the lower the selectivity. Indexes with higher selectivity perform better than those with lower selectivity.

The Firebird optimizer looks up a factor for calculating selectivity when a table is first accessed and stores it in memory for use in calculating plans for subsequent queries on that

table. Over time, the initially calculated factors on frequently updated tables gradually become outdated, perhaps affecting the optimizer's index choices in extreme cases.

Recalculating selectivity

Recalculating index selectivity updates a statistical factor stored in the system tables. The optimizer reads this just once when choosing a plan—it is not highly significant to its choices. Frequent, large DML operations don't necessarily spoil the distribution of distinct index key values. If the indexing is reasonable, the “demographics” of value distribution may change very little.

Knowing the most accurate selectivity of an index has its greatest value to the developer. It provides a metric to assist in determining the usefulness or otherwise of the index.

If the efficacy of a plan degrades over time because large numbers of inserts or changes to the key column(s) that change the distribution of index key values, the query may slow down gradually. Any index whose selectivity drops dramatically over time should be dropped because of its effect on performance.

Understanding and dealing with a rogue index that deteriorates with table growth, to the extent that it interferes with plans, is an important part of database tuning. However, the most crucial effect of using an index that intrinsically has very low selectivity has nothing to do with the optimizer and everything to do with index geometry.

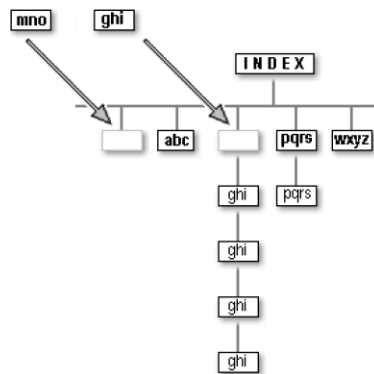
Why low selectivity can be a killer

Firebird builds binary trees for indexes. It stores these structures on index pages, which are nothing more special than pages that are allocated for storing index trees. Each distinct value in an index segment has its own node off the root of the tree. As a new entry is added to an index, it is either

- placed into a new node if its value does not already exist
- or
- stacked on top of any existing duplicate values

The diagram below illustrates this binary mechanism in simplified form:

Figure 19.2 Building a binary index tree



When duplicate values arrive, they are slotted into the first node at the front of the “chain” of other duplicates—this is what is occurring with value “ghi” in the diagram. This structure is referred to as a *duplicate chain*.

Duplicate chains

A duplicate chain⁵, per se, is fine—all non-unique indexes have them. An update of the segment value, or the deletion of a row, is very costly when the chain of duplicates is very long. One of the worst things you can do to a Firebird database is define a table with a million rows, all with the same key value for a secondary index, then delete all those rows. The last duplicate stored appears first in the list, and the first stored is last. The deletion ordinarily takes the first row stored first, then the second, and so on. The index walking code has to run through the whole duplicate chain for each deletion, always finding the entry it wants in the very last position. To quote Ann Harrison, “It churns the cache like nothing you ever saw”.

The cost of all that churning and roiling is never borne by the transaction that deletes or updates all the rows in a table. Updating a key value or deleting a row affects the index later, when the old back version is garbage collected. The cost goes to the next transaction that touches the row, following the completion of all of the transactions that were active when the update or delete occurred.

Index toolkit

- For finding out about the selectivity and other meaningful characteristics of indexes, use *gstat*, a data statistics analyzer. Later in this topic, we look at what *gstat* can tell you about your indexes.
- The tool for recalculating index selectivity is the `SET STATISTICS` statement—discussed below. `SET STATISTICS` does not rebuild indexes.
- The best tool of all for cleaning up tired indexes is *gbak*, the backup and restore utility. Restoring a database from its latest backup will both rebuild all indexes and cause their selectivity to be freshly computed.

Using SET STATISTICS

In some tables, the number of duplicate values in indexed columns can increase or decrease radically as a result of the relative “popularity” of a particular value in the index when compared with other candidate values. For example, indexes on dates in a Sales system might tend to become less selective when business starts to boom.

Periodically recomputing index selectivity can improve the performance of indexes that are subject to significant swings in the distribution of distinct values. `SET STATISTICS` recomputes the selectivity of an index. It is a statement which can be run from an *isql* interactive session. To run this statement, you need to be connected to the database and logged in as the user that created the index or a user with `SYSDBA` privileges.

Syntax `SET STATISTICS INDEX name;`

The following statement recomputes the selectivity for an index in the `employee.fdb` database:

`SET STATISTICS INDEX MINSALX;`

On its own, `SET STATISTICS` will not cure current problems resulting from previous index maintenance that depended on obsolete selectivity statistics, since it does not rebuild the index.

5. This passage speaks in the voice of Ann Harrison, “the midwife of InterBase”. Twenty-five years on, Ann remains the most lucid Explicator of Curly Questions about the RDBMS that evolved into Firebird. You can read her articulate and often humorous responses regularly in the Firebird-support mailing list.



If response is getting progressively slower, old record versions are more likely to be the culprit than the condition of indexes. They bloat indexes, they bloat pages, and client transactions have to spend time removing them. The backup/restore cycle gets rid of old versions—it works when rebuilding the indexes doesn't.

Getting index statistics

Firebird provides a command-line tool that displays real-time statistical reports about the state of objects in a database. The tool produces a number of reports about what is going on in a database, especially index statistics. Refer to *Collecting Database Statistics—gstat* in Chapter 38, **Monitoring and Logging Features**.

CHAPTER

20

EXPRESSIONS AND PREDICATES

In Algebra, an expression like “ $a + b = c$ ” can be resolved as “true” or “false” by substituting constant values into a , b and c . Alternatively, given values for any two of a , b or c , we can calculate the missing value. So it is with SQL expressions because, at the simplest level, they are substitution formulae.

SQL expressions provide formal shorthand methods for calculating, transforming and comparing values. In this chapter we take a close look at the ways and means of expressions in Firebird SQL.

Expressions

Storing data in its plainest, most abstract state is what databases do. The retrieval language—in Firebird's case, usually SQL—combines with the database engine to provide an armory of expression syntaxes into which actual data can be substituted at run-time to transform these pieces of abstract data into information that is meaningful to humans.

To take a simple example, a table, `MEMBERSHIP`, has columns `FIRST_NAME`, `LAST_NAME` and `DATE_OF_BIRTH`. To get a list of members' full names and birthdays, we can use a statement containing expressions to transform the stored data:

```
SELECT
  FIRST_NAME || ' ' || LAST_NAME AS FULL_NAME,
  EXTRACT(MONTH FROM DATE_OF_BIRTH) || '/' || EXTRACT (DAY FROM DATE_OF_BIRTH) AS BIRTHDAY
FROM MEMBERSHIP
WHERE FIRST_NAME IS NOT NULL AND LAST_NAME IS NOT NULL
ORDER BY 2;
```

At the time we send this request to the server, we do not know what the stored values are. However, we know what they should be like—their semantic values and data types—and that is sufficient for us to construct expressions to retrieve a list that is meaningful in the ways we want it to be.

In this single statement we make use of three different kinds of SQL expression:

- for the first field, FULL_NAME, the concatenation operator (in SQL, the double-pipe '||' symbol) is used to construct an expression that joins two database fields into one, separating them with a space.
- for the second field, BIRTHDAY, a function is used to extract first the month and then the day of the month from the date field. In the same expression, the concatenation operator is again used, to tie the extracted numbers together as a birthday—day and month separated by a slash.
- for the search condition, the WHERE clause uses another kind of expression—a logical predicate—to test for eligible rows in the table. Rows that failed this test would not be returned in the output.

In the example, expressions were used

- to transform data for retrieval as an output column; and
- to set search conditions in a WHERE clause for a SELECT statement. The same approach can be used also for searched UPDATE and DELETE statements

Other contexts in which expressions can be used include

- to set validation conditions in CHECK constraints
- to define COMPUTED BY columns in CREATE TABLE and ALTER TABLE definitions
- to transform or create input data in the process of storing them in a table using INSERT or UPDATE statements
- to decide the ordering or grouping of output sets
- to set run-time conditions to determine output
- to condition the flow of control in PSQL modules

Predicates

A predicate is simply an expression that asserts a fact about a value. SQL statements generally test predicates in WHERE clauses and CASE expressions. ON is the test for JOIN predicates; HAVING tests attributes in grouped output. In PSQL, flow-of-control statements test predicates in IF, WHILE and WHEN clauses. Decisions are made according to whether the predicates are evaluated as true or false.

Strictly speaking, a predicate can be true, false or not proven. For SQL purposes, false and not proven results are rolled together and treated as if all were false. In effect, “if it is not true, then it is false.”

The standard SQL language has formal specifications for a number of expression operators that are recognized as necessary for constructing search predicates. A predicate consists of three basic elements: two comparable values and an operator which predicates the assertion to be tested on the pair of values.

All of the operators included in Table 20.1 below can be predicate operators. The values involved with the predicate can be simple or they can be extremely complex, nested expressions. As long as it is possible for the expressions being compared to melt down to constant values for the test, they will be valid, no matter how complex.

Take this simple statement where the equivalence operator “=” is used to test for exact matches:

```
SELECT * FROM EMPLOYEE
WHERE LAST_NAME = 'Smith';
```

The predicate is “that the value in the column LAST_NAME is 'Smith'”. Two constants (the current column value and a string literal) are compared to test the assertion that they are equal. Taking each row in the Employee table, the engine will discard any where the predicate is either false (the value is something other than 'Smith') or not proven (the column has NULL, so it cannot be determined whether it is 'Smith' or a value that is not 'Smith').

The Truth Testers

The syntax elements that test the truth or non-truth of an assertion form a no-name paradigm of “truth-testers” or “condition testers”: they all test predicates. They are:

In DDL: CHECK, for testing validation conditions

In SQL: WHERE (for search conditions), HAVING and NOT HAVING (for group selection conditions), ON (for join conditions) and the multi-conditional case testers CASE...WHEN, WHEN MATCHING, COALESCE and NULLIF

In PSQL: IF (the universal true/false tester), WHILE (for testing loop conditions) and WHEN (for testing exception codes)

Assertions

Often, the condition being tested by WHERE, IF and so on, is not a single predicate, but a cluster of several predicates, each of which, when resolved, contributes to the truth or otherwise of the ultimate assertion. The assertion might be a single predicate, or it might contain multiple predicates logically associated by AND or OR within it, which themselves might nest predicates. Resolution of an assertion to the ultimate true or false result is a process that works from the inner predicates outwards. Each “level” must be resolved in precedence order, until it becomes possible to test the overall assertion.

In the next set of search conditions, the assertion tested encloses two conditions. The keyword AND ties the two predicates to each other, causing the ultimate assertion to be that both predicates must be true in order for the whole assertion to be true:

```
SELECT * FROM EMPLOYEE
WHERE (
    (HIRE_DATE > CURRENT_DATE - 366)
    AND (SALARY BETWEEN 25000.00 AND 39999.99));
```

Rows where one assertion is true but the other is false will be discarded.

The first predicate (HIRE_DATE > CURRENT_DATE - 366) uses an expression consisting of a variable and a calculation to establish the value that is to be tested against the column value. In this case, the assertion uses a different operator—the comparison is not that the column value and the resolved value of the expression be equal, but that the column value be greater than that value.

The second predicate uses a different operator again: the BETWEEN. symbol implements the test “greater than or equal to the value on the left AND less than or equal to the value on the right”.

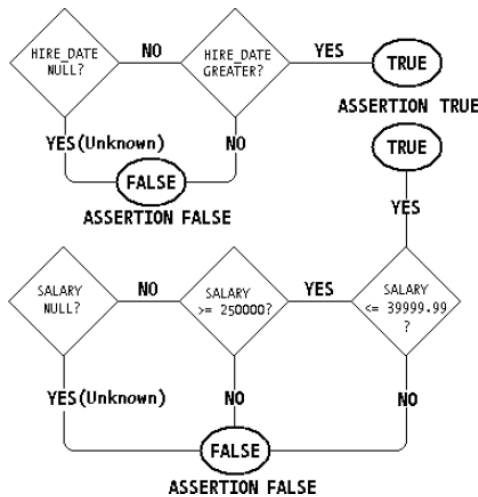
Parentheses

The parentheses here are not obligatory although, in many complex expressions, parentheses must be used to specify the order of precedence for evaluation of both expressions and the predicates. In situations where a series of predicates is to be tested, choosing to parenthesize predicates even where it is optional can be helpful for documentation and debugging.

Deciding What is True

Consider the possible results of the two predicates in the previous example.

Figure 20.1 Truth evaluation



In our example, `(HIRE_DATE > CURRENT_DATE - 366)` is tested first, because it is the leftmost predicate. If it nested any predicates, those would be evaluated first. There is no test for NULL in either of the predicates, but it's included here to illustrate that a null ("not known") encountered in the test data will cause a result of false because it can not return true. Predicates must be provably true in order to be true.

If NULL occurs in `HIRE_DATE`, testing will stop immediately and return false. The AND association causes that: logically, two predicates conjoined by AND can not be true if one is false, so there is nothing to gain by continuing on to test the second predicate.

If the data being tested is not null, the next test is whether it is true that the value is greater than the test condition. If not, testing finishes here. In other words, NULL and false have the same outcome because neither is true.

If the first predicate is true, a similar process of elimination occurs with the second predicate. Only when that is all done can the ultimate assertion be evaluated.

Elements Used in Expressions

The following table describes the symbols that can appear in SQL expressions:

Symbols	Numerics	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
---------	----------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Table 20.1 Elements of SQL expressions

Element	Description
Column names	Identifiers of columns from specified tables, representing a field value to be used in a calculation or comparison or as a search condition. Any database column can be referred to in an expression except columns of ARRAY types. (Exception: an array column can be tested for IS [NOT] NULL)
Array elements	Array elements can be referred to in an expression.
Output-only column names	Identifiers provided for run-time computed columns or as aliases for database columns
The keyword AS	Used (optionally) as a marker for the name of an output-only column in SELECT column lists
Arithmetic operators	Symbols +, −, *, and /, used to calculate numerical values.
Logical operators	Reserved words, NOT, AND and OR, used in simple search conditions, or to combine simple search conditions to form complex predicates.
Comparison operators	<, >, <=, >=, = and <> used for predicating assertions about pairs of values.
Other comparison operators	LIKE, STARTING WITH, CONTAINING, BETWEEN, IS [NOT] NULL, IS [NOT] DISTINCT
Existential operators	Predicators used for determining the existence of a value in a set. IN can be used with sets of constants or with scalar subqueries. EXISTS, SINGULAR, ALL, ANY and SOME can only be used with subqueries.
Concatenation operator	The pair of ‘single-pipe’ symbols (ASCII 124) , used to combine character strings. NOTE that ‘+’ and ‘&’ are not valid string concatenators in standard SQL.
Constants	Hard-coded numbers or single-quoted literals, such as 507 or 'Tokyo' that can be involved in calculations and comparisons or output as run-time fields
Date literals	String-like quoted expressions, that will be interpreted as date, time or date-time values in EXTRACT, SELECT, INSERT, and UPDATE operations. Date literals can be the pre-defined literals ('TODAY', 'NOW', 'YESTERDAY', 'TOMORROW') or acceptable date and time string literals as described in Chapter 8, Date and Time Types In Dialect 3, date literals usually need to be CAST as a valid date/time type when used in EXTRACT and SELECT expressions.
Internal context variables	Variables retrievable from the server, returning context-dependent values such as the server clock time or the ID of the current transaction
Subqueries	In-line SELECT statements that return single (scalar) values for run-time output or for a comparison in a predicate.
Local variables	Named stored procedure, EXECUTE BLOCK, trigger or (in ESQL) application program variables containing values that will vary during execution.
Function identifiers	Identifiers of internal or external functions, in function expressions
CAST(value AS type)	Function expressions explicitly casting a value of one data type to another data type

Element	Description
Conditional expressions	Functions predicating two or more mutually exclusive conditions for a single column, beginning with the keywords CASE, COALESCE or NULLIF. In some constructs, the keyword WHEN marks a condition.
Parentheses	Used to group expressions into hierarchies. Operations inside parentheses are performed before operations outside them. When parentheses are nested, the contents of the innermost set is evaluated first and evaluation proceeds outward.
COLLATE clause	Can be used with CHAR and VARCHAR values to force string comparisons to use a specific collation sequence.

SQL Operators

Firebird SQL syntax includes operators for comparing and evaluating columns, constants, variables and embedded expressions to produce distinct assertions.

Precedence of Operators

Precedence determines the order in which operators and the values they affect are evaluated in an expression.

When an expression contains several operators of the same type, those operators are evaluated from left to right unless there is a conflict where two operators of the same type affect the same values. When there is a conflict, operator precedence within type determines the order of evaluation. Table 20.2 lists the precedence of Firebird operator types, from highest to lowest.

Table 20.2 Operator type precedence

Operator type	Precedence	Explanation
Concatenation	1	Strings are concatenated before all other operations take place.
Arithmetic	2	Arithmetic is performed after string concatenation, but before comparison and logical operations.
Comparison	3	Comparison operations are evaluated after string concatenation and arithmetic, but before logical operations.
Logical	4	Logical operations are evaluated after all other operations: When search conditions are combined, the order of evaluation is determined by precedence of the operators that connect them. A NOT condition is evaluated before AND, and AND is evaluated before OR. Parentheses can be used to change the order of evaluation.

Concatenation Operator

The concatenation operator is composed of two pipe symbols: `||` for combining two character strings to produce a single string. Character strings can be constants or values retrieved from a column:

```
SELECT Last_name ||', ' || First_Name AS Full_Name
FROM Membership;
```

Arithmetic Operators

Arithmetic expressions are evaluated from left to right, except when ambiguities arise. In these cases, arithmetic operations are evaluated according to the precedence specified in Table 20.3. For example, multiplications are performed before divisions, and divisions are performed before subtractions.

Arithmetic operations are always performed before comparison and logical operations. To change or force the order of evaluation, group operations in parentheses.

Table 20.3 Arithmetic operator precedence

Operator	Purpose	Precedence
*	Multiplication	1
/	Division	2
+	Addition	3
–	Subtraction	4

The following example illustrates how a complex calculation would be nested to ensure that the expression would be computed in the correct, unambiguous order:

```
...
SET COLUMN_A = 1/((COLUMN_B * COLUMN_C/4) - ((COLUMN_D / 10) + 28))
```

The engine will detect syntax errors—such as unbalanced parentheses and embedded expressions that do not produce a result—but it cannot detect logic errors or ambiguities that are correct syntactically. Very complicated nested calculations should be worked out on paper. To simplify the process of composing these predicates, always start by isolating the innermost operation and working “toward the outside”, testing predicates with a calculator at each step.

Comparison Operators

Comparison operators test a specific relationship between a value to the left of the operator, and a value or range of values to the right of the operator. Every test predicates a result that can be either true or false. Rules of precedence apply to evaluation, as indicated in Table 20.4, below.

Comparisons that encounter NULL on either the left or right side of the operator always follow SQL rules of logic and evaluate the result of the comparison as NULL and return false. For example:

```
IF (YEAR_OF_BIRTH < 1950)
```

returns false if YEAR_OF_BIRTH is 1951 or NULL.

For more discussion, refer to the topic *Considering NULL*, later in this chapter.

Value pairs compared can be columns, constants, or calculated expressions and must resolve to the same data type. The CAST() function can be used to translate a value to a compatible data type for comparison.

Table 20.4 Comparison operator precedence

Operator	Purpose	Precedence
=	Is equal to, identical to	1
<>, !=, ~=, ^=	Is not equal to, does not equal	2
>	Is greater than	3
<	Is less than	4
>=	Is greater than or equal to	5
<=	Is less than or equal to	6
!>, ~>, ^>	Is not greater than	7
!<, ~<, ^<	Is not less than	8
IS DISTINCT FROM	Is not the same as	

Arithmetical predicates using these symbols need no explanation. However, string tests can use them, as well. Tests using the “=” and “<>” comparisons are clear enough. The following statement uses the “>=” operator to return all rows where LAST_NAME is equal to the test string and also where LAST_NAME evaluates as alphanumerically “greater”:

```
SELECT * FROM EMPLOYEE
WHERE LAST_NAME >= 'Smith';
```

There is no 'Smith' in Employee, but the query returns 'Stansbury', 'Steadman'...right through to 'Young'. The Employee database in the example uses default character set NONE, in which lower case characters take precedence over upper case. In our example, 'SMITH' would not be selected because the search string, 'SM' (83 + 77) evaluates as less than 'Sm' (83 + 109).

Character Sets and Collations

Alphanumeric sequence is determined at two levels: character set and collation sequence. Each character set has its own, unique rules of precedence and, when an alternative collation sequence is in use, the rules of the collation sequence override those of the character set.

The default collation sequence for any character set—that is, the one whose name matches that of the character set—is binary. The binary collation sequence is determined to ascend according the numeric code of the character in the system in which it is encoded (ASCII, ANSI, UNICODE, etc.). Alternative collation sequences typically override the default order to comply with the rules of the locale or with special rules, such as case-insensitivity or dictionary orderings.

Other Comparison Predicators

The predicators BETWEEN, CONTAINING, LIKE and STARTING WITH have equal precedence in evaluations, behind the comparison operators listed above in Table 20.4. When they conflict with one another they are evaluated strictly from left to right.

Table 20.5 Other comparison predicators

Predicator	Purpose
BETWEEN .. AND ..	Value falls within an inclusive range of values
CONTAINING	String value contains specified string. The comparison is case-insensitive
IN	Value is present in the given set †
IS [NOT] DISTINCT FROM	Tests equivalence involving one or two NULL operands
LIKE	Equals specified string, with wildcard substitution (normally % symbol is wild)
STARTING WITH	String value begins with specified string

† Most efficiently when the set is a small group of constants. The set cannot exceed 1500 members. See also Existential predicators, below.

BETWEEN

takes two arguments of compatible data types separated by the keyword AND. The lower value must be the first argument. If not, the query will find no rows matching the predicate. The search is inclusive: that is, values matching both arguments will be included in the returned set.

Example `SELECT * FROM EMPLOYEE`
 `WHERE HIRE_DATE BETWEEN '01.01.1992' AND CURRENT_TIMESTAMP;`
BETWEEN will use an ascending index on the searched column if there is one available.

CONTAINING

searches a string or “string-like” type looking for sequence of characters that matches its argument. By some internal magic, it can be used for an alphanumeric (“string-like”) search on numbers and dates. A CONTAINING search is case-insensitive.

Examples `SELECT * FROM PROJECT`
 `WHERE PROJ_NAME CONTAINING 'Map';`
It returns two rows, for the projects “AutoMap” and “MapBrowser port”.
A “string-like” usage:
 `SELECT * FROM SALARY_HISTORY`
 `WHERE CHANGE_DATE CONTAINING 93;`
In the case of this data (the Employee database) it returns all rows where the date is in 1993. That data contains no “93” dates in other centuries.

Note *In some very old releases, CONTAINING is unpredictable when used with BLOBs of more than 1024 bytes. It was a bug that was fixed in v.1.5.1.*

IN

takes a comma-separated list of constants as an argument and returns all rows where the column value on the left side matches any value in the set.

Example

```
SELECT * FROM EMPLOYEE
WHERE FIRST_NAME IN('Pete', 'Ann', 'Roger');
```

NULL can not be included in the search set.

Limitation

From a performance point of view, IN(<list-of-constants>) is not useful for more than a few values. The number of constant values that can be supplied for a set is limited to a maximum of 1499 values—possibly less, if the values are large and the size of the query string would exceed its 64 Kb limit.

IN() can be made to form its own argument list from a subquery that returns single-column rows, without the set size limitation. For details, see the topic *Existential Predicates*, later in this chapter.



It is a common “newbie” mistake to treat the predicate “IN(<single value>)” as if it were equivalent to “= <value>” because both return the same result. However, IN() does not use an index. Since equality searches are usually better optimized by an index, usage of IN as a substitute is a poor choice from the perspective of performance.

IS [NOT] DISTINCT FROM

makes it possible (“2” series onward) to compare two items of data and determine whether both are NULL. If one of the operands is NULL and the other has a value, then IS DISTINCT returns True and IS NOT DISTINCT returns False. If both operands have values, IS DISTINCT returns False unless they are equal.

Syntax <operand1> IS [NOT] DISTINCT FROM <operand2>

FROM is a required keyword.

Examples

```
SELECT
  a.AUTHOR_NAME,
  b.TITLE FROM BOOKS b
JOIN AUTHORS a
  ON a.PRIME_PUBLISHER IS NOT DISTINCT FROM b.PUBLISHER;
--
SELECT * FROM TESTS
WHERE T_STATUS IS DISTINCT FROM 'COMPLETE';
```

When IS [NOT] DISTINCT returns False, it always means “false”, never “unknown”. The table below shows how the results differ when two operands are compared using each of the operators DISTINCT, NOT DISTINCT, the equivalence operator “=” and the non-equivalence operator “<>” or “!+”.

Table 20.6 Comparative results of DISTINCT and equivalence operators

OPERAND	2 Values, equal	2 Values, not equal	2 NULLS	1 NULL, 1 Value
DISTINCT	False	True	False	True
NOT DISTINCT	True	False	True	False

OPERAND	2 Values, equal	2 Values, not equal	2 NULLS	1 NULL, 1 Value
=	True	False	NULL	NULL
<>, !=	False	True	NULL	NULL

For more information about NULL, see the later topic in this chapter, *Considering NULL*.

LIKE

is a case-sensitive pattern search operator. It is quite a blunt instrument compared to pattern searches in regular expression engines. It recognizes two “wildcard” symbols—% and _ (the underscore character) that work as follows:

% can be substituted into a search string to represent any number of unknown characters, including none. For example,

```
LIKE 'mit%'
```

will be true with strings such as 'blacksmith', 'mitgenommen', 'commit', as well as 'mit'.

An underscore character (_) can be substituted into a search string to represent a single, unknown character. For example, to search for records where the surname might be 'Smith', 'Smyth' or a similar pattern:

```
LIKE 'Sm_th'
```

Escape characters

If you need to search for a string that contains one or both of the wildcard characters in the literal part of the pattern, you can “escape” the literal character: that is, mark it as a literal character by prepending a special escape character. To make this work, you need to tell the engine to recognize your escape character.

For example, say you want to use the LIKE operator to select all of the names of the system tables. Assume that all system table identifiers have at least one underscore character—well, it is almost true! You decide to use “#” as your escape character. Here is how you would do it:

```
SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME LIKE '%#_%' ESCAPE '#';
```



LIKE searches, per se, do not use an index. However, a predicate using LIKE 'Your string%' will be resolved to a STARTING WITH predicate, which does use an index, if one is available.

SIMILAR TO

Supported in versions 2.5+, SIMILAR TO makes an SQL-style comparison between a regular expression pattern and a complete string. The result may be True (there is a complete match), False (there is no match or an incomplete match) or NULL (either or both operands NULL).

Syntax <string-expression> [NOT] SIMILAR TO <pattern> [ESCAPE <escape-char>]

<string-expression> The string field or string expression that the regular expression will be applied to for comparison.

<pattern> must be a regular expression conforming to the SQL convention. For details and examples, refer to *Building regular expressions* which is copied in Appendix I from the Firebird Project documentation, for your convenience.

<escape-char> a single character, user’s choice, specified when required to make literal use a character that is “special” in a regular expression.

The formal “top-down” syntax of SQL regular expressions is abstruse. It is fully detailed in the most recent **Firebird Language Reference Update**, which was the Firebird 2.5 version at the time of this writing, attached to the formal description of SIMILAR TO.

STARTING [WITH]

predicates a case-sensitive search for a string that starts with the supplied string. It follows the byte-matching rules of the prevailing character set and collation. If necessary, it can take a COLLATION argument to force the search to use a specific collation sequence.

Examples SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEE
 WHERE LAST_NAME STARTING WITH 'Jo';

would return rows where the last name was “Johnson”, “Jones”, “Joabim”, etc.

In a database storing character columns in German, where the default character set was ISO8859_1, you might want to force the collation sequence, as follows:

```
SELECT FAM_NAME, ERSTE_NAME FROM ARBEITNEHMER
WHERE COLLATE DE_DE FAM_NAME STARTING WITH 'Schmi';
```

STARTING WITH will use an index if one is available.

Logical Operators

Firebird provides three logical operators that can operate on or with other predicates in distinct ways:

NOT predicates the negative of the search condition to which it is applied. It has highest precedence.

AND forms a complex predicate by combining two or more predicates, all of which must be true for the entire predicate to be true. It is next in precedence.

OR forms a complex predicate by combining two or more predicates, of which one must be true for the entire predicate to be true. OR is last in precedence.

By default, Firebird “shortcut” Boolean logic to determine the outcome of an OR’ed predicate: as soon as one embedded predicate returns as true, evaluation stops and the entire predicate returns as true.

Example of a logical predicate

```
...
COLUMN_1 = COLUMN_2
OR ((COLUMN_1 > COLUMN_2 AND COLUMN_3 < COLUMN_1)
AND NOT (COLUMN_4 = CURRENT_DATE))
```

The effect of this predicate will be as follows:

If COLUMN_1 and COLUMN_2 are equal, evaluation stops and the predicate returns true.

Otherwise, the next predicate—with two embedded predicates—is tested to see whether it could be true. The first embedded predicate itself has two predicates embedded, both of which must be true to make it true as a whole. If COLUMN_1 is less than COLUMN_2, evaluation stops and the predicate as a whole returns false.

If COLUMN_1 is greater, then the COLUMN_3 is evaluated against COLUMN_1. If COLUMN_3 is equal to or greater than COLUMN_1, evaluations stops and the predicate as a whole returns false.

Otherwise, evaluation proceeds to the final embedded predicate. If COLUMN_4 is equal to the current date, the predicate as a whole returns false; if not it returns true.

Inclusive vs. Exclusive OR

Firebird's OR is the inclusive OR (any single condition or all conditions can be true).

The exclusive OR (evaluates to true if any single condition is true but false if all conditions are true) is not supported.

The IS [NOT] NULL Predicate

IS NULL and its alter ego, IS NOT NULL are a pair of predictors that defy grouping. Because NULL is not a value, they are not comparison operators. They test the assertion that the value to the left has a value (IS NOT NULL) or has no value (IS NULL). If there is a value, IS NOT NULL returns true, regardless of the content, size or data type of the value.

Newcomers to SQL sometimes confuse the NOT NULL constraint with the IS [NOT] NULL predicates. It is a common mistake to presume that IS NULL and IS NOT NULL test whether a column has been defined with the NOT NULL constraint. It is not true. They test the current contents of a column to determine the presence or absence of a value.

An IS [NOT] NULL predicate can be used to test input data destined for posting to database columns that are constrained as NOT NULL. It is very common to use this predicate in Before Insert triggers.

Example

This PSQL fragment tests the operand ID for NULL and, if it is, generates a value for it from a sequence:

```
IF (NEW.ID IS NULL) THEN
  NEW.ID = NEXT VALUE FOR MY_GENERATOR;
```

For more information about NULL, see the later topic in this chapter, [*Considering NULL*](#).

Existential Predicates

The last group of predicates comprises those that use subqueries to supply values for assertions of various kinds in search conditions. The predicating keywords are ALL, ANY, EXISTS, IN, SINGULAR and SOME. They came to be referred to as *existential* because all play roles in search predicates that test in some way for the existence of the left-side value in the output of embedded queries on other tables.

All of these predictors are involved in some way with *subqueries*. The topic of subqueries gets detailed treatment in the next chapter, [*Multi-table Queries*](#).

Table 20.7 Existential predicates

Predicator	Purpose
ALL	Tests whether the comparison is true for ALL values returned by the subquery
[NOT] EXISTS	Exists (or not) in at least one value returned by the subquery
[NOT] IN	Exists (or not) in at least one value returned by the subquery
[NOT] SINGULAR	Tests whether exactly one value is returned by the subquery. If NULL is returned, or more than one value, then SINGULAR is false (and NOT SINGULAR is true).
SOME	Tests whether the comparison is true for at least one value returned by the subquery
ANY	Tests whether the comparison is true for at least one value returned by the subquery SOME and ANY are equivalent.

The EXISTS() Predicate

By far the most useful of all of the existential predicates, EXISTS provides the fastest possible method to test for the existence of a value in another table.

Often, in stored procedures or queries, you want to know whether there are any rows in a table meeting a certain set of criteria. You are not interested in how many such rows exist, only in determining whether there is at least one.



The strategy of performing a count() on the set and evaluating any returned value greater than 0 as “true” is costly in Firebird. Furthermore, a row count performed in the context of one transaction to establish a condition for proceeding with work in another—for example, to calculate the value for a “next” key—is completely unreliable.*

The standard SQL predicate EXISTS(subqueried value) and its negative counterpart NOT EXISTS provide a way to perform the set-existence test very cheaply, from the point of view of resources use. It does not generate an output set but merely courses through the table until it meets a row complying with the conditions predicated in the subquery. At that point, it exits and returns true. If it finds no match in any row, it returns false.

In the first example, the EXISTS() test predicates the conditions for performing an update using a dynamic SQL statement:

```
UPDATE TABLEA
  SET COL6 ='SOLD'
 WHERE COL1 = 99
 AND EXISTS(SELECT COLB FROM TABLEB WHERE COLB = 99);
```

A statement like the example would typically take replaceable parameters on the right side of the predicate expressions in the WHERE clauses.

In reality, many subqueries in EXISTS() predicates are correlated—that is, the search conditions for the subquery are relationally linked to one or more columns in the main query. Taking the example above and replacing the hard-coded constant in the search condition with a column reference, a more likely query would be:

```
UPDATE TABLEA
  SET TABLEA.COL6 ='SOLD'
 WHERE EXISTS(
   SELECT TABLEB.COLB FROM TABLEB
   WHERE TABLEB.COLB = TABLEA.COL1);
```

The effect of the existential expression is to set the condition “if there is at least one matching row in TableB, then perform the update.”

The IN () Predicate

The IN() predicate, when used with a subquery, is similar to EXISTS() insofar as it can predicate on the output of a subquery. For example:

```
UPDATE TABLEA
  SET COL6 = 'SOLD'
  WHERE COL1 IN (
    SELECT COLB FROM TABLEB WHERE COLJ > 0);
```

In this case, the subquery returns a set of all values for COLB in the second table that comply with its own WHERE clause. The IN() predicate causes COL1 to be compared with each returned value in the set, until a match is found. It will perform the update on every row of TABLEA whose COL1 value matches any value in the set.



Except in Firebird 1.0, an IN() predicate is actually resolved to an EXISTS() predicate for the comparison operation. In v.1.0.x, indexing can not be used for this style of search.

The ALL () Predicate

Usage is best illustrated by starting with an example:

```
SELECT * FROM MEMBERSHIP
  WHERE
    (EXTRACT (YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM DATE_OF_BIRTH))
    < ALL (SELECT MINIMUM_AGE FROM AGE_GROUP);
```

The left-side expression calculates the age in years of each member in the Membership table and outputs only those members who are younger than the minimum age in the Age_Group table. ALL() is of limited use, since it is really only appropriate for predicating this sort of greater-than or less-than search to test for exclusion.

The SINGULAR () Predicate

SINGULAR is similar to ALL(), except that it predicates that one and only one matching value be found in the embedded set. For example, this query is looking for all orders that have only a single order detail line:

```
SELECT OH.ORDER_ID FROM ORDER_HEADER OH
  WHERE
    OH.ORDER_ID = SINGULAR (SELECT OD.ORDER_ID FROM ORDER_DETAIL OD);
```

ANY () and SOME () Predicates

These two predicates are identical in behavior. Apparently, both are present in the SQL standard for interchangeable use to improve the readability of statements! For equality comparisons, they are logically equivalent to the EXISTS() predicate. However, because they can predicate on other comparisons, such as >, <, >=, <=, STARTING WITH, LIKE and CONTAINING they are especially useful for existential tests that EXISTS() can not perform.

The following statement will retrieve a list of all employees who had at least one salary review within a year of being hired:

```
SELECT E.EMP_NO, E.FULL_NAME, E.HIRE_DATE
  FROM EMPLOYEE E
```

```
WHERE E.HIRE_DATE + 365 > SOME (
  SELECT SH.CHANGE_DATE FROM SALARY_HISTORY SH
  WHERE SH.EMP_NO = E.EMP_NO);
```

Table aliases

Notice the use of table aliases in the last two examples to eliminate ambiguity between matching column names in two tables. Firebird is very particular about aliasing tables in multi-table queries. Table aliases are discussed in the next chapter, *Querying Multiple Tables*.



Prior to the “2” series, Firebird and, before that, InterBase, produced incorrect results for the ALL and NOT IN predicates that were hard to detect. Corrections performed at v.2.0 mean that

- logical assumptions made by legacy application code or PSQL modules that used them in prior versions may now produce different results to those returned before
 - indexes on the inner tables cannot be used and performance may be significantly slower than it was for the same query in InterBase and previous versions of Firebird
- NOT EXISTS is approximately equivalent to NOT IN and will allow Firebird to use indexes.*

Considering NULL

NULL can be quite a sticking point for folk who have previously worked with desktop databases that conveniently swallow unknowns by storing them as “zero values”: empty strings, 0 for numerics, false for logicals, and so on. In SQL, any data item in a nullable column—that is, a column that does not have the NOT NULL constraint—will be stored with a NULL token if no value is ever provided for it in a DML statement or through a column default.

All column definitions in Firebird default to being nullable. Unless your database has been consciously designed to prevent NULLs from being stored, your expressions need to be prepared to encounter them.

NULL in Expressions

NULL is not a value, so it cannot be “equal to” any value. For example, a predicate such as

```
WHERE (COL1 = NULL)
```

will return an error because the equivalence operator is not valid for NULLs. NULL is a state and the correct predicator for a NULL test is IS NULL. The corrected predicate for the failed test above would be

```
WHERE (COL1 IS NULL)
```

You can also test for NOT NULL:

```
WHERE (COL1 IS NOT NULL)
```

Two NULLs are not equal to each other, either. When constructing expressions, be mindful of the cases when a predicate might resolve to

```
WHERE <NULL result> = <NULL result>
```

because false is always the result when two NULLs are compared.

One predictor that you can use when NULL is a possible operand is `IS DISTINCT FROM`. If one operands is NULL, this predictor returns a True; if both are NULL, it returns a False. It also returns a result (not an exception) if both operands are values. For more information about the behaviour of this predictor, refer to the topic above, [*IS \[NOT\] DISTINCT FROM*](#).

An expression like

```
WHERE COL1 > NULL
```

will fail because an arithmetic operator is not valid for NULL.

NULL in Calculations

In an expression where a column identifier “stands in” for the current value of a data item, a NULL operand in a calculation will produce NULL as the result of the calculation, e.g.

```
UPDATE TABLEA
```

```
SET COL4 = COL4 + COL5;
```

will set COL4 to NULL if COL5 is NULL.

In aggregate expressions using operators like `SUM()` and `AVG()` and `COUNT(<specific_column_name>)`, rows containing NULL in the targeted column are ignored for the aggregation. `AVG()` forms the numerator by aggregating the non-null values and the denominator by counting the rows containing non-null values.

Gotchas with True and False

Semantically, if a predicate returns “unknown”, it is neither false nor true. However, in SQL, assertions resolve as either “true” or “false”—an assertion that does not evaluate as “true” shakes out as “false”.

The “IF” condition implicit in search predicates can trip you up when the NOT predictor is used on an embedded assertion, i.e.

```
NOT <condition evaluating to false> evaluates to TRUE
```

whereas

```
NOT <condition evaluating to null> evaluates to NULL
```

To take an example of where our assumptions can bite, consider

```
WHERE NOT (COLUMNA = COLUMNB)
```

If both COLUMNA and COLUMNB have values and they are not equal, the inner predicate evaluates to false. The assertion `NOT(FALSE)` returns true—the flip-side of false.

However, if either of the columns is NULL, the inner predicate evaluates to NULL, standing for the semantic meaning “unknown” (“not proven”, “unknowable”). The assertion that is finally tested is `NOT(NULL)` and the result returns NULL. Take note also that `NOT(NULL)` is not the same as `IS NOT NULL`—a pure binary predicate that never returns “unknown”.



The lesson in this is to be careful with SQL logic and always to test your expressions hard. Cover the null conditions and, if you can do so, avoid NOT assertions altogether in nested predicates.

NULL and External Functions

NULL cannot be returned as output from external functions (“UDFs”). Many third-party external function libraries cannot take NULL as input parameters, either, because they follow the original InterBase convention of passing arguments by reference or by value.

From the “2” series onward, it is possible to pass NULLs to the functions in Firebird’s external function library, *ib_udf*, and to any other external functions that you know have been written (or updated) to handle the NULL in the input parameter. To do so, you must “signal” to the Firebird engine by appending “null” to the declaration of the parameter in the DECLARE FUNCTION statement for that function. For example:

```
DECLARE EXTERNAL FUNCTION SAMPLE
  INT NULL
  RETURNS INT BY VALUE...;
```

v.1.X This form of NULL passing is not supported in Firebird 1.5. A NULL cannot be passed in v1.0.x at all.

Passing parameters by descriptor

The Firebird engine is capable of passing arguments to UDFs by descriptor—a mechanism that standardizes arguments on Firebird data types—making it possible to pass NULL as an argument to host code, although not to receive NULL as a return value. The functions in the library *fbudf* (in the /UDF directory of your server installation, from v.1.5 onward) use descriptors.

Legacy code

If you have legacy code in applications or stored procedures that works around the old limitations by presenting NULLs to external functions as empty strings or zero, that code will have to be updated if you want NULL detection for the v.2+ *ib_udf* functions.

Installation kits for Firebird 2.0 and higher come with a script for upgrading the *ib_udf* function declarations in legacy databases that you are migrating to v.2 or higher, if you want to use the NULL signalling feature with them. It can be found in the directory \$firebird/misc/upgrade/ib_udf (where \$firebird is your Firebird root directory). The file name is *ib_udf_update.sql*.

The string functions in *ib_udf* that can handle the NULL signal are ASCII_CHAR, LOWER, "LOWER", LPAD, LTRIM, RPAD, RTRIM, SUBSTR and SUBSTRLEN. Their code can detect NULL only if it is signalled by the engine—hence the need to update the declarations in the databases. The functions won’t crash if you don’t upgrade: they will simply be unable to detect NULL.



If you are making new declarations for functions, you should use the declarations from \$firebird/UDF/ib_udf2.sql as these contain the null signal for all functions that support it.

Setting a Value to NULL

A data item can be made NULL only in a column that is not subject to the NOT NULL constraint—refer to the topic *The NOT NULL Constraint* in Chapter 15, **Tables**.

In an UPDATE statement the assignment symbol is “=”:

```
UPDATE F00
```

```
SET COL3 = NULL
WHERE COL2 = 4;
```



The use of the “=” symbol in this context is as an assignment operator. It is different to the use of the same symbol as an equivalence operator.

In an INSERT statement, pass the keyword NULL in place of a value:

```
INSERT INTO FOO (COL1, COL2, COL3)
VALUES (1, 2, NULL);
```

In this case, NULL overrides any default set for the column. It will cause an exception if COL3 is not nullable.

Another way to cause NULL to be stored by an INSERT statement is to omit the nullable column from the input list. For example, the following statement has the same effect as the previous one, as long as no default is defined for the column COL3:

```
INSERT INTO FOO (COL1, COL2)
VALUES (1, 2);
```

In PSQL (stored procedure language), use the “=” symbol as the assignment operator when assigning NULL to a variable and use IS [NOT] NULL or IS [NOT] DISTINCT FROM in the predicate of an IF test:

```
...
DECLARE VARIABLE foobar integer;
...
BEGIN
  IF (COL1 IS NOT NULL) THEN
    FOOBAR = NULL;
...

```

Alternatively:

```
...
BEGIN
  IF (COL1 IS DISTINCT FROM NULL) THEN
    FOOBAR = NULL;
...

```

Using Expressions

Next, we take a closer look at some of the ways expressions can be used.

Computed Columns

A useful feature of SQL is its ability to generate run-time output fields using expressions. Firebird supports two kinds of computed output: fields that are created in DML statements and columns that are pre-defined by DDL in tables using the COMPUTED BY keywords and a contextual expression. Usually, such fields are derived from stored data, although they need not be. They can be constants or, more commonly, context variables or values derived from context variables.

Fields Created in Statements

When an output column is created using an expression, it is known as a computed output column. The output value is always read-only, because it is not a stored value. Any expression that performs a comparison, computation or evaluation and returns a single value as a result can be involved in specifying such a column.

It often happens that an expression uses more than one “formula” to get the result. In our earlier example, both a function (EXTRACT()) and an operation (concatenation) were used to derive the BIRTHDAY string that was output in the list. Such complex expressions are not unusual at all.

Derived fields and column aliasing

Expressions, subquery output and constants can be used in Firebird to return derived or “made-up” fields in output sets. To enable you to provide run-time names for such fields, Firebird supports the SQL column-aliasing standard which allows any column to be output using an alias of one or more characters.

For example,

```
SELECT COL_ID, COLA||'|'||COLB AS comput_col
FROM TABLEA;
```

returns a column named comput_col, concatenating two column values separated by a comma.



When two columns are concatenated in this fashion in any expression, the output field will be NULL if either of the columns is NULL. This is standard SQL behaviour.

In the next example, a scalar subquery on another table is used to create a run-time output field:

```
SELECT
  COL_ID,
  COLA,
  COLB,
  (SELECT SOMECOL FROM TABLEB
   WHERE UNIQUE_ID = '99') AS B_SOMECOL
FROM TABLEA
```

A scalar [sub]query is one which returns the value of a single column from a single row. Subqueries are discussed Chapter 21, **Querying Multiple Tables**.

Firebird permits the standard to be relaxed slightly in respect of the AS keyword—it is optional. Omitting it is not recommended, since it can make aliased column names harder to find in source code.

Omitting names for derived fields

Another part of the standard requires columns which are computed or subqueried in the statement to be explicitly named with an alias. Current versions of the Firebird engine let you omit the alias name of computed or subqueried columns altogether. For example, the following query:

```
SELECT
  CAST(CURRENT_DATE as VARCHAR(10))||'-'||REGISTRATION
FROM AIRCRAFT;
```

generates the following output:

<blank title line>

```
=====
2003-01-01-GORILLA
2004-02-28-KANGAROO
...
```

Some regard this as a bug, others as a feature. It is convenient for a DBA to make a quick query without bothering with the detail. It is definitely not recommended as a “feature” to exploit in applications. Aside from the obscurity it creates, it is a rotten future-proofing strategy to employ in a database system whose development continually strives to keep up with standards compliance. Output columns with no name too often cause bugs and ambiguities in client interfaces. The same difficulties can arise with application interfaces if you use the no-alias loophole more than once in a single output.

Constants and variables as run-time output

It is possible to “compute” an output column in a SELECT statement using an expression that involves no columns but is just a constant or a context variable and an alias. In this trivial example, the query adds a column to every row, carrying the constant value “This is just a demo”:

```
SELECT
    LAST_NAME,
    FIRST_NAME,
    'This is just a demo' AS DEMO_STRING
FROM MEMBERSHIP;
```

Trivial as it appears in this example, it can be a handy way to customize output, especially if used with a CASE expression (q.v.).

Using context variables

Firebird supplies a range of variables supplying server snapshot values. As we have already seen, many of these values can be used in expressions that perform calculations. They can also be used in expressions that store them into database columns, either as “hard” values or in *COMPUTED BY Column Definitions* (see below).

Context variables can also be used in an output column expression to return a server value directly to the client or to a PSQL module. To illustrate:

```
SELECT
    LOG_ID,
    LOG_DATE,
    ...,
    CURRENT_DATE AS BASE_DATE,
    CURRENT_TRANSACTION AS BASE_TRANSACTION,
    ...
FROM LOG
WHERE ...
```

Refer to the topic *Context Variables* in Chapter 6 for a detailed list of available variables and more examples of use.

Expressions Using CASE() and Friends

Firebird provides the SQL-99 standard CASE conditional expression syntax and some “shorthand” derivative functions, COALESCE(), IIF() and NULLIF(). A CASE

expression can be used to output a constant value that is determined conditionally at run-time according to the value of a specified column in the current row of the queried table

CASE()

Allows the output for a column to be determined by the outcome of evaluating a group of mutually exclusive conditions.

V.1.0.x *CASE() and the other conditional operators are not supported in v.1.0.x.*

Syntax `SELECT`
 `...`
 `CASE {value 1 | <empty-clause>`
 `WHEN {{NULL |<value 2> } | <search-predicate> } THEN {result 1 | NULL }`
 `WHEN...THEN {result 2 | NULL}`
 `[WHEN...THEN {result n | NULL}]`
 `[ELSE {result (n + 1) | NULL}]`
 `END [,]`

Additional keywords

WHEN ... THEN are keywords in each condition/result clause. At least one condition/result clause is required.

ELSE precedes an optional “last resort” result, to be returned if none of the conditions in the preceding clauses is met.

Arguments

- value 1 is the identifier of the column value that is to be evaluated. It can be omitted, in which case, each WHEN clause must be a search predicate containing this column identifier.
- value 2 is the match part for the search condition: a simple constant, or an expression that evaluates to data type that is compatible with the column's data type
- if the column identifier (value 1) is named in the CASE clause, value 2 stands alone in each WHEN clause and will be compared with value 1
 - if the column identifier is not named in the CASE clause, both value 1 and value 2 are included in each WHEN clause as a search predicate of the form (value1 = value 2)
- result 1* is the result that will be returned in the event that value 1 matches value 2.
- result 2* is the result that will be returned in the event that value 1 matches value 3, and so on.

Return value

The CASE function returns a single result. If no condition is met and no ELSE clause is specified, the result returned will be NULL.

Examples The following two examples demonstrate the two possible syntaxes for CASE expressions, illustrating how each could be used to operate on the same set of predicates. You should use one form of the syntax or the other. Mixed syntax is not valid.

Simple syntax

`SELECT`

```

o.ID,
o.Description,
CASE o.Status
  WHEN 1 THEN 'confirmed'
  WHEN 2 THEN 'in production'
  WHEN 3 THEN 'ready'
  WHEN 4 THEN 'shipped'
  ELSE 'unknown status ' || o.Status || ''
END
FROM Orders o;

```

Syntax using search predicates in the WHEN clause:

```

SELECT
o.ID,
o.Description,
CASE
  WHEN (o.Status IS NULL) THEN 'new'
  WHEN (o.Status = 1) THEN 'confirmed'
  WHEN (o.Status = 3) THEN 'in production'
  WHEN (o.Status = 4) THEN 'ready'
  WHEN (o.Status = 5) THEN 'shipped'
  ELSE 'unknown status ' || o.Status || ''
END
FROM Orders o;

```



Using a single WHEN...THEN clause makes sense only if there is an ELSE clause. However, it is less elegant than the related functions COALESCE() and NULLIF().

COALESCE()

Allows a column value to be calculated from a series of expressions, from which the first expression to return a non-NULL value is returned as the output value.

COALESCE() can be used to evaluate a pair of conditions or a list of three or more conditions.

Syntax COALESCE (value 1> { , value 2 [, ...value n })

Arguments

value 1 The primary column value or expression that is evaluated. If it is not NULL, it is the value returned.

value 2 The column value or expression that is evaluated if value 1 resolves to NULL.

value n The final value of expression that is evaluated if the preceding value in the list resolves to NULL.

Return value

Returns the first non-null result from the list

In the first (simple, binary) syntax, COALESCE(value1, value2) the evaluation logic is equivalent to

```

CASE
  WHEN V1 IS NOT NULL THEN V1
  ELSE V2
END

```

When there are three or more arguments—`COALESCE(value1, value2, ...valuen)`—the evaluation logic is equivalent to

```

CASE
  WHEN V1 IS NOT NULL THEN V1
  ELSE COALESCE (V2,...,Vn)
END

```

The last value in the list should be specified so as to ensure that something is returned.

Example In the first query, if the join fails to find a match in the `EMPLOYEE` table for the `TEAM_LEADER` in the `PROJECT` table, the query will return the string '[Not assigned]' in place of the `NULL` that the outer join would otherwise return as the value for `FULL_NAME`:

```

SELECT
  p.PROJ_NAME AS Projectname,
  COALESCE(e.FULL_NAME, '[Not assigned]') AS Employeeename
FROM PROJECT p
  LEFT JOIN EMPLOYEE e ON (e.EMP_NO = p.TEAM_LEADER);

```

In the next query, evaluation starts at the leftmost position in the list. If no `PHONE` value is present, the query looks to see whether a `MOBILEPHONE` value is present. If so, it returns this in `PHONENUMBER`; otherwise, as a last resort, it returns the string 'Unknown':

```

SELECT
  ID,
  COALESCE(Phone, MobilePhone, 'Unknown') AS Phonenumner
FROM Relations;

```

NULLIF()

Substitutes `NULL` for a value if the value resolves to a non-null value, otherwise returns the value of the sub-expression.

Syntax `NULLIF (value 1, value 2)`

`NULLIF()` is a shorthand for the `CASE` expression

```

CASE
  WHEN (value_1 = value_2) THEN NULL
  ELSE value_1 END

```

Arguments

value 1 The column or expression that is to be evaluated.

value 2 A constant or expression with which value 1 is to be compared. If there is a match, the `NULLIF` expression will return `NULL`.

Return value

Will be NULL if value 1 and value 2 resolve to a match; if there is no match, then value 1 is returned.

Example This statement will cause the STOCK value in the PRODUCTS table to be set to NULL on all rows where it is currently stored as zero:

```
UPDATE PRODUCTS
SET STOCK = NULLIF(STOCK, 0)
```

IIF()

Takes three arguments. If the first evaluates to true, the second argument is returned; otherwise the third is returned.

Syntax: IIF (<condition>, True_result, False_result)

Arguments

<condition> is a Boolean expression, i.e., one that returns a result of either True or False.

True_result is the result to be returned if the condition expression evaluates to True.

False_result is the result to be returned if the condition expression evaluates to False.

Example

```
SELECT
    FIRST_NAME || ' ' || LAST_NAME || ' ' IIF(SEX = 'M' '(Boy)', '(Girl)')
FROM STUDENTS;
```

V.1.X IIF() is not supported in Firebird 1.5.x or 1.0.x.

COMPUTED BY Column Definitions

In table specifications, you can create columns—known as *computed columns*—that store no “hard” values but, instead, store an expression that computes a value whenever the column is referred to by a query. The expression defining the column usually involves the values of one or more other columns in the current row or a context variable. The following is a simple illustration:

```
ALTER TABLE MEMBERSHIP
ADD FULL_NAME COMPUTED BY FIRST_NAME || ' ' || LAST_NAME;
```

It is also possible to use a subquery expression to obtain the run-time value of such a column, a feature that needs to be designed with care to avoid undesirable dependencies. For more information about computed columns, see the topic [*COMPUTED columns*](#) in Chapter 15, *Tables*.

Search Conditions

The ability to construct “formulae” to specify search conditions for selecting sets, locating rows for updates and deletes and for applying rules to input is a fundamental characteristic of a query language. Expressions pervade SQL because they provide the algebra for inserting context into abstract stored data and delivering them as information. Expressions also play an important role in stripping context from input data.

WHERE clauses

A WHERE clause in a statement sets the conditions for choosing rows for an output set or for targeting rows for searched operations (UPDATE, DELETE). Almost any column in a

table can be targeted for a search expression in a WHERE clause. If a data item can be operated on by SQL, it can be tested. Recall that a search predicate is an assertion. Simple or complex assertions are the formulae we construct to specify the conditions that must be true for every row in the set that the main clause of our query is to operate on.

Reversal of operands

The simplified syntax of predicates for the WHERE clause specifies only that it take the form

```
WHERE value operator value
```

In other words, according to syntax, both of these assertions are valid:

```
WHERE ACOL = value
```

and

```
WHERE value = ACOL
```

For predicate types involving the equivalence symbol operators (= and <>) and for a few others, such as LIKE, the SQL parser understands both ways and treats them as equivalent. Other operators will throw exceptions or undetected errors if the left and right sides of a predicate are reversed.

In the case of the “reversible” types, the placement of the operands in predicates is a question of style. However, in the author's experience, the readability of complex SQL predicates in client code and PSQL sources is inversely proportional to the number of predicates presented with the test value placed to the left of the operator, as in the second example.

For example, a nest of predicates of the form

```
<expression-or-constant> = COLUMNX
```

makes hard labor out of troubleshooting and code review, compared with

```
COLUMNX = <expression-or-constant>
```

It is worth any perceived extra effort to make the orientation and structures of all expressions consistent.

Arrays, BLOBs and Strings

An ARRAY type cannot be used in a search predicate at all, because SQL has no way to access the data stored in arrays.

By nature, the contents of BLOBs are without data type or structure from the engine's point of view. Hence, expressions targeting BLOB columns are quite limited. A BLOB can not be compared for equivalence with another BLOB, nor with any other data type.

For text BLOBs, the situation is better, especially from Firebird 2.1 onward. The contents of a text BLOB can be targeted using most of the string operators and functions, such as the predictors STARTING WITH, LIKE and CONTAINING, along with many of the internal string functions that have been implemented in versions 2.0 and 2.1.

A few external functions are available for BLOB types.

Strings can be tested with any comparison operator, although some, such as >, <, >= and <=, tend to be confusing and not very meaningful when working with international character code strings, given that the “arithmetic value” of a string derives not from what is conveyed by the symbolic image but from the cardinality of code points.



If you have numeral strings not mixed with non-numeral characters (other than the point character) you can use a `CAST` expression to have them treated as numbers.

Ordering and Grouping Conditions

When an output field is created from an expression, it can be used for setting the conditions for ordering or grouping the set. However, the syntax rules for expressions in `ORDER BY` and `GROUP BY` are different.

ORDER BY with expressions

A field created using an expression at run-time can not be referred to as a condition in an `ORDER BY` clause by its alias. It can be used by referring to its degree in the set—that is, its position across the row, counting 1 as the position of the leftmost output field. For example:

```
SELECT
  MEMBER_ID,
  LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME,
  JOIN_DATE
FROM MEMBERSHIP
ORDER BY 2;
```

produces a list that is ordered by the concatenated field. In fact, the `ORDER BY` `<degree-number>` is just a shorthand way to write the full syntax for ordering by an expression field that has been defined previously as part of the output set. The full syntax for our example would be:

```
SELECT
  MEMBER_ID,
  LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME,
  JOIN_DATE
FROM MEMBERSHIP
ORDER BY LAST_NAME || ', ' || FIRST_NAME;
```

An `ORDER BY` `<expression>` condition need not reflect any derived field in the output set. In sets other than `UNIONS`, the ordering condition can be any expression that returns a value by which the set can be meaningfully ordered. For example, the `ORDER BY` clause in the following statement calls a function that returns the length of a description column and uses it to order the output from longest to shortest:

```
SELECT
  DOCUMENT_ID,
  TITLE,
  DESCRIPTION
FROM DOCUMENT
ORDER BY STRLEN(DESCRIPTION) DESC;
```

v.1.0.x Ordering by expression is not supported in v.1.0.x.

GROUP BY with expressions

In SQL, a `GROUP BY` statement is used to collect sets of data and organize or summarize them hierarchically.

In all versions of Firebird, you can form a hierarchical grouping based on a call to an external function (UDF) and, in all versions but v.1.0.x, you can use the degree number of an output column as a grouping condition. You can also group by certain other types of expression, including subquery expressions, in all versions except v.1.0.x.

v.1.0.x Certain illegal grouping syntaxes are permitted in Firebird 1.0.x, due to inherited implementation bugs. If you have legacy application or procedure code that was written for v.1.0.x or, indeed, for InterBase, that exploited these “undocumented features”, you will find you have work to do before the code will work with any other versions of Firebird.

The rules and interactions of grouping and ordering by expression are sometimes complicated to implement. The topics are covered in detail in chapter 22, **Ordered and Aggregated Sets**.

Example The following example queries a Membership table and outputs a statistic showing how many members joined in each month, by month:

```
SELECT
  MEMBER_TYPE,
  EXTRACT(MONTH FROM JOIN_DATE) AS MONTH_NUMBER, /* 1, 2, etc. */
  F_MONTHLONG(JOIN_DATE) AS SMONTH, /* UDF that returns month as a string */
  COUNT (*) AS MEMBERS_JOINED
FROM MEMBERSHIP
GROUP BY MEMBER_TYPE, EXTRACT(MONTH FROM JOIN_DATE);
-- or GROUP BY 1, 2;
```

Expression Indexes

From the “2” series forward, an index can be created using arbitrary expressions applied to columns in the table structure. The keywords **COMPUTED BY** preface the expression that is to define the index nodes. The index created is known as an *expression index*.

Using the **ORDER BY** example above, suppose you had created this expression index:

```
CREATE DESC INDEX IXE_DOCLENGTH ON DOCUMENT
COMPUTED BY (STRLEN(DESCRIPTION))
```

The optimizer could use that index for ordering the set because the index matches the **ORDER BY** expression in the query, viz.,

```
SELECT DOCUMENT_ID, TITLE, DESCRIPTION FROM DOCUMENT
ORDER BY STRLEN(DESCRIPTION) DESC;
```

For more information, see the topic *COMPUTED BY <expr>* in the discussion of index syntax in Chapter 16, **Indexes**.

CHECK expressions in DDL

The use of expressions is not restricted to DML, as we have already seen from their use in table definitions for computed columns and in expression indexes. Any time you define a **CHECK** constraint for a table, column or domain, you will use expressions. By nature, a **CHECK** constraint performs a check on one or more values; that is, it tests a predicate. Here is an example, where the member's cellphone number is checked to make sure that, if it is present, it begins with a zero:

```
ALTER TABLE MEMBERSHIP
  ADD CONSTRAINT CHECK_CELLPHONE_NO
    CHECK (CELLPHONE_NO IS NULL OR CELLPHONE_NO STARTING WITH '0');
```

Expressions in PSQL

PSQL, the procedure language for triggers and stored procedures, makes extensive use of expressions for flow control. PSQL provides IF (<predicate>) THEN and WHILE (<predicate>) DO structures. Any predicate that can be tested as a search condition can also be a predicate for program flow condition.

An important function of triggers is to test incoming new values, using expressions, and to use other expressions to transform or create values in the current row or in rows in related tables. For example, this common trigger checks whether a value is null and, if so, makes a function to create a value for it:

```
CREATE TRIGGER BI_MEMBERSHIP FOR MEMBERSHIP
  ACTIVE BEFORE INSERT POSITION 0
  AS
  BEGIN
    IF (NEW.MEMBER_ID IS NULL) THEN
      NEW.MEMBER_ID = GEN_ID(GEN_MEMBER_ID, 1);
    END
```

For detailed information about writing triggers and procedures, refer to Part Six, *Programming on the Server*.

Function Calls

Firebird “out-of-the-box” comes with a set of internally-implemented SQL functions. Until the “2” series, the set of internal functions was kept quite minimal to maintain the small footprint that has always been an important feature for many Firebird server deployments.

The server’s functional capabilities can be extended simply, by its ability to access functions in externally-implemented libraries. Traditionally, such functions were called “user-defined functions”, or UDFs. Correctly, they are *external functions*. In reality, most DBAs use well-tested libraries that are in common use and freely distributed.

As time has passed and the resources available on low-end servers have grown, so has the process of implementing of the most wanted external functions internally. Some string functions appeared in v.2.0 and, for the v.2.1 release, a major internalisation project brought into the language virtually all of the functions traditionally distributed in *ib_udf*.

Appendix I contains the full list of both the internal functions and the external ones in the *ib_udf* and *fbudf* libraries as at the release of v.2.5.

Table 20.8 summarises the internal functions that are available in all versions of Firebird.

Many more internal functions were added as the “2” series progressed. You can find the descriptions and examples in Appendix I.



Table 20.8 SQL functions available in all versions

Function	Type	Purpose
CAST()	Conversion	Converts a value from one data type to another
EXTRACT()	Conversion	Extracts date and time parts (year, month, day, etc.) from DATE, TIME, and TIMESTAMP values
SUBSTRING() †	String manipulation	Retrieves any sequence of characters from a string
UPPER()	String manipulation	Converts a string to all uppercase
GEN_ID()	General	Returns a value from a generator
AVG()	Aggregating	Calculates the average of a set of values
COUNT()	Aggregating	Returns the number of rows that satisfy a query’s search condition
MAX()	Aggregating	Retrieves the maximum value from a set of values
MIN()	Aggregating	Retrieves the minimum value from a set of values
SUM()	Aggregating	Totals the values in a set of numeric values

† Not available in v.1.0.x

Conversion Functions

Conversion functions transform data types, for example, by converting them from one type to another, compatible type, changing the scale or precision of numeric values or by distilling some inherent attribute from a data item. Many string functions can be said to be conversion functions, too, because they transform the way stored string values are represented in output.

CAST()

A wide-ranging function that allows a data item of one type to be converted to or treated as another type.

Syntax `CAST(value AS <sql-data-type>)`

The function returns a computed field of the designated data type. AS <data-type> within the argument phrase is mandatory.

Arguments

value A column or expression that evaluates to a data type that can validly be converted to the data-type named by the AS keyword.

Until v.2.0, the **<sql-data-type>** must be a native Firebird data type. Casting to a domain is not valid in prior versions.

From the “2” series onward, a CAST expression can use the data type of a domain in lieu of a native data type, using the **TYPE OF <domain>** form that was made available for PSQL. The pattern is:

`CAST (expression AS TYPE OF <domain-name>)`

Also available, from v.2.5 onward, is the use of **TYPE OF COLUMN** with a reference to a column defined in a table or view elsewhere. It can be a column in the same table but it doesn’t have to be. The pattern is:

CAST (expression AS TYPE OF COLUMN relation-name.column-name)

Note that these casting inherit the data type of the AS object but not other attributes. However, for CHAR and VARCHAR types, they do inherit the character set, if it was explicitly specified in the domain or column definition.

Table 6.4, under *Valid Conversions*, in Chapter 6, shows all allowed type-to-type castings.

Examples In the following PSQL snippet, a `TIMESTAMP` field, `LOG_DATE`, is cast to a `DATE` type because a calculation needs to be performed on whole days:

```
...
IF (CURRENT_DATE - CAST(LOG_DATE AS DATE) = 30) THEN
    STATUS = '30 DAYS';
```

The following statement takes the value of an integer column, casts it as a string and concatenates it to a `CHAR(3)` column to form the value for another column:

```
UPDATE MEMBERSHIP
SET MEMBER_CODE = MEMBER_GROUP || CAST(MEMBER_ID AS CHAR(8))
WHERE MEMBER_CODE IS NULL;
```

Refer to the topic *Converting Data Types* in Chapter 6 for a more detailed discussion of `CAST()` and to the succeeding chapters that deal with each data type individually.

EXTRACT()

Extracts a part of a `DATE`, `TIME` or `TIMESTAMP` field as a number. It will not work with values that do not resolve as date/time types.

All parts return `SMALLINT` except `SECOND`, which is `DECIMAL(6,4)`

Syntax `EXTRACT(<part> FROM <field>)`

Arguments

<part> One member of the optional keywords set `YEAR` | `MONTH` | `DAY` | `HOURL` | `MINUTE` | `SECOND` | `MILLISECOND`[†] | `WEEKDAY` | `YEARDAY`.

`WEEKDAY` extracts the day of the week (having Sunday = 0, Monday = 1, and so on) and `YEARDAY` extracts the day of the year (from January 1 = 1 to 366).

[†] The `MILLISECOND` part is not supported in versions 1.X.

<field> is a valid `DATE`, `TIME` or `TIMESTAMP` field (column, variable or expression).

Example This statement returns names and birthdays, in birthday order, of all members who have their birthdays in the current month:

```
SELECT
    FIRST_NAME,
    LAST_NAME,
    EXTRACT(DAY FROM DATE_OF_BIRTH) AS BIRTHDAY
FROM MEMBERSHIP
WHERE DATE_OF_BIRTH IS NOT NULL
AND EXTRACT(MONTH FROM DATE_OF_BIRTH) = EXTRACT(MONTH FROM CURRENT_DATE)
ORDER BY 3;
```

String Functions

A large variety of string functions is available in versions of Firebird 2.1+. Here we discuss the ones that are available to lower versions. All of the string functions, including the external functions that can be used with lower versions, are described in Appendix I.

SUBSTRING()

Internal function implementing the ANSI standard SQL SUBSTRING() function. It will return a stream of bytes consisting of the byte at <startpos> and all subsequent bytes up to the end of the <value> string. If the optional FOR length clause is specified, it will return the lesser of <length> bytes or the number of bytes up to the end of the input stream.

Syntax SUBSTRING (<value> FROM <startpos> [FOR <length>])

Arguments

- <value> can be any expression, constant or column identifier that evaluates to a string.
- <startpos> can be any expression that evaluates to an integer not less than 1. In versions older than Firebird 2.0 it cannot be an expression or a replaceable parameter.
- <length> can be any expression that evaluates to an integer not less than 1. In versions older than Firebird 2.0 it cannot be an expression or a replaceable parameter..



- 1 The values <startpos> and <length> are byte positions—this matters for multi-byte character sets.
- 2 If the input <value> is a BLOB, then the result will be a string in versions 2.0.x and lower and a BLOB in versions 2.1+.
- 3 For a string argument, the function will handle any character set. The calling statement is responsible for handling any issues arising with multi-byte character sets.
- 4 For BLOB column arguments, the column named can be a binary BLOB (SUB_TYPE 0) or a text BLOB (SUB_TYPE 1). It is the responsibility of the application to make sense of the stream of bytes returned.
- 5 Thanks to the SQL standards committee, the length of the result is the same as the length of <value>. This means, for example, that given value x is a VARCHAR(50), the expression SUBSTRING(x FROM 1 FOR 2) will return a VARCHAR(50), not a VARCHAR(2).

Example The following statement will update the value in COLUMNB to be a string of up to 99 characters, starting at the fourth position of the original string:

```
UPDATE ATABLE
SET COLUMNB = SUBSTRING(COLUMNB FROM 4 FOR 99)
WHERE ...
```

UPPER()

Converts a string to all upper case characters. If the character set or collation sequence supports uppercasing, returns a stream of converted, all upper-case characters, of the same byte length as the input value. For unsupported character sets, returns the input value unchanged. Versions 2.1+ can take a BLOB as input and will return a BLOB.

Syntax UPPER(<value>)

Arguments

<value> A column, variable or expression that evaluates to a character type. In versions prior to v.2.1, it cannot be a BLOB type.

Example The following CHECK constraint validates a string input by testing whether it consists of all upper-case characters:

```
ALTER TABLE MEMBERSHIP
  ADD CONSTRAINT CHECK_LOCALITY_CASE
  CHECK(LOCALITY = UPPER(LOCALITY));
```

LOWER()

Supported from the “2” series onward, LOWER() converts a string to all lower case characters. If the character set or collation sequence supports uppercasing, returns a stream of converted, all lower-case characters, of the same byte length as the input value. For unsupported character sets, returns the input value unchanged. Versions 2.1+ can take a BLOB as input and will return a BLOB.

Syntax LOWER(<value>)

Example:

```
SELECT MEMBER_NAME FROM MEMBERSHIP
  WHERE LOWER(LOCALITY) = 'uptown';
```



If the external function LOWER() has been declared in the database, the internal function takes precedence. If you really want to call the external function instead, enclose the fully upper-cased function name in double quotes, viz., "LOWER".

Function for Getting a Generator (Sequence) Value

The GEN_ID() function is the mechanism by which PSQL modules and applications draw numbers from generators in all versions of Firebird.

GEN_ID()

Calculates and returns a value from a generator. It returns a BIGINT in dialect 3, an INTEGER in dialect 1.

GEN_ID() always executes outside of all transactions. It is the only user-accessible operation in Firebird that can do so. Once a number has been acquired from a generator, it can never be generated again from the same generator, unless a user intervenes and breaks the sequence using a negative step or a SET GENERATOR statement.

Syntax GEN_ID(<value1>, <value2>)

Arguments

<value1> is the identifier of an existing generator.

<value2> is the stepping value, an integer type, or an expression that evaluates to an integer type.

Normally, the stepping value is 1. A stepping value of 0 will return the last value generated. Larger steps as well as negative ones are possible. However, you should avoid negative stepping unless you really mean to break the forward sequence.

Example The following statement returns a new value from a generator named GEN_SERIAL:

```
SELECT GEN_ID(GEN_SERIAL, 1) FROM RDB$DATABASE;
```

In the next example, a generator is used in a BEFORE INSERT trigger to populate a primary key:

```
CREATE TRIGGER BI_AUTHORS FOR AUTHORS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    IF (NEW.AUTHOR_ID IS NULL) THEN
        NEW.AUTHOR_ID = GEN_ID(GEN_AUTHOR_ID, 1);
    END ^
```

NEXT VALUE FOR

From the “2” series onward, Firebird also supports CREATE SEQUENCE <sequence-name> and the DML call NEXT VALUE FOR <sequence-name>. It is not a function call but a statement embodied in the DML lexicon. For example, using a fragment from the example above:

```
...
BEGIN
    IF (NEW.AUTHOR_ID IS NULL) THEN
        NEW.AUTHOR_ID = NEXT VALUE FOR GEN_AUTHOR_ID;
    END ^
```

A generator and a sequence are exactly the same thing. You can apply NEXT VALUE FOR to a generator created using CREATE GENERATOR or to a sequence created using CREATE SEQUENCE. The function GEN_ID(), although non-standard, has more “grunt” than NEXT VALUE FOR, since the latter does not support any stepping value other than 1.

Details about creating and working with generators (sequences) are in Chapter 12, **Database Basics**, in the topic *Generators (Sequences)*. The technique for using them to implement and maintain primary keys and other automatically incrementing series is described in Chapter 30, in the topic *Implementing Auto-Incrementing Keys*.

Aggregating Functions

Aggregating functions perform calculations over a column of values, such as the values selected in a numeric column from a queried set. Firebird has a group of aggregating functions that are most typically used in combination with grouping conditions to calculate group-level totals and statistics. The aggregating functions are SUM() which calculates totals, MAX() and MIN() returning the highest and lowest values, respectively, and AVG() which calculates averages. The COUNT() function also behaves as an aggregating function in grouped queries and DISTINCT queries aggregate by excluding duplicates. The LIST() function aggregates a singular value expression into a BLOB.

Chapter 22, **Ordered and Aggregated Sets**, addresses the participation of aggregating functions in grouped queries.

“Non-grouped” aggregations

In a few situations, aggregating functions can operate on sets that are not subject to GROUP BY, returning at most, a single row. Logically, the result of an aggregating query can not output any database column values or values derived from non-aggregating functions. To illustrate, the following query aggregates the budgets for one project for one fiscal year. The table has one budget record for each of five departments:

```

SELECT
  'MKTPR-1994' AS PROJECT,
  SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE PROJ_ID = 'MKTPR' AND FISCAL_YEAR = 1994;

```

The output is one single row consisting of the run-time string field and the calculated total.

Functions for Setting and Getting Contextual Data

From the “2” series onward, the engine makes available some useful contextual information that is additional to context variables discussed earlier in this chapter. Values are stored as variables in three discrete areas of memory, known as *namespaces*. The stored information is retrieved using a call to the function `RDB$GET_CONTEXT()`. What can be retrieved depends on which namespace you want to refer to and what information is stored there.

The Namespaces

The three namespaces are `SYSTEM`, `USER_SESSION` and `USER_TRANSACTION`.

- In the `SYSTEM` namespace the context variables are static and read-only and are maintained by the engine.
- In the `USER_SESSION` and `USER_TRANSACTION` namespaces, which are initially empty, the context variables are read/write. Variables in these namespaces are defined in a PSQL module, by the application making a call to the function `RDB$SET_CONTEXT`. The variables can be whatever are needed by the application and they can be set and reset at any point during the scope of the namespace:
 - variables stored in the `USER_SESSION` namespace persist until the end of the client’s session
 - variables stored in the `USER_TRANSACTION` namespace exist for the duration of the transaction in which they are set. Variables existing in this namespace are not affected by `ROLLBACK RETAIN` or `ROLLBACK TO SAVEPOINT`.

A read/write variable can have a new value assigned to it during its life. Setting it to `NULL` makes it inactive.

The `SYSTEM` namespace

Table 20.9 enumerates the read-only variables available in the `SYSTEM` namespace.

Table 20.9 Context variables in the `SYSTEM` namespace

Variable	Information	Availability
<code>CLIENT_ADDRESS</code>	For a TCP connection (v4), returns the IP address of the client. Returns the local process ID if the XNET protocol is used. Returns <code>NULL</code> for any other protocol.	
<code>CURRENT_ROLE</code>	Returns the role name the connected user used at login, i.e., the same information returned by the general variable <code>CURRENT_ROLE</code> . Returns <code>NONE</code> if an explicit role was not used at login.	

Variable	Information	Availability
CURRENT_USER	Returns the user name of the connected user, i.e., the same information returned by the general variables CURRENT_USER and USER.	
DB_NAME	The full path to the database or the database alias, depending on the configuration of the DatabaseAccess parameter in firebird.conf.	
ENGINE_VERSION	Returns the Firebird engine (server) version.	Not in v.2.0
ISOLATION_LEVEL	The isolation level of the current transaction: READ COMMITTED SNAPSHOT CONSISTENCY'.	
NETWORK_PROTOCOL	The protocol used for the current connection: 'TCPv4', 'WNET', 'XNET' or NULL.	
SESSION_ID	Returns the variable that is otherwise visible as CURRENT_CONNECTION.	
TRANSACTION_ID	Returns the variable that is otherwise visible as CURRENT_TRANSACTION.	

RDB\$GET_CONTEXT()

Applications and PSQL modules retrieve a context variable from a given namespace by polling it using the function RDB\$GET_CONTEXT. It is called like a UDF—which in fact it is—with its arguments passed as case-sensitive strings enclosed in single quotes. The syntax for a retrieval expression is

```
RDB$GET_CONTEXT ('<namespace>', '<variable-name>')
```

Arguments

RDB\$GET_CONTEXT takes two arguments:

<namespace> must be one of SYSTEM | USER_SESSION | USER_TRANSACTION

<variable-name> is a case-sensitive string of not more than 80 characters and it cannot be a null string

If the given namespace contains the polled variable, its value is returned as a VARCHAR(255). If the SYSTEM namespace is polled for a non-existent variable, an exception occurs. Polling a non-existent variable in one of the other namespaces does not incur an exception: NULL is returned.

Examples The first example polls the SYSTEM namespace for the NETWORK_PROTOCOL variable:

```
SELECT RDB$GET_CONTEXT('SYSTEM', 'NETWORK_PROTOCOL')
FROM RDB$DATABASE;
RDB$GET_CONTEXT
=====
TCPv4
```

In this snippet from a BEFORE INSERT OR UPDATE trigger, the USER_SESSION namespace is polled for a variable named “APP_NAME” that has been set at the start of the session and the returned value is assigned to the NEW value of a column in the table:

```
...
NEW.Application_Name = RDB$GET_CONTEXT('USER_SESSION', 'APP_NAME');
```

...

The same data might be used directly in an INSERT statement:

```
INSERT INTO ACTIVITY_LOG (
    ...,
    APPLICATION_NAME,
    ...)
VALUES (
    ...,
    RDB$GET_CONTEXT('USER_SESSION', 'APP_NAME')
    ...);
```

RDB\$SET_CONTEXT()

With the internally declared UDF RDB\$SET_CONTEXT, you can create your own context variable in one of the read/write namespaces USER_SESSION or USER_TRANSACTION. It can be invoked from an application using a SELECT statement, from which it returns an integer result code—1 if the variable existed already or 0 if not. In PSQL it can be used like a void function in C/C++ or a procedure in Pascal.

Up to 1000 variables can exist concurrently in any one namespace.

no matter at which point during the transaction they were set.

The first two arguments are passed as case-sensitive strings enclosed in single quotes. The syntax for a variable-setting expression is

```
RDB$SET_CONTEXT ('<namespace>', '<variable-name>', <value> | NULL)
```

Arguments

RDB\$SET_CONTEXT takes three arguments:

<namespace> must be one of USER_SESSION | USER_TRANSACTION

<variable-name> is a case-sensitive string of not more than 80 characters and it cannot be a null string

<value> can be of any type, provided it can be cast to VARCHAR(255). Literals should be enclosed in single quotes. NULL is valid if the variable already exists: passing NULL deactivates the variable.

Examples

The first example uses a SELECT statement to invoke the function

```
SELECT RDB$SET_CONTEXT ('USER_SESSION', 'APP_NAME', 'General Data Entry')
FROM RDB$DATABASE;
```

The next example sets the context variable directly, calling RDB\$SET_CONTEXT like a void function (procedure):

```
CREATE PROCEDURE SET_SESSION_VARS (...)
    ...
AS
BEGIN
    ...
    RDB$SET_CONTEXT('USER_SESSION', 'APP_NAME_ROLE', 'General Data Entry' || ' ' ||
    CURRENT_ROLE);
    ...
END
```



RDB\$GET_CONTEXT and RDB\$SET_CONTEXT are not internal functions. They are implemented as UDFs but you do not have to declare them to databases yourself—they are pre-declared in databases of ODS 11 or higher at create or restore time. Neither function is available to databases of ODS lower than 11.

External Functions (UDFs)

External functions are code routines written in a host language such as C, C++ or Pascal and compiled as shared binary libraries—DLLs on Windows, shared objects on other platforms that support dynamic loading. Like the standard, built-in SQL functions, external functions can be designed to do conversions or calculations that are either too complex or impossible to do with the SQL language.

You can access external functions by using them in expressions, just as you use a built-in SQL function. Like the internal functions, they can also return values to variables or SQL expressions in stored procedure and trigger bodies.

The “user-defined” part comes in because you can write your own functions. The possibilities for creating custom functions for your Firebird server are limited only by your imagination and skill as a host-language programmer. Possibilities include statistical, string, date and mathematical functions, data-formatting routines and more.

External functions should not be designed to make their own connections to databases. Like the internal functions, they must operate on data, in expressions, in the context of the current server process and transaction and a single statement. Because external functions can take only arguments that are (or can be resolved as) native Firebird data types, they can not take a query set specifier as an argument. Thus, for example, it is not possible to write aggregate functions that do not operate on literal arguments.

Existing libraries

Firebird ships with two pre-built libraries of external functions (UDFs). The default installations place these shared object libraries in the `./UDF` directory beneath the Firebird root directory. File extensions are “.dll” on Windows, “.so” on other supported platforms.

ib_udf—a library of useful, basic functions which, in its original form, Firebird inherited with its InterBase® ancestry. This library needs to call some memory utilities which are located in a companion shared object name *ib_util*, located in the `./bin` directory. This library passes parameters either by value or by reference, in the conventional InterBase® style. Several routines have been bug-fixed in Firebird, so do make certain you avoid versions that ship with Borland products.

fbudf—by Claudio Valderrama, passes parameters by Firebird descriptor, considered a more robust way to ensure that internal errors do not occur as a result of memory allocation and type conversion errors.

Also freely available are several public domain UDF libraries, including FreeAdHocUDF, developed by an international group from an old library named FreeUDFLib, originally written in Borland Delphi™ by Gregory Deatz. FreeAdHocUDF contains a vast number of string, math, BLOB, statistical and date functions. It is a continual work in progress, with library versions available for multiple platforms.

You can find out more about FreeAdHocUDF and download the libraries at http://freeadhocudf.org/index_eng.html.

Configuration and security issues

External code modules are inherently vulnerable to malicious intruders and careless administrators. They are, after all, just files in the filesystem.

Access to external files of any sort can be restricted to various levels of access. By default, if you decide to place external function libraries in non-default locations, they will be inaccessible to the server. Study the notes in Chapter 34 about configuring the [UdfAccess](#) parameter in `firebird.conf` to resolve such problems.

v.1.0.x In Firebird 1.0.x, you can—and should—configure the location(s) of external function libraries for the server explicitly, using the `external_function_directory` parameter in the configuration file (`isc_config` on POSIX servers, `ibconfig` on Windows). It behaves the DBA to ensure that libraries cannot be overwritten by accident or by unauthorised visitors. Instructions can be found in the same section of Chapter 34 as that for `UDFAccess` for more recent versions.

Stability of UDFs

A bad UDF will crash the server and is capable of corrupting data. It is important to test your home-built external function routines with utter thoroughness, both outside and from within the server, before deciding to deploy them into production databases.

Declaring a function to a database

Once a UDF has been written, compiled, tested thoroughly and installed into the appropriate directory on the server, it must be declared to the database, in order to be used as an SQL function. To do this, use the DDL statement, `DECLARE EXTERNAL FUNCTION`. You can declare functions using *isql*, another interactive SQL tool, or a script.

After the function has been declared in any database on the server, the library which contains it will be loaded dynamically at run time the first time an application calls any function included in it.

It is necessary to declare each function you want to use to each database in which it will be used.

Declaring a function to a database informs the database of

- The function name as it will be used in SQL statements. You can use your own custom name in the declaration—see the syntax notes below.
- The number and data types of its arguments
- The data type of the return value
- The name (`ENTRY_POINT`) of the function as it exists in the library
- The name of the library (`MODULE_NAME`) that contains the function

Declaration syntax

```
DECLARE EXTERNAL FUNCTION name {[data-type | CSTRING (int)] [NULL]
[, data-type | CSTRING (int) ...]}
RETURNS {data-type [BY VALUE] | CSTRING (int)} [FREE_IT]
[RETURNS PARAMETER n]
ENTRY_POINT 'entry-name'
MODULE_NAME 'module-name';
```

Table 20.10 Arguments for DECLARE EXTERNAL FUNCTION

Argument	Description
name	Name of the UDF to use in SQL statements; can be different from the name of the function specified after the ENTRY_POINT keyword. If you are using an existing library, it is fine to change the supplied name—just avoid making it too confusing by declaring the same function with different names in different databases!
data-type	Data type of an input or return parameter. Unless arguments are being passed by descriptor, this is a data type of the host language, not SQL. All input parameters are passed to a UDF by reference or by descriptor. Return parameters can be passed by reference, by value or by descriptor. The data type cannot be an array or an array element
NULL	From “2” series onward, used to indicate NULL signalling. This aspect is discussed in detail in NULL and External Functions in the topic above, Considering NULL.
RETURNS	Specification for the return value of a function
BY VALUE	Specifies that a return value should be passed by value rather than by reference
CSTRING (int)	Specifies a UDF that returns a null-terminated string up to a maximum of int bytes in length
FREE_IT	Frees memory of the return value after the UDF finishes running. Use only if the memory is allocated dynamically in the UDF using the ib_util_malloc function defined in the ib_util library. The ib_util library must be present in the /bin directory for FREE_IT to work.
RETURNS PARAMETER n	Specifies that the function returns the nth input parameter; required for returning BLOBs
entry-point-name	Quoted string specifying the name of the UDF in the source code and as stored in the UDF library
module-name	Quoted file specification identifying the library that contains the UDF. The library must reside on the server. See note below regarding path names.

Altering a UDF declaration

ALTER EXTERNAL FUNCTION was added to the lexicon at v.2.0, to enable the name of the entry_point or the name of the module to be changed when the UDF declaration cannot be dropped due to existing dependencies. The syntax is:

```
ALTER EXTERNAL FUNCTION name
  {[[ENTRY_POINT 'entry-name']] [MODULE_NAME 'module-name']]};
```

Attempts to include any other attributes in the statement will cause an exception.

External function library scripts

Most external function libraries are distributed with their own DDL scripts, containing a declaration for each function and, usually, some brief documentation. The convention is to name the script after the library, using the file extension “.SQL” or “.sql”. Not all public domain libraries adhere to that convention, however.

The ib_udf.sql, ib_udf2.sql[†] and fbudf.sql scripts are in the /UDF directory of your server installation. The FreeAdHocUDF kit comes with several scripts. You can freely copy-and-paste declarations to assemble your own library of favorite declarations.

[†]The `ib_udf2.sql` script contains the declarations for the functions that were modified to detect NULL signalling.

Examples This declaration is an example from the `ib_udf.sql` script:

```
DECLARE EXTERNAL FUNCTION lpad
  CSTRING(80), INTEGER, CSTRING(1)
  RETURNS CSTRING(80) FREE_IT
  ENTRY_POINT 'IB_UDF_lpad'
  MODULE_NAME 'ib_udf';
```

The next comes from `fbudf.sql`, which passes arguments by descriptor in some functions:

```
DECLARE EXTERNAL FUNCTION sright
  VARCHAR(100) BY DESCRIPTOR, SMALLINT,
  VARCHAR(100) RETURNS PARAMETER 3
  ENTRY_POINT 'right'
  MODULE_NAME 'fbudf';
```

The function identifier (name)

When declaring an external function to a database, you are not restricted to using the name that appears in the script. A function name must be unique among all function declarations in the database. The libraries in general circulation usually conform to a convention of supplying declarations that do not step on identifiers that are commonly in use in another library—hence the strangeness of some names that you see in the scripts.

Sometimes, you want to declare the same function more than once in your database—for example, see the note below regarding functions with string arguments. As long as you use different names for the `EXTERNAL FUNCTION`, you can declare a function having the same `ENTRY_POINT` and `MODULE_NAME` as many times as you need to.



Never alter the `ENTRY_POINT` argument. The `MODULE_NAME` argument should not be altered except where it is necessary to use the library's full path—see below.

String argument sizes

The scripts contain default size declarations for external functions that take variable string arguments. If string functions receive input or return results that are larger than the declared size, exceptions are thrown. For security reasons, the default sizes are kept small, to avoid the risk of accidental or malicious overflows.

If you need an argument for a string function that is larger than the default, declare the argument according to your requirement, ensuring that the inputs and outputs are consistent with one another and no argument exceeds the maximum `VARCHAR` size of 32,765 bytes—note **bytes**, not characters.

In the following example, the function that is scripted in `fbudf.sql` as `sright` is declared with the name `sright200` and the parameters are resized to permit 200-bytes:

```
DECLARE EXTERNAL FUNCTION sright200
  VARCHAR(200) BY DESCRIPTOR, SMALLINT,
  VARCHAR(200) RETURNS PARAMETER 3
  ENTRY_POINT 'right' MODULE_NAME 'fbudf';
```

Path-names in function declarations

On any platform, the module can be referenced with no path name or file extension and the engine will try to find it in the \$firebird/UDF directory. This is desirable if you want to be able to transport a database containing function declarations to multiple operating systems. The default configuration of all versions later than v.1.0.x enables this.

Other configuration options are available for function library locations, through the configuration parameter **UDFAccess** in `firebird.conf`. With the default **RESTRICT** you can specify other directory locations besides /UDF. Setting **NONE** prevents external function modules from being used at all, while **FULL** offers no restriction. However, if you use **FULL**, it is necessary to include the module's full path specification, including file extension, in the declaration. In this case, the database will not be portable to another OS unless you drop the functions and their dependencies from the database first.

For more about configuring this parameter, see [UdfAccess](#) in Chapter 34, **Configuration Parameters in Detail**.

This example shows a function declaration with a full path specification:

```
DECLARE EXTERNAL FUNCTION lpad
  CSTRING(80), INTEGER, CSTRING(1)
  RETURNS CSTRING(80) FREE_IT
  ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME '/opt/extlibs/ib_udf.so';
```

v.1.0.x On v.1.0.x, there is no default path. It will be necessary to configure the **external_function_directory** parameter in `isc_config/ibconfig` on each platform where the functions will be used.

External function reference

Appendix I contains a detailed reference, with examples, for the external functions in the *ib_udf* and *fbudf* libraries. Refer to the section [External Functions](#).

CHAPTER 21

QUERYING MULTIPLE TABLES

Firebird dynamic SQL supports three ways to query multiple tables with a single SQL statement: by joining, subqueries and unions. The output from joins, unions and subqueried columns is read-only by nature.

Firebird does not support a multi-table query that spans multiple databases. However, it can query tables from multiple databases simultaneously inside a single transaction, with full two-phase commit (“2PC”). Client applications can correlate data and perform atomic DML operations across database boundaries. Multi-database transactions are discussed in Part Five, *Transactions*.

In this chapter, we explore the three methods for dynamically retrieving sets from multiple tables¹ in the context of a single statement and a single database: by joining, by subqueries and by unions.

Kinds of Multi-table Queries

The three methods of retrieving data from multiple tables are quite distinct from one another and, as a rule, perform different kinds of retrieval task. Because joining and subquerying both involve merging streams of data from rows in different tables, their roles overlap to under some conditions. A correlated subquery, which can form a relational linkage with columns in the main table, can sometimes enable a set specification to produce the same output as a join, without use of a join. Union queries, on the other hand, do not merge streams, they “stack” rows. Their role never overlaps with those of joining queries or subqueries.

Joins, subqueries and unions are not mutually exclusive, although union sets can not be joined to one another or to other sets. Joined and union set specifications can include subqueries and some subqueries can contain joins.

1. Program logic, in combination with the DSQL techniques described in this chapter, can be employed in PSQL modules that return virtual tables to clients and to other PSQL modules. Refer to Part Six, Programming on the Server. Views and other run-time set objects, discussed in Chapter 23, provide other options for extracting multiple-table sets.

Joining

Joining is one of the most powerful features of a relational database because of its capacity to retrieve abstract, normalized data from storage and deliver denormalized sets to applications in context. In JOIN operations, two or more related tables are combined by linking related columns in each table. Through these linkages, a virtual table is generated that contains columns from all of the tables.

Join operations produce read-only sets that can not be targeted for INSERT, UPDATE or DELETE operations. Some application interface layers implement ways to make joined sets behave as if they were updatable.

Subqueries

A subquery is a SELECT statement that is embedded within another query. Embedded query, in-line query, nested query and sub-select are all synonyms for subquery. They are used under a variety of conditions for reading data from other tables into the enclosing query. The rules for subqueries vary according to purpose. The data they retrieve are always read-only.

UNION queries

Union queries provide the ability to extract rows with matching formats from different sets into a unified set that applications can use as if it were a single, read-only table. The sub-sets retrieved from the tables do not need to be related to one another: they simply have to match one another structurally.

Using Relation Aliases

When the name of the table or view is long or complicated, or there are many relations, relation aliases are useful to clarify statements. In some cases, they are mandatory to disambiguate and distinguish the separate cursor streams in self-referencing queries and multi-table queries that have some column names in common. The query engine treats a table alias as a synonym for the table it represents.



In a database that was created using delimited identifiers combined with lower-case or mixed case object names and/or “illegal characters”, typing queries could get quite exhausting and error-prone if table-aliasing were not available.

An alias can be used wherever it is valid to use the relation’s name as a qualifier and all relation identifiers must be substituted. Mixing table identifiers with aliases in the same query will cause exceptions.

Examples

For the purpose of these examples we’ll use a JOIN query. However, the rules and rationale for using aliases applies to all simple and complex types of query that involve linkages between multiple tables and sets.

Using table identifiers:

```
SELECT
  TABLEA.COL1,
  TABLEA.COL2,
  TABLEB.COLB,
```

```

TABLEB.COLC,
TABLEC.ACOL
FROM TABLEA
  JOIN TABLEB ON TABLEA.COL1 = TABLEB.COLA
  JOIN TABLEC ON TABLEB.COLX = TABLEC.BCOL
WHERE TABLEA.STATUS = 'SOLD'
      AND TABLEC.SALESMAN_ID = '44'
      ORDER BY TABLEA.COL1;

```

The same example, using aliases:

```

SELECT
  A.COL1,
  A.COL2,
  B.COLB,
  B.COLC,
  C.ACOL
FROM TABLEA A /* identifies the alias */
  JOIN TABLEB B ON A.COL1 = B.COLA
  JOIN TABLEC C ON B.COLX = C.BCOL
WHERE A.STATUS = 'SOLD'
      AND C.SALESMAN_ID = '44'
      ORDER BY A.COL1;

```

Legal relation alias names

Use any useful string composed of up to 31 of the characters that are valid for metadata qualifiers, i.e. alphanumeric characters with ASCII codes in the ranges 35-38, 48-57, 64-90 and 97-122. Spaces, diacritics and alias names that begin with a numeral are not legal. You can use double-quoted alias names but it is not recommended, especially in complex queries where simplicity rules supreme.

The internal cursor

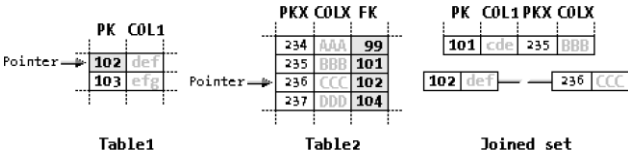
When reading through a stream, the database engine implements a pointer, whose address changes as the read advances from from top to bottom. This pointer is known as a *cursor*—not to be confused with the cursor sets that are implemented in PSQL. Internal cursors are not accessible to clients except through the medium of join and subquery syntax.

The current address of an internal cursor is an absolute offset from the address of the first row in the set, which means it can only advance forwards. Internally, the optimizer utilizes indexes and organizes input streams into a plan, to ensure that a query begins returning output in the shortest possible time.

In any multi-table operation, the Firebird engine maintains one internal cursor for each stream. In the preceding JOIN examples, TableA and TableB each has its own cursor. When a match occurs, the engine creates a merged image for the output stream by copying the streams from the current addresses of the two cursors.

Figure 21.1 illustrates how a match for the current row in the left-hand set is detected when reached by the pointer that is coursing the right-hand set, the sets at the two pointers are merged and the result is added to the output set:

Figure 21.1 Internal cursors for a two-table join



Joining

Joining is used in `SELECT` statements to generate denormalized sets containing columns from multiple tables that store related data. The sets of columns extracted from each table are known as streams. The joining process merges the selected columns into a single output set. The simplest case is a two-table join, where two tables are linked by matching elements of a key that occurs in both tables.

The key columns for making the linkage are chosen on the basis that they implement the relationship unambiguously—that is, a key in the left-side table of the join will find only the rows in the right-side table that belong to the relationship, and it will find all of them. Typically, the linkage is made between a unique key in the left-side table (primary key or another unique key) and a formal or implied foreign key in the right-side table.

However, join conditions are not restricted to primary and foreign key columns and the engine does not refuse to output duplicate rows. Duplicate rows may cause undesirable results—refer to the note on this subject in the topic .

The INNER join

The following statement joins two tables that are related by a foreign key, `FK`, on the right-side table (`Table2`) that links to the primary key (`PK`) of `Table1`:

```
SELECT
  Table1.PK,
  Table1.COL1,
  Table2.PKX,
  Table2.COLX
FROM Table1
[INNER] JOIN Table2
  ON Table1.PK = Table2.FK
WHERE... <search-conditions>
```

This is a specification for an *inner join*. The keyword `INNER` is optional. We will examine the outer join presently. Figure 21.1 shows the two streams as they exist in the tables and the set that is generated.

Figure 21.2 Inner join

PK	COL1	PKX	COLX	FK
99	abc	234	AAA	99
100	bcd	235	BBB	101
101	cde	236	CCC	102
102	def	237	DDD	104
103	efg	238	EEE	105
104	fgh	239	FFF	106
105	ghi	240	GGG	109
106	hij	241	HHH	111
107	ijk	242	III	114
108	klm	243	JJJ	115
109	klm	244	KKK	116

PK	COL1	PKX	COLX
99	abc	234	AAA
101	cde	235	BBB
102	def	236	CCC
104	fgh	237	DDD
105	ghi	238	EEE
106	hij	239	FFF
109	klm	240	GGG

As the diagram indicates, the inner join merges the two streams so that any non-matching rows in either stream are discarded. Another name for the inner join is *exclusive join*, because its rules stipulate that non-matching pairs in both streams be excluded from the output.

Implicit join vs explicit JOIN

The SQL standards describe two syntaxes for the inner join. The example above uses the more modern SQL-92 *explicit* join syntax, distinguished from the older, more limited SQL-89 implicit join, because it uses an explicit JOIN clause to specify the join conditions.

SQL-89 implicit INNER JOIN syntax

Under SQL-89, the tables participating in the join are listed as a comma-separated list in the FROM clause of a SELECT query. The conditions for linking tables are specified among the search conditions in the WHERE clause. There is no special syntax to indicate which conditions are for searching and which are for joining. The join conditions are assumed to be self-evident. In hindsight, with the introduction of the JOIN clause, the old syntax came to be called *implicit* join syntax.

The implicit join syntax can implement only the inner join—SQL implementations that do not support a JOIN clause cannot do outer joins.

Here is the example above, rewritten as an implicit join:

```
SELECT
  Table1.PK,
  Table1.COL1,
  Table2.PKX,
  Table2.COLX
FROM Table1
WHERE Table1.PK = Table2.FK
AND <search-conditions>
```

The implicit join is supported in Firebird for compatibility with legacy application code. It is not recommended for new work because it is inconsistent with the syntaxes for other styles of joins, making maintenance and self-documentation unnecessarily awkward. Some data access software, including drivers, may not handle the SQL-89 syntax well because of parsing problems distinguishing join and search conditions. It can be anticipated that it will be dropped from future standards.

SQL-92 explicit INNER JOIN syntax

The explicit inner join is preferred for Firebird and other RDBM systems that support it. If the optimizer is allowed to compute the query plan, there is no reason why the SQL-92

syntax would perform better or worse than the older syntax, since the DSQL interpreter translates either statement into an identical binary language form for analysis by the optimizer.

Explicit joining makes statement code more readable and consistent with other styles of join supported by SQL-92 and subsequent standards. It is sometimes referred to as conditional join syntax because the JOIN...ON clause structure enables the join conditions to be distinguished from search conditions. Not surprisingly, this usage of the term “conditional” can be confusing!



The keyword INNER is entirely optional and is usually omitted. Alone, JOIN means exactly the same as INNER JOIN. (If JOIN is preceded by LEFT, RIGHT or FULL, then it is not an inner join!)

Three or more streams

If there are more than two streams (tables), just continue to add JOIN...ON clauses for each relationship. The following example adds a third stream to the previous example, joining it to the second stream through another foreign key relationship:

```
SELECT
    Table1.PK,
    Table1.COL1,
    Table2.PK,
    Table2.COLX,
    Table3.COLY
FROM Table1
    JOIN Table2 ON Table1.PK = Table2.FK
    JOIN Table3 ON Table2.PKX = Table3.FK
WHERE Table3.STATUS = 'SOLD'
AND <more-search-conditions>
```

Multi-column key linkages

If a single relationship is linked by more than one column, use the keyword AND to separate each join condition, just as you would do in a WHERE clause with multiple conditions. Take, for example, a table TableA, having a primary key (PK1, PK2), being linked to by TableB through a foreign key (FK1, FK2):

```
SELECT
    TableA.COL1,
    TableA.COL2,
    TableB.COLX,
    TableB.COLY
FROM TableA
    JOIN TableB ON TableA.PK1 = TableB.FK1
    AND TableA.PK2 = TableB.FK2
WHERE...
```

Mixing implicit and explicit syntaxes

Writing statements that include a mixture of implicit and explicit syntaxes is illegal in all Firebird versions except Firebird 1.0.x (where it is discouraged!). The following is an example of how NOT to write a join statement:

BAD Example! SELECT


```

Table1.PK,
Table1.COL1,
Table2.PK,
Table2.COLX,
Table3.COLY
FROM Table1, Table2
    JOIN Table3 ON TABLE1.PK = Table3.FK
    AND Table3.STATUS = 'SOLD' /* this is a search condition !! */
WHERE Table1.PK = Table2.FK
AND <more-search-conditions>

```

CROSS JOIN

A CROSS JOIN produces an output set that is the Cartesian product of two tables. That is, for each row in the left stream, an output stream will be generated for each row in the right stream:

```

SELECT
    Table1.PK,
    Table1.COL1,
    Table2.PKX,
    Table2.COLX
FROM Table1 CROSS JOIN Table2
WHERE... <search-conditions>

```

Explicit join syntax is not available (an ON clause would not make sense logically) so any limits must be set in WHERE conditions. The author's view is that a cross join with a WHERE clause is no different from an INNER JOIN.

In reality, the situations where a cross join between two persistent tables would be useful are rare. However, since joins can be made between various set types besides tables—views, derived tables, common table expressions—a cross join has the potential to be useful to track, break out or “ungroup” data that is summarised into one table from multiple sources.

Firebird did not support CROSS JOIN prior to the “2” series. Derived tables and CTEs are not supported in older versions, either, making the notion of a Cartesian set somewhat academic. There are ways to get a Cartesian product with the v.1.X versions, if you must! You can use the deprecated SQL-89 syntax without any joining criteria in the WHERE clause. For example:

```
SELECT T1.*, T2.* FROM T1 TABLE1, T2 TABLE2;
```

Alternatively, you can do an explicit join on a condition that could never be false, such as

```
SELECT * T1.*, T2.* FROM T1 TABLE1
JOIN T2 TABLE2 ON 1=1;
```



The Firebird query engine sometimes forms cross joins internally, when constructing “rivers” during joining operations

OUTER joins

In contrast to the inner join, an outer join operation outputs rows from participating tables, even if no match is found in some cases. Wherever a complete matching row cannot be

formed from the join, the “missing” data items are output as NULL. Another term for an outer join is an *inclusive join*.

Each outer join has a left and right side, the left being the stream that appears to the left of the JOIN clause, the right being the stream that is named as the argument of the JOIN clause. In a statement that has multiple joins, “leftness” and “rightness” may be relative—the right stream of one join clause may be the left stream of another, typically where the join specifications are flattening a hierarchical structure.

“Leftness” and “rightness” are significant to the logic of outer join specifications. Outer joins can be left, right or full. Each type of outer join produces a different output set from the same input streams.

The keywords LEFT, RIGHT and FULL are sufficient to establish that the joins are “outer” and the keyword OUTER is an optional part of the syntax.

The LEFT [OUTER] JOIN

A LEFT OUTER JOIN causes the query to output a set consisting of fully populated columns where matching rows were found (as in the inner join) and also partly-populated rows for each instance where a right-side match is not found for the left-side key. The unmatchable columns are populated with NULL. Here is a LEFT JOIN statement using the same input streams as our INNER JOIN example.

```
SELECT
  Table1.PK,
  Table1.COL1,
  Table2.PKX,
  Table2.COLX
FROM Table1 LEFT OUTER JOIN Table2
  ON Table1.PK = Table2.FK
WHERE... <search-conditions>
```

Figure 21.3 Left join

PK	COL1	PKX	COLX	FK	PK	COL1	PKX	COLX
99	abc	234	AAA	99	99	abc	234	AAA
100	bcd	235	BBB	101	100	bcd	NULL	NULL
101	cde	236	CCC	102	101	cde	235	BBB
102	def	237	DDD	104	102	def	236	CCC
103	efg	238	EEE	105	103	efg	NULL	NULL
104	fgh	239	FFF	106	104	fgh	237	DDD
105	ghi	240	GGG	109	105	ghi	238	EEE
106	hij	241	HHH	111	106	hij	239	FFF
107	ijk	242	III	114	107	ijk	NULL	NULL
108	lkl	243	LLL	115	108	lkl	NULL	NULL
109	klm	244	KKK	116	109	klm	240	GGG
Table1		Table2			Joined set			

This time, instead of discarding the left stream rows for which there is no match in the right stream, the query creates a row containing the data from the left stream and NULL in each of the specified right-stream columns.

The RIGHT [OUTER] JOIN

A RIGHT OUTER JOIN causes the query to output a set consisting of fully populated columns where matching rows were found (as in the inner join) and also partly-populated rows for each instance where a right-side row exists with no corresponding set in the left-side stream. The unmatchable columns are populated with NULL. Here is a RIGHT JOIN

statement using the same input streams as our INNER JOIN example. The optional keyword OUTER is omitted here:

```
SELECT
    Table1.PK,
    Table1.COL1,
    Table2.PKX,
    Table2.COLX
FROM Table1 RIGHT JOIN Table2
    ON Table1.PK = Table2.FK
WHERE... <search-conditions>
```

Figure 21.4 Right join

PK	COL1	PKX	COLX	FK	PK	COL1	PKX	COLX
99	abc	234	AAA	99	99	abc	234	AAA
100	bcd	235	BBB	101	101	cde	235	BBB
101	cde	236	CCC	102	102	def	236	CCC
102	def	237	DDD	104	104	fgh	237	DDD
103	efg	238	EEE	105	105	ghi	238	EEE
104	fgh	239	FFF	106	106	hij	239	FFF
105	ghi	240	GGG	109	109	klm	240	GGG
106	hij	241	HHH	111	NULL	NULL	241	HHH
107	ijk	242	III	114	NULL	NULL	242	III
108	jkl	243	JJJ	115	NULL	NULL	243	JJJ
109	klm	244	KKK	116	NULL	NULL	244	KKK

Instead of discarding the right stream rows for which there is no match in the left stream, the query creates a row containing the data from the right stream and NULL in each of the specified left-stream columns.

The FULL [OUTER] JOIN

The full outer join is fully inclusive—it returns one row for each pair of matching streams and one partly-populated row for each row in each stream where matches are not found. It combines the behaviors of the right and left joins. Here is the statement, using the same input streams as our INNER JOIN example:

```
SELECT
    Table1.PK,
    Table1.COL1,
    Table2.PKX,
    Table2.COLX
FROM Table1 FULL JOIN Table2
    ON Table1.PK = Table2.FK
WHERE... <search-conditions>
```

Figure 21.5 illustrates the conjunction of the streams for a FULL JOIN:

Figure 21.5 Full join

PK	COL1	PKX	COLX	FK
99	abc	234	AAA	
100	bcd		NULL	99
101	cde	235	BBB	101
102	def	236	CCC	102
103	efg	237	DDD	104
104	fgh	238	EEE	105
105	ghi	239	FFF	106
106	hij	240	GGG	109
107	ijk	241	HHH	111
108	jkl	242	III	114
109	klm	243	JJJ	115
		244	KKK	116

PK	COL1	PKX	COLX
99	abc	234	AAA
100	bcd		NULL
101	cde	235	BBB
102	def	236	CCC
103	efg		NULL
104	fgh	237	DDD
105	ghi	238	EEE
106	hij	239	FFF
107	ijk		NULL
108	jkl		NULL
109	klm	240	GGG
		241	HHH
		242	III
		243	JJJ
		244	KKK

PK	COL1	PKX	COLX
99	abc	234	AAA
100	bcd		NULL
101	cde	235	BBB
102	def	236	CCC
103	efg		NULL
104	fgh	237	DDD
105	ghi	238	EEE
106	hij	239	FFF
107	ijk		NULL
108	jkl		NULL
109	klm	240	GGG
		241	HHH
		242	III
		243	JJJ
		244	KKK

The result set contains rows for all rows in each stream, whether matched or not. NULLs are returned to “fill out” the unmatched rows from both the left and right streams.

Equi-joins

With the introduction of two styles of run-time set objects in the “2” series—derived tables in v.2.0 and common table expressions (CTEs) in v.2.1—Firebird has provided support for two styles of “equi-join” that might prove handy when joining with these run-time objects. Equi-joins detect eligible rows for merging between cursor sets by matching the names of columns.

Usage advice for equi-joining

As a general rule, equi-join queries will have direct equivalents in regular INNER or OUTER joins. However, unless care is taken to identify the sources of the retrieved data, there is the potential for anomaly. For example, when SELECT * is used instead of column lists to define the output set, the query returns only one column for each of the USING columns. NULL data in the USING columns sometimes has unexpected results when the join is inclusive.

Using aliases for the sets is optional but strongly recommended to ensure that you get the result you expect.

Named Columns JOIN

In this form of join, supported from v.2.1 onward, an exclusive (INNER) or inclusive (OUTER) equi-join is formed between two tables or sets by matching the names of *specific* columns in each. The join criteria are limited to specific columns listed in a USING clause. The paired columns must, of course, have identical names and compatible data types.

Syntax SELECT ...
 FROM <set-1> [[AS] <alias>]
 [[INNER] | {LEFT | RIGHT | FULL [OUTER]}] JOIN <set-2> [[AS] <alias>]
 USING (column-name [, column-name ...])
 ...

Note Either <set-1> or <set-2> can be a table, view, derived table or common table expression (CTE). For details of these derivative objects, refer to Chapter 23, **Views and Other Run-time Set Objects**.

Example

```
SELECT
    T.TRANSACTION_ID,
    TT.DESRIPTION
FROM TRANSACTIONS T JOIN TRANSAC_TYPE TT
    USING (TRANSAC_TYPE)
```

Its equivalent in regular JOIN syntax:

```
SELECT
    T.TRANSACTION_ID,
    TT.DESRIPTION
FROM TRANSACTIONS T JOIN TRANSAC_TYPE TT
    ON TT.TRANSAC_TYPE = T.TRANSAC_TYPE
```

Natural JOIN

Versions from v.2.1 onward support *automatic* equi-joining between sets that have common column names of compatible types. It is called a NATURAL JOIN. No ON or USING clause is involved. If the two sets have no column names in common, a CROSS JOIN is performed instead.

Syntax

```
SELECT ...
FROM <set-1> [[AS] <alias>]
NATURAL [[INNER] | {LEFT | RIGHT | FULL [OUTER]}] JOIN <set-2> [[AS] <alias>]
...
```

Note *Either <set-1> or <set-2> can be a table, view, derived table or common table expression (CTE). Views and these other derivative table-like objects are discussed in the next chapter.*

Example This query works just like the previous example for the Named Columns equi-join, given that the column TRANSAC_TYPE is common to both tables:

```
SELECT
    T.TRANSACTION_ID,
    TT.DESRIPTION
FROM TRANSACTIONS T NATURAL JOIN TRANSAC_TYPE TT
```

Re-entrant joins

Design conditions sometimes require forming a joined set from two or more streams that come from the same table. Commonly, a table has been designed with a hierarchical or tree structure, where each row in the table is logically the “child” of a “parent”. The table’s primary key identifies the child-level node of the tree, while a foreign key column stores a “parent” key referring to the primary key value of a different row.

A query to flatten the parent-child relationship requires a join that draws one stream for “parents” and another for “children” from the same table. The popular term for this is *self-referencing join*. The term *re-entrant join* is morphologically more appropriate, since there are other types of re-entrant query that do not involve joins. The [Subqueries](#) topic, later in this chapter, discusses these other types of re-entrant query.

Cursors for re-entrant joins

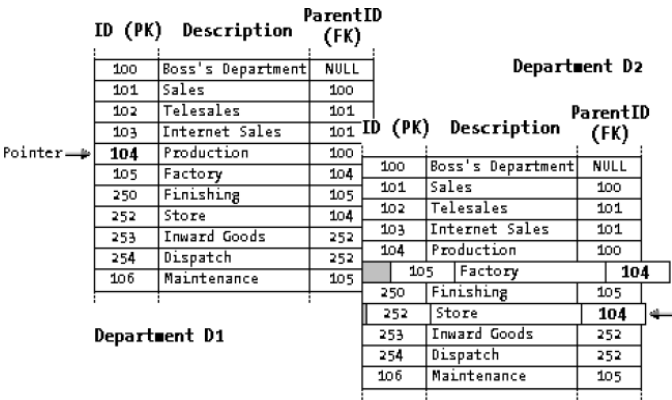
To perform the re-entrant join, the engine maintains one internal cursor for each stream, within the same table image. Conceptually, it treats the two cursor contexts as though they

were separate tables. In this situation, the syntax of the table references uses **mandatory aliasing** to distinguish the cursors for the two streams.

In the following example, each department in an organization is stored with a parent identifier that points to the primary key of its supervising department. The query treats the departments and the parent departments as if they were two tables:

```
SELECT
  D1.ID,
  D1.PARENTID,
  D1.DESCRPTION AS DEPARTMENT,
  D2.DESCRPTION AS PARENT_DEPT
FROM DEPARTMENT D1
LEFT JOIN DEPARTMENT D2 /* left join catches the root of the tree, ID 100 */
ON D1.PARENTID = D2.ID;
```

Figure 21.6 Internal cursors for a simple re-entrant join



The simple, two-layered output looks like this:

Figure 21.7 Output from the simple re-entrant join

ID	ParentID	Department	Parent_Dept
100	NULL	Boss's Department	NULL
101	100	Sales	Boss's Department
102	101	Telesales	Sales
103	101	Internet Sales	Sales
104	100	Production	Boss's Department
105	104	Factory	Production
250	105	Finishing	Factory
252	104	Store	Production
253	252	Inward Goods	Store
254	252	Dispatch	Store
106	105	Maintenance	Factory

The output from this query is very simple—a one-level denormalization. Tree output operations on tables with this kind of structure are often recursive, using stored procedures to implement and control the recursions.²

2. In SQL for Smarties, Joe Celko offers some interesting solutions for storing and maintaining tree structures in relational databases (Morgan Kaufmann, 1995, ISBN 1-55860-323-9).

Subqueries

A subquery is a special form of expression that is actually a `SELECT` query on another table, embedded within the main query specification. The embedded query expression is referred to variously as a subquery, a nested query, an in-line query and sometimes (erroneously) as a sub-select.

In Firebird, subquery expressions are used in four ways:

- 1 To provide a single or multi-row input set for an `INSERT` query. The syntax is described in the topic *[INSERT INTO](#)* in Chapter 19.
- 2 To specify a run-time, read-only output column for a `SELECT` query
- 3 To provide values or conditions for search predicates
- 4 In “2” series and beyond, to define a *derived table*. Refer to the topic *[Derived Tables](#)* in Chapter 23, ***Views and Other Run-time Set Objects***.

Specifying a column using a subquery

A run-time output column can be specified by querying out to a single column in another table. The output column should be given a new identifier that, for completeness, can optionally be marked by the `AS` keyword.

The nested query must always have a `WHERE` condition to restrict output to a single column from a single row—known as a scalar query—otherwise you will see some variant of the error message “Multiple rows in singleton select”.

This query uses a subquery to derive an output column:

```
SELECT
  LAST_NAME,
  FIRST_NAME,
  ADDRESS1,
  ADDRESS2,
  POSTCODE,
  (SELECT START_TIME FROM ROUTES
   WHERE POSTCODE = '2261' AND DOW = 'MONDAY') AS START_TIME
FROM MEMBERSHIP
WHERE POSTCODE = '2261';
```

The subquery targets single postcode in order to return the start time of a transport route. The `POSTCODE` columns in both the main query and the subquery could be substituted with replaceable parameters (one for each occurrence). To make the query more general and more useful, we can use a *correlated* subquery.

Correlated subqueries

When the data item extracted in the nested subquery needs to be selected in the context of a linking value in the current row of the main query, a correlated subquery is possible.



Firebird requires fully-qualified identifiers in correlated subqueries.

In the next example, the linking columns in the main query and the subquery are correlated and table aliasing is enforced to eliminate any ambiguity:

```
SELECT
    .LAST_NAME,
    M.FIRST_NAME,
    M.ADDRESS1,
    M.ADDRESS2,
    M.POSTCODE,
    (SELECT R.START_TIME FROM ROUTES R
     WHERE R.POSTCODE = M.POSTCODE AND M.DOW = 'MONDAY') AS START_TIME
FROM MEMBERSHIP M
WHERE...;
```

The query returns one row for each selected member, whether or not a transport route exists that matches the member's postcode. `START_TIME` will be `NULL` for non-matches.

Subquery or Join?

The query in the previous example could have been framed using a left join instead of the subquery:

```
SELECT
    M.LAST_NAME,
    M.FIRST_NAME,
    M.ADDRESS1,
    M.ADDRESS2,
    M.POSTCODE,
    R.START_TIME
FROM FROM MEMBERSHIP M
LEFT JOIN ROUTES R
    ON R.POSTCODE = M.POSTCODE
WHERE M.DOW = 'MONDAY';
```

The relative cost of this query and the preceding one, using the subquery, is similar. Although each may arrive at the result by a different route, both require a full scan of the searched stream with evaluation from the searching stream.

Cost differences can become significant when the same correlated subquery is used in place of an *inner* join:

```
SELECT
    M.LAST_NAME,
    M.FIRST_NAME,
    M.ADDRESS1,
    M.ADDRESS2,
    M.POSTCODE,
    R.START_TIME
FROM FROM MEMBERSHIP M
JOIN ROUTES R
    ON R.POSTCODE = M.POSTCODE
WHERE M.DOW = 'MONDAY';
```


The inner join does not have to traverse every row of the searched stream, since it discards any row in the searched stream (ROUTES) that does not match the search condition. In contrast, the context of the correlated subquery changes with each row, with no condition to exclude scanning any having a non-matching POSTCODE. Thus, the correlated subquery must be run once for each row in the set.

If the output set is potentially large, consider how important the need is to do an inclusive search. A well-judged correlated subquery is useful for small sets. There is no absolute threshold of numbers for choosing one over another. As always, testing under real-life load conditions is the only reliable way to decide what works best for your own particular requirements.

When NOT to consider a subquery

A subquery can return one and only one field. The subquery approach becomes outlandish when you need to fetch more than a single field from the *same* other table. To fetch multiple fields, a separate correlated subquery and alias is required for each field fetched. If a left join is possible to meet these conditions, it should always be chosen.

Searching using a subquery

The use of existential predicators with subqueries—especially the EXISTS() predictor—has already been discussed in the previous chapter. Subqueries can also be used in other ways to predicate search conditions for WHERE clauses and groupings.

Re-entrant (“self-referencing”) subqueries

A query can use a re-entrant subquery to set a search condition derived from the same table. **Table aliasing is mandatory.** In the following example, the statement subqueries the main table to find the date of the latest transaction, in order to set the search condition for the main query:

```
SELECT
    A1.COL1,
    A1.COL2,
    A1.TRANSACTION_DATE
FROM ATABLE A1
WHERE A1.TRANSACTION_DATE =
    (SELECT MAX(A2.TRANSACTION_DATE) FROM ATABLE A2);
```

Inserting using a subquery with joins

In Chapter 19, we examined the in-line select method of feeding data into an INSERT statement. For example:

```
INSERT INTO ATABLE (
    COLUMN2, COLUMN3, COLUMN4)
SELECT BCOLUMN, CCOLUMN, DCOLUMN FROM BTABLE
WHERE...;
```

The method is not limited to a single-stream query. Your input subquery can be joined. This capability can be very useful when you need to export denormalized data to an external table for use in another application, such as a spreadsheet, desktop database or off-the-shelf accounting application. For example:

```
INSERT INTO EXTABLE (
    TRANDATE, INVOICE_NUMBER, EXT_CUST_CODE, VALUE)
SELECT
    INV.TRANSACTION_DATE,
    INV.INVOICE_NUMBER,
    CUS.EXT_CUS_CODE,
    SUM(INVD.PRICE * INVD.NO_ITEMS)
FROM INVOICE INV
JOIN CUSTOMER CUS
    ON INV.CUST_ID = CUS.CUST_ID
JOIN INVOICE_DETAIL INVD
    ON INVD.INVOICE_ID = INV.INVOICE_ID
WHERE...
GROUP BY...;
```

The Derived Table

In the “2” series and beyond, it is possible to specify a set using a subquery and treat that set much like a relation object (such as a table or view) in the same query, viz.

```
SELECT
    <columns, etc.>
FROM (SELECT... ) AS SOME_NAME
...
```

It can be joined with another set and its fields can be treated like table columns inside expressions the search clauses.

Refer to the topic *Derived Tables* in Chapter 23, **Views and Other Run-time Set Objects** for a full description and directions for using them.

UNION Queries

The set operator UNION can be used to combine the output of two or more SELECT statements and produce a single, read-only set comprising rows derived from different tables, or from different sets queried from the same table.

Multiple queries are “stacked”, each subset specification being linked to the next by the UNION keyword. Data must be chosen and the query constructed in such a way as to ensure the contributing input sets are all *union compatible*.

Union compatible sets

For each SELECT operation contributing a stream as input to a UNION set, the specification must be a column list which exactly matches those of all of the others by degree (number and order of columns) and respective data type. For example, suppose we have these two table specifications:

```
CREATE TABLE CURRENT_TITLES (
    ID INTEGER NOT NULL,
```

```

    TITLE VARCHAR(60) NOT NULL,
    AUTHOR_LAST_NAME VARCHAR(40),
    AUTHOR_FIRST_NAMES VARCHAR(60),
    EDITION VARCHAR(10),
    PUBLICATION_DATE DATE,
    PUBLISHER_ID INTEGER,
    ISBN VARCHAR(15),
    LIST_PRICE DECIMAL(9,2));
/**/
CREATE TABLE PERIODICALS (
    PID INTEGER NOT NULL,
    PTITLE VARCHAR(60) NOT NULL,
    EDITOR_LAST_NAME VARCHAR(40),
    EDITOR_FIRST_NAMES VARCHAR(60),
    ISSUE_NUMBER VARCHAR(10),
    PUBLICATION_DATE DATE,
    PUBLISHER_ID INTEGER,
    LIST_PRICE DECIMAL(9,2));

```

The tables are union-compatible because we can query both to obtain sets with matching “geometry”:

```

SELECT
    ID,
    TITLE,
    AUTHOR_LAST_NAME,
    AUTHOR_FIRST_NAMES,
    EDITION VARCHAR(10),
    LIST_PRICE
FROM CURRENT_TITLES
UNION
SELECT
    ID,
    TITLE,
    EDITOR_LAST_NAME,
    EDITOR_FIRST_NAMES,
    ISSUE_NUMBER,
    LIST_PRICE
FROM PERIODICALS;

```

A `UNION` having `SELECT * FROM` either table would not work because the table structures are different—the second table has no `ISBN` column.

UNION ALL | DISTINCT

If duplicate rows are formed during the creating of the union set, the default behavior is to exclude the duplicate rows from the set. To include the duplicates, use `UNION ALL` instead of `UNION` on its own.

From the “2” series onward, the default behaviour can be optionally marked by the keyword `DISTINCT`. It does not change the way a `UNION` query works: it is merely “window-dressing” that adds a minor piece of missing compliance to the DML lexicon:

Syntax `SELECT (...) FROM (...)`
 `UNION [DISTINCT | ALL]`
 `SELECT (...) FROM (...)`

v.1.X `UNION DISTINCT` is not supported in Firebird 1.X versions and will incur an exception if used in those environments.

Using run-time columns in unions

The column identifiers of the entire output set—regardless of how many `UNION`s are present—are determined by those first `SELECT` specification. If you would like to use alternative column names, column aliasing can be used in the output list of the first `SELECT` specification. Additionally, if needed, run-time fields derived from constants or variables can be included in the `SELECT` clause of each contributing stream. The next query provides a much more satisfactory listing of publications from these two tables:

```
SELECT
  ID,
  TITLE as PUBLICATION,
  'BOOK      ' AS PUBLICATION_TYPE,
  CAST (AUTHOR_LAST_NAME || ', ' || AUTHOR_FIRST_NAMES AS VARCHAR(50))
    AS AUTHOR_EDITOR,
  EDITION AS EDITION_OR_ISSUE,
  PUBLICATION_DATE DATE,
  PUBLISHER_ID,
  CAST(ISBN AS VARCHAR(14)) AS ISBN,
  LIST_PRICE
FROM CURRENT_TITLES
WHERE ...
UNION
SELECT
  ID,
  TITLE,
  'PERIODICAL',
  EDITOR_LAST_NAME || ', ' || EDITOR_FIRST_NAMES AS AUTHOR_EDITOR,
  CAST (AUTHOR_LAST_NAME || ', ' || AUTHOR_FIRST_NAMES AS VARCHAR(50)),
  ISSUE_NUMBER,
  PUBLICATION_DATE,
  PUBLISHER_ID,
  'Not applicable',
  LIST_PRICE
FROM PERIODICALS
WHERE ...
ORDER BY 2;
```

Search and ordering conditions

Notice in the example that search conditions are possible in each contributing `SELECT` specification. They are normal search expressions, which must be confined to the contributing set controlled by the current `SELECT` expression. There is no way to correlate search conditions across the boundaries of the subsets.

Only one ordering clause is allowed and it must follow *all* subsets. The `ORDER BY` degree syntax, i.e. ordering by column position number, is required for ordering union sets.

Re-entrant UNION queries

It is possible to apply a re-entrant technique to produce a union query that draws multiple subsets from a single table. Table aliases are required in the `FROM` clauses but column references need not be fully qualified. Returning to our `CURRENT_TITLES` table, suppose we want a list that splits out the book titles according to price range. The re-entrant query could be something like this:

```
SELECT
    ID,
    TITLE,
    CAST('UNDER $20' AS VARCHAR(10)) AS RANGE,
    CAST (AUTHOR_LAST_NAME || ', ' || AUTHOR_FIRST_NAMES AS VARCHAR(50))
    AS AUTHOR,
    EDITION,
    LIST_PRICE
FROM CURRENT_TITLES CT1
    WHERE LIST_PRICE < 20.00
UNION
SELECT
    ID,
    TITLE,
    CAST('UNDER $40' AS VARCHAR(10)),
    CAST (AUTHOR_LAST_NAME || ', ' || AUTHOR_FIRST_NAMES AS VARCHAR(50)),
    EDITION,
    LIST_PRICE
FROM CURRENT_TITLES CT2
    WHERE LIST_PRICE >= 20.00 AND LIST_PRICE < 40.00
UNION
SELECT
    ID,
    TITLE,
    CAST('$40 PLUS' AS VARCHAR(10)),
    CAST (AUTHOR_LAST_NAME || ', ' || AUTHOR_FIRST_NAMES AS VARCHAR(50)),
    EDITION,
    LIST_PRICE
FROM CURRENT_TITLES CT3
    WHERE LIST_PRICE >= 40.00;
```

Recursive Queries

In versions prior to v.2.1, the only way to execute an operation recursively on the server is to write a procedural module that calls itself, or iteratively calls a sub-procedure, inside a controlled loop. This technique is described in **Chapter 29, Stored Procedures and Executable Blocks**.

From v.2.1 onward, Firebird supports *common table expressions* (CTEs). A CTE is a set defined in the preamble of query, using the keywords WITH [RECURSIVE] .. and then used in one or more ways during execution of the query. This useful mechanism has more usage capabilities than a derived table and is more akin to a view in the range of table-like operations it can be used with.

An important usage of the CTE comes in its ability to perform a DML operation recursively over discrete UNION sets extracted progressively from the same table. In many cases it is possible to replace a “heavy” recursive PSQL module with a WITH RECURSIVE.. DML statement that will lighten the overhead considerably.l

You can read about CTEs and their rules in in Chapter 23, **Views and Other Run-time Set Objects**, in the topic Common Table Expressions.

CHAPTER 22

ORDERED AND AGGREGATED SETS

In this chapter we take a look at the syntax and rules for specifying queries that output ordered and grouped sets.

Sets specified with `SELECT` are, by default, fetched in no particular order. Often, especially when displaying data to a user or printing a report, you need some form of sorting. A phone list is clearly more useful if surnames are in alphabetical order. Groups of sales figures or test results are more meaningful if they are collated, grouped and summarized. SQL provides two clauses for specifying the organization of output sets.

An `ORDER BY` clause can be included to sort sets lowest first (ascending order) or highest first (descending order), according to one or more of the set's columns. A `GROUP BY` clause can partition a set into nested groups—levels—according to columns in the `SELECT` list and, optionally, perform aggregating calculations on sets of numeric columns in the bounds of a group.

Considerations for Sorting

Although ordering and aggregation operations are two operations with distinctly different outcomes, they interact to some degree when both are used in a query and placement of their specifying clauses is important. Under the hood they share some characteristics with respect to forming intermediate sets and using indexes.

Presentation order of sorting clauses

The following abridged `SELECT` syntax skeleton shows the position of `ORDER BY` and `GROUP BY` clauses in an ordered or grouped specification. Both clauses are optional and both may be present:

```
SELECT [FIRST m] [SKIP N] | [DISTINCT | ALL ]
    {<column-list>}
FROM <table-specification>
```

```
[WHERE <search-condition>]
[GROUP BY <grouping-item> [COLLATE collation ]
[,<grouping-item> [COLLATE collation ]...]]
[HAVING <search-condition>]
[UNION [ALL ]<select-expression>]
[PLAN <plan-expression>]
[ORDER BY <list of sorting items>]
[FOR UPDATE [OF col [,col ...]] [WITH LOCK]];
```

Temporary sort space

Queries with ORDER BY or GROUP BY clauses “park” the intermediate sets for sorting operations in temporary storage space, in memory if enough is available, otherwise in temporary files on disk. It is recommended to make approximately 2.5 times the size of the largest table you will sort available in the temporary disk space. The sort spaces in both RAM and on disk can be configured. The engine will not split a sort file between RAM and disk.

Sort memory

The default block size of sort memory at 1 Mb—this is the size of each chunk of RAM that the server will allocate, up to a default maximum of 64 Mb on Superserver and 8 Mb on Classic. Both of these values can be reconfigured by means of the configuration parameters **TempBlockSize** and **TempCacheLimit**, respectively, in `firebird.conf`.



*In versions prior to v.2.1, the TempBlockSize parameter is called **SortMemBlockSize**, while TempCacheLimit is called **SortMemUpperLimit**.*



Don't overstretch the memory resources of sort cache on a Classic or SuperClassic host machine. Since Classic and Superclassic spawn a separate set of resources for each connection, an upper limit that is too high will cause huge amounts RAM to be consumed in a busy system.

For more information about the configuration, refer to [TempBlockSize](#) and [TempCacheLimit](#) in Chapter. 34, **Configuration Parameters in Detail**.

Sort space on disk

The default installation does not configure any explicit sort space on disk. If sort space on disk is not configured, the engine will use the `/tmp` filesystem on POSIX or the directory pointed to by the environment variables `TMP` and/or `TEMP` on Windows by default. It is strongly recommended that you configure sort space explicitly for Firebird's use.

You can configure sort space in two ways. The first is to set up a directory location using the environment variable `FIREBIRD_TMP` (`INTERBASE_TMP` for v.1.0.x) pointing to the directory where the temporary files are to go. The second is to configure the directories using the configuration parameter **TempDirectories** in `firebird.conf`.

v.1.0.x In Firebird 1.0.x, add one or more **temp_directory** entries in `isc_config` (POSIX) or `ibconfig` (Windows).

For more information about the configuration, refer to [TempDirectories](#) in Chapter. 34, **Configuration Parameters in Detail**.

Indexing

Ordered sets are costly to server resources and to performance in general. When evaluating a query and determining a plan, the optimizer has to choose between three methods of accessing the sets of data—known as streams—that are contributed by the specified tables: NATURAL (search in no particular order), INDEX (use an index to control the search) and MERGE (form bitmap images of the two streams and merge them on a one-to-one basis).

When a set is read in index order, the read order is contrived, i.e. it is in out-of-storage-order. The likelihood that indexed reads will involve processing multiple pages is very high and the cost grows with the size of the table. In the worst case, I/O operations will increase.

With the MERGE (also referred to as SORT) method, each row (and thus, each page) is read only once and the reads are in storage order. The MERGE method is quicker when the intermediate sets can be processed in RAM.

It is right to emphasise the importance of good indexing to speed up and rationalize the sorting process in ordered queries. In ordered sets, if conditions are right, the INDEX method with a good index will speed the output of the first rows. However, the index can also slow down fetching considerably. The cost may be that the whole set is retrieved more slowly than when the alternative MERGE access method is used. With MERGE, the first row is found more slowly but the whole set is retrieved faster.

The same trade-off does not necessarily apply to grouped queries, since they require the full set to be fetched at the server before any row can be output. With grouping conditions, it may transpire that an index worsens, rather than improves, performance.



To ascertain the effectiveness of indexes for sorting operations there is no substitute for testing with realistic sets of typical data.

The ORDER BY Clause

The ORDER BY clause is used for sorting the output of a SELECT query and is valid for any SELECT statement that is capable of retrieving multiple rows for output. It is placed after all other clauses except a FOR UPDATE clause (if any) or an INTO clause (in a stored procedure).

The general form of the ORDER BY clause syntax is

```
...
ORDER BY <list of sorting items>
<sorting item> = <column> | <expression> | <degree number>
ASC | DESC
[NULLS LAST | NULLS FIRST]
```

Sorting items

The comma-separated **<list of sorting items>** determines the vertical sort order of the rows. Sorting for each sorting item can be in ascending (ASC) or descending (DESC) order. Ascending order is the default and need not be specified.

Sorting precedence

When there are multiple sorting items, horizontal position of items in the clause is significant—the sorting precedence is from left to right.

Columns

Sorting items are commonly columns. Indexed columns are usually sorted much faster than non-indexed. If a sort involves multiple columns in an unbroken left-to-right sequence, a compound index made up of the sorting items in the same left-to-right sequence and the appropriate sort direction (ASC/DESC) could speed up the sort dramatically.

In UNION and GROUP BY queries, the column used for sorting has to be present in the output list. In other queries, non-output columns and expressions on non-output columns are valid ordering criteria.

Example 1 Simple sorting by columns:

```
SELECT
  COLA,
  COLB,
  COLC,
  COLD
FROM TABLEA
ORDER BY COLA, COLD;
```

Use of relation aliases

When specifying an ordered set that specifies joining or a correlated subquery, for all sorting items that are database columns or expressions referring to database columns, you must provide either fully-qualified identifiers or aliases, using the same table identifiers or aliases that were initialized in the FROM clause.



Do not mix table or view name identifiers and aliases in the same query. From v.2.1 onward, this inadvisable practice will incur an exception.

Example 2 Sorting with aliases:

```
SELECT
  A.COLA,
  A.COLB,
  B.COL2,
  B.COL3
FROM TABLEA A
JOIN TABLEB B
  ON A.COLA = B.COL1
WHERE A.COLX = 'Sold'
ORDER BY A.COLA, B.COL3;
```

Expressions

Valid expressions are allowed as sort items, except in v.1.0.x. An expression can specify an ordering item even if the expression is not output as a run-time column. A grouping expression has to be in the SELECT list.

You can sort sets on internal or external function expressions or correlated subqueried scalars. If the expression column you want to sort on is present in the output list, you can't use its alias name as the ordering item—either repeat the total expression in the ORDER BY clause or use the column's degree number (see below).

Example 3 Ordering by expression:

```
SELECT EMP_NO || '-' || SUBSTRING(LAST_NAME FROM 1 FOR 3) AS NAMECODE
FROM EMPLOYEE
ORDER BY EMP_NO || '-' || SUBSTRING(LAST_NAME FROM 1 FOR 3);
```

Alternatively, using the degree number in the ordering clause:

```
SELECT EMP_NO || '-' || SUBSTRING(LAST_NAME FROM 1 FOR 3) AS NAMECODE
FROM EMPLOYEE
ORDER BY 1;
```

v.1.0.x If you need to sort by an expression in version 1.0.x, this is the only way to do it, i.e., it is necessary to include the expression in the output list and use its degree number in the ORDER BY clause.

Unless you are using v.1.0.x, it is not mandatory to include the ordering expression in the output list:

```
SELECT
  LAST_NAME,
  FIRST_NAME,
  EMP_NO
FROM EMPLOYEE
ORDER BY EMP_NO || '-' || SUBSTRING(LAST_NAME FROM 1 FOR 3);
```



When using expressions as ordering criteria, bear in mind the considerable extra load placed on server resources by the doubling of the expression calculation for each row retrieved. The impact is, at best, unpredictable. Be conservative about the cardinality of the sets that you expose to this kind of operation.

A sensible sort order

A sort on values returned from an expression is just like any other sort—it will follow the regular ordinal rules for the data type of the returned field. It is your responsibility to specify sorting only on values that accord to some logical sequence. For an example of an inappropriate sequence, consider

```
...
ORDER BY CAST(SALES_DATE AS VARCHAR(24))
```

Rows would be ordered in the alphabetical collation sequence of the characters, with no relationship to the dates on which the expression operated.

Degree number

The degree number of the output column list is the position of the column in the output, counting from left to right, starting at 1. It can be used as a special kind of expression when using an expression item from the SELECT list as a sorting criterion.

Example 4 Respecifying the simple query in Example 1, using degree numbers instead of column identifiers:

```
SELECT
  COLA,
  COLB,
```

```
COLC,
COLD
FROM TABLEA
ORDER BY 1, 4;
```

Ordering a UNION

An ordering clause for sorting the output of a UNION query can use only degree numbers to refer to the ordering columns:

Example 5 Ordering UNIONed sets

```
SELECT
  T.FIRST_NAME,
  T.LAST_NAME,
  'YES' AS "TEAM LEADER?"
FROM EMPLOYEE T
WHERE EXISTS (
  SELECT 1 FROM PROJECT P1
  WHERE P1.TEAM_LEADER = T.EMP_NO)
UNION
SELECT
  E.FIRST_NAME,
  E.LAST_NAME,
  'NO ' AS "TEAM LEADER?"
FROM EMPLOYEE E
WHERE NOT EXISTS (
  SELECT 1 FROM PROJECT P2
  WHERE P2.TEAM_LEADER = E.EMP_NO)
ORDER BY 2, 1;
```



Sorting by degree number does not speed up the query: the engine recalculates expressions for the sorting operation.

Using the degree number, even when it is not mandatory, is considered useful to save typing and avoid clutter, especially when the output set involves joins.

Sort direction

By default, sorts are performed in ascending order. To have the sort performed in descending order, include the keyword DESC. If you need a descending sort, create a descending index for it: Firebird cannot "invert" an ascending index and use it for a descending sort, nor use a descending index for an ascending sort.

Example 6 Sorting in descending order:

```
SELECT FIRST 10 * FROM DOCUMENT
ORDER BY STRLEN(DESCRIPTION) DESC
```

NULLS placement

By default, Firebird positions sort columns having null in the sort column at the bottom of the output set. The NULLS FIRST keyword can be used to specify that nulls be placed first, at the top of the set.



An index can not be used on any sorting item if NULLS FIRST is specified.

Example 7 In this query, any rows with a NULL in PHONE_EXT will be output ahead of those with non-null:

```
SELECT * FROM EMPLOYEE
ORDER BY PHONE_EXT NULLS FIRST
```

v.1.0.x NULLS placement is not supported in Firebird 1.0.x.

The GROUP BY Clause

When a query includes a GROUP BY clause, the output of the column and table specification, viz.

```
SELECT {<column-list>
FROM <table-specification>
[WHERE <search-condition>]
```

gets passed to a further stage of processing, where the rows are partitioned into one or more nested groups. Each partition typically extracts summaries by way of expressions that perform some operation on groups of numeric values. This type of query is known as a grouped query and its output is often referred to as a grouped set.

Syntax

```
SELECT <groupable-field-list> FROM <table-specification>
[WHERE...]
GROUP BY <grouping-item> [COLLATE collation-sequence] [, <grouping-item [...]]
HAVING <grouping-column predicate>
[ORDER BY ...];
```

The groupable field list

A group is formed by collecting together—aggregating—all of the rows where a column named in both the column list and the GROUP BY clause share a common value. The logic of aggregation means that the field list specified by SELECT for a grouped query is tightly restricted by the fields named as arguments in the GROUP BY clause. Fields having specifications that meet these requirements are often referred to as *groupable*. If you specify a column or field expression that is not groupable, the query will be rejected.

- A database column or non-aggregating expression can not be specified in the column list if it is not also specified in the GROUP BY clause
- An aggregating expression operating on a database column that is not in the GROUP BY clause can be included in the column list. The use of an alias for the result of the expression is strongly recommended

v.1.0.x In Firebird 1.0.x the enforcement of the groupability requirement is less restrictive. Grouped queries in client SQL calls and stored procedures may cause exceptions that did not occur previously. It is a long-standing bug that you should be aware of when migrating your application's back-end to any Firebird version from v.1.5 onward.

Aggregating expressions

Firebird has a group of aggregating functions that are typically used in combination with grouping conditions to calculate levelled totals and statistics.

The aggregating functions are SUM() which calculates totals, MAX() and MIN() returning the highest and lowest values, respectively, and AVG() which calculates averages. The COUNT() function also behaves as an aggregating function in grouped queries, providing a row count for all row members of the lowest level group. LIST(), added to the lexicon at v.2.1, gathers all of the values for a column in a set and returns them in a comma-separated stream.

Unlike other groupable items, an aggregating expression from the SELECT list can not be used as a grouping item (see below), since it outputs a value that is calculated from values beneath the context of the group.

The table PROJ_DEPT_BUDGET contains rows intersecting projects and departments. We are interested in retrieving a summary of the budgets allocated to each project, regardless of department. The following item list—which will be explored later in this topic—specifies a field list of the two items we want:

```
SELECT
    PROJ_ID,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
GROUP BY PROJ_ID;
```

The two field specifications are going to be fine as groupable items. The department id (DEPT_NO) is not in the list, because it is the project totals that are wanted. To get them, we make PROJ_ID the argument of the GROUP BY clause.

On the other hand, if we wanted to list the department budgets, regardless of project, the field list would be set up for DEPT_NO to be the argument of the GROUP BY clause:

```
SELECT
    DEPT_NO,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
GROUP BY DEPT_NO;
```

Effects of NULL in aggregations

In aggregate expressions using operators like SUM() and AVG() and COUNT(<specific_column_name>), rows containing NULL in the targeted column are ignored for the aggregation. AVG() forms the numerator by aggregating the non-null values and the denominator by counting the rows containing non-null values.



If you have columns on which you want to calculate averages, it will be important at design-time to consider whether you want average calculations to treat “empty” instances

as NULL (and be excluded from the calculation) or as zero. You can implement a rule by means of a default or, better, by a Before Insert trigger.

The grouping item

The GROUP BY clause takes one or a list of grouping items. A grouping item can be:

- a column name
- degree number, an ordinal number representing the left-to-right position of the corresponding item in the SELECT field list, parallel to the existing option for ORDER BY arguments.
- an expression using a function such as CAST(), EXTRACT(), SUBSTRING(), UPPER(), CASE(), COALESCE().



A replaceable parameter cannot be a grouping item.

v.1.0.x In Firebird 1.0.x, a grouping item can be only a column name or an appropriate external function (UDF) expression.

The following statement completes the query begun by the previous example:

```
SELECT
    PROJ_ID,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
GROUP BY PROJ_ID;
```

PROJ_ID	TOTAL_BUDGET
=====	=====
GUIDE	650000.00
HWR11	520000.00
MAPDB	111000.00
MKTPR	1480000.00
VBASE	2600000.00

Important restriction

A grouping item can not be any expression involving an aggregating function—such as AVG(), SUM(), MAX(), MIN() OR COUNT()—that would aggregate in the same grouping context (level) as any grouping item. This restriction includes any aggregate expressions that are embedded inside another expression. For example, the DSQL parser will complain if you attempt this:

```
SELECT
    PROJ_ID,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
GROUP BY 2;
```

ISC ERROR CODE:335544569

Cannot use an aggregate function in a GROUP BY clause

Non-aggregating expressions

The ability to use internal and external function expressions for grouping opens up a wide range of possibilities for “massaging” stored attributes to generate output sets that would not be possible otherwise.

For example, the internal function `EXTRACT()` operates on date and time types, to return “date-parts”—numbers that isolate the year, month, day, hour, etc. parts of temporal types. The following example queries a Membership table and outputs a statistic showing how many members joined in each month, by month, regardless of the year or day of their join dates:

```
SELECT
    MEMBER_TYPE,
    EXTRACT(MONTH FROM JOIN_DATE) AS MONTH_NUMBER, /* 1, 2, etc. */
    COUNT (JOIN_DATE) AS MEMBERS_JOINED
FROM MEMBERSHIP
GROUP BY
    MEMBER_TYPE,
    EXTRACT(MONTH FROM JOIN_DATE);
```

A plethora of useful functions is available for transforming dates, strings and numbers into items for grouping. The following examples illustrate grouping by the results of function expressions:

```
SELECT
    CHAR_LENGTH (TRIM TRAILING (RDB$RELATION_NAME)) AS LEN,
    COUNT (*)
FROM RDB$RELATIONS
GROUP BY 1
ORDER BY 2;
```

The internal function `S TRIM` and `CHAR_LENGTH` were not available prior to v.2.0 so the following is a similar query that would achieve similar results using external functions from *ib_udf*:

```
SELECT
    STRLEN(RTRIM(RDB$RELATION_NAME)),
    COUNT(*)
FROM RDB$RELATIONS
GROUP BY STRLEN(RTRIM(RDB$RELATION_NAME))
ORDER BY 2;
```

It will work in any version of Firebird.

Grouping by degree (ordinal number)

Using the degree number of the output column in the `GROUP BY` clause ‘copies’ the expression from the select list (as does the `ORDER BY` clause). This means that, when a degree number refers to a subquery, the subquery is executed at least twice.

Before Firebird 2.0, some expressions are disallowed inside the `GROUP BY` list. For example, the parser rejects a grouping item that contains the concatenation symbol, “||”:

```
SELECT
```



```

    PROJ_ID || '-1994' AS PROJECT,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
    GROUP BY PROJ_ID || '-1994';

```

returns this exception in Firebird 1.5:

```

ISC ERROR CODE:335544569
Token unknown - line 6, char 21
||

```

Using the degree number of the expression field will work around the problem:

```

SELECT
    PROJ_ID || '-1994' AS PROJECT,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
    GROUP BY 1;

```



Grouping by degree number does not speed up the query: the engine recalculates expressions for the sorting operation.

Using the degree number, even when it is not mandatory, is considered useful to save typing and avoid clutter, especially when the output set involves joins.

The HAVING sub-clause

The HAVING clause is a filter for the grouped output, corresponding to the way a WHERE clause filters the incoming, ungrouped set.

A HAVING condition uses exactly the same predicating syntax as a WHERE condition but it should not be confused with the WHERE clause. The HAVING filter is applied to the groups after the set has been partitioned. You may still need a WHERE clause to filter the incoming set.

Only groupable items can be used with HAVING.

v.1.0.x In Firebird 1.0.x, you can specify a HAVING condition using columns that are not included in the groupable items—“a lazy man's WHERE clause” and certainly not standards conformant.

It is important to recognise the impact of the HAVING condition on performance. It is processed after grouping is done. If it is used instead of WHERE conditions, to filter out unwanted members returned in groups named in the GROUP BY list, rows are wastefully double-processed, only to be eliminated when it is nearly all done.

For best performance, use WHERE conditions to pre-filter named groups and use HAVING for filtering on the basis of results returned by aggregating expressions. For example, a group total calculated using a SUM(x) expression would be filtered by HAVING SUM(x) > <minimum-value>. A HAVING clause thus typically takes an aggregate expression as its argument.

Taking the previous query, we can use a WHERE clause to filter the project groups that are to appear in the output and a HAVING clause to set the starting range of the totals that we want to view:

```

SELECT

```

```
PROJ_ID
SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
AND PROJ_ID STARTING WITH 'M'
GROUP BY PROJ_ID
HAVING SUM(PROJECTED_BUDGET) >= 100000;
```

PROJ_ID	TOTAL_BUDGET
=====	=====
MAPDB	111000.00
MKTPR	1480000.00

The HAVING clause can take complex ANDed and ORed arguments, using the same precedence logic as the WHERE clause.

The COLLATE sub-clause

If you want a text grouping column to use a different collation sequence from the one defined as default for it, you can include a COLLATE clause. For more information about COLLATE, see the topic *Collation Sequence* in Chapter 9, *Character Types*.

Using ORDER BY in a grouped query

Items listed in the ORDER BY clause of a grouped query must be either aggregate functions that are valid for the grouping context or items that are present in the GROUP BY list.

v.1.0.x Firebird 1.0.x is less restrictive—it will permit ordering on items or expressions that are out of the grouping context.

Advanced grouping conditions

Some advanced grouping conditions are possible, although not in v.1.0.x.

Subqueries with embedded aggregations

A groupable field that is a correlated subquery expression can contain an aggregate expression that refers to an aggregate expression item in the GROUP BY list.

Example In the following example, a re-entrant subquery on the system table RDB\$RELATION_FIELDS contains an aggregate expression (MAX(r.RDB\$FIELD_POSITION) whose result is used to locate the name (RDB\$FIELD_NAME) of the column having the highest field position number for each table (RDB\$RELATION_NAME) in the database:

```
SELECT
  r.RDB$RELATION_NAME,
  MAX(r.RDB$FIELD_POSITION) AS MAXFIELDPOS,
  (SELECT r2.RDB$FIELD_NAME FROM RDB$RELATION_FIELDS r2
   WHERE r2.RDB$RELATION_NAME = r.RDB$RELATION_NAME
```

```

        and r2.RDB$FIELD_POSITION = MAX(r.RDB$FIELD_POSITION)) AS FIELDNAME
FROM RDB$RELATION_FIELDS r
/* we use a WHERE clause to filter out the system tables */
WHERE r.RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
      GROUP BY 1;

```

RDB\$RELATION_NAME	MAXFIELDPOS	FIELDNAME
=====	=====	=====
COUNTRY	1	CURRENCY
CROSS_RATE	3	UPDATE_DATE
CUSTOMER	11	ON_HOLD
DEPARTMENT	6	PHONE_NO
EMPLOYEE	10	FULL_NAME
EMPLOYEE_PROJECT	1	PROJ_ID
JOB	7	LANGUAGE_REQ
PHONE_LIST	5	PHONE_NO
PROJECT	4	PRODUCT
PROJ_DEPT_BUDGET	4	PROJECTED_BUDGET
SALARY_HISTORY	5	NEW_SALARY
SALES	12	AGED

Along the same lines, this time we use COUNT() to aggregate the number of columns stored in each table:

```

SELECT
    rf.RDB$RELATION_NAME AS "Table Name",
    (SELECT r.RDB$RELATION_ID FROM RDB$RELATIONS r
     WHERE r.RDB$RELATION_NAME = rf.RDB$RELATION_NAME) AS ID,
    COUNT(*) AS "Field Count"
FROM RDB$RELATION_FIELDS rf
WHERE rf.RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
      GROUP BY rf.RDB$RELATION_NAME;

```

Table Name	ID	Field Count
=====	=====	=====
COUNTRY	128	2
CROSS_RATE	139	4
CUSTOMER	137	12
DEPARTMENT	130	7
... and so on		

Aggregations with embedded subqueries

An aggregating function expression—COUNT, AVG, etc.—can take an argument that is a subquery expression returning a scalar result. For example, the following query passes the result of a SELECT COUNT(*) query to a higher-level SUM() expression that outputs, for each table (RDB\$RELATION_NAME) the product of the field count and the number of indexes on that table:

```

SELECT
    r.RDB$RELATION_NAME,

```

```
SUM(
  (SELECT COUNT(*) FROM RDB$RELATION_FIELDS rf
    WHERE rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME))
AS "Fields * Indexes"
FROM RDB$RELATIONS r
JOIN RDB$INDICES i
  ON (i.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
WHERE r.RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
GROUP BY r.RDB$RELATION_NAME;
```

RDB\$RELATION_NAME	Fields * Indexes
=====	=====
COUNTRY	2
CROSS_RATE	4
CUSTOMER	48
DEPARTMENT	35
... and so on	

Aggregations at mixed grouping levels

Aggregate functions from different grouping levels can be mixed in the same grouped query.

In the following example, an expression result based on a subquery that does a COUNT() on a column in a lower level group (RDB\$INDICES) is passed as output to the grouping level. The HAVING clause performs filtering predicated on two further aggregations on the lower-level group.

```
SELECT
  r.RDB$RELATION_NAME,
  MAX(i.RDB$STATISTICS) AS "Max1",
  /* one aggregating expression nested within another */
  (SELECT COUNT(*) || ' - ' || MAX(i.RDB$STATISTICS)
    FROM RDB$RELATION_FIELDS rf
    WHERE rf.RDB$RELATION_NAME = r.RDB$RELATION_NAME) AS "Max2"
FROM RDB$RELATIONS r
JOIN RDB$INDICES i
  ON (i.RDB$RELATION_NAME = r.RDB$RELATION_NAME)
WHERE r.RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
GROUP BY r.RDB$RELATION_NAME
HAVING MIN(i.RDB$STATISTICS) <> MAX(i.RDB$STATISTICS);
```

RDB\$RELATION_NAME	Max1	Max2
=====	=====	=====
MTRANSACTION	0000000000000001	18 - 1.00000000000000
MEMBER	0.0135135138407	12 - 0.01351351384073496

v.1.0.x You can get results from running this query in Firebird 1.0.x but they will be incorrect.

Nesting aggregate functions

The previous query also illustrates how an aggregating expression can be nested inside another aggregate expression if the inner aggregate function is from a lower context.

Aggregating Functions

The aggregating functions—all implemented internally—are AVG(), SUM(), MIN(), MAX(), COUNT() and LIST().

AVG()

Returns the average of a range of values in a column of a numeric type or an expression that resolves to a scalar set of a numeric type. The input must be numeric. If any values in the range are NULL, they are excluded from the calculation and the cardinality of the range is reduced accordingly.

If the SELECT expression returns no rows, AVG returns NULL.

Syntax AVG ([ALL] <input-val> | DISTINCT <input-val>)

<input-value> is a column or expression that resolves to a numeric data type

The default ALL, which is optional, returns a result that is the average of all the **<input-value>** members in the set.

The DISTINCT option eliminates duplicates before performing the calculation.

Example

```
SELECT
  JOB_COUNTRY,
  AVG (DISTINCT SALARY) AS AVG_SALARY
FROM EMPLOYEE
GROUP BY 1;
```

SUM()

Returns the total of a range of values in a column of a numeric type or an expression that resolves to a scalar set of a numeric type. The input must be numeric. If any values in the range are NULL, they are excluded from the calculation.

If the SELECT expression returns no rows, SUM returns NULL.

Syntax SUM ([ALL] <input-val> | DISTINCT <input-val>)

<input-value> is a column or expression that resolves to a numeric data type

The default ALL, which is optional, returns a result that is the sum of all the **<input-value>** members in the set.

The DISTINCT option eliminates duplicates before performing the calculation.

Example

```
SELECT
  JOB_COUNTRY,
  SUM (SALARY) AS AVG_SALARY
FROM EMPLOYEE
```

GROUP BY 1;

MAX()

MAX returns the maximum value in the group for the input argument, which is normally a field in the FROM set but can be an expression, including a scalar subquery. If the argument is a string, this is the value that comes last, according to the collation that is active when the function is called. That unfortunate situation can be easily reversed by creating a DESCending index with the appropriate COLLATE attribute on any column that is subject to being queried for its maximum value.

The data type of the result will be the same as that of the resolved input argument. If the group has no eligible rows, or only rows for which the input expression returns NULL, the result is NULL.

Syntax

MAX (expression)



Text BLOB input is not supported in versions prior to v.2.1. Binary BLOBs are not supported.

MIN()

MIN returns the minimum value in the group for the input argument, which is normally a field in the FROM set but can be an expression, including a scalar subquery. If the argument is a string, this is the value that comes first according to the collation that is active when the function is called.

The data type of the result will be the same as that of the resolved input argument. If the group has no eligible rows, or only rows for which the input expression returns NULL, the result is NULL.

Syntax

MIN (expression)



Text BLOB input is not supported in versions prior to v.2.1. Binary BLOBs are not supported.

Using COUNT() as an aggregating function

The much-maligned COUNT() function has its legitimate uses when employed in a grouped query to output summary counts for groups. Consider the following: the column DEPT_NO is not an eligible contender as either a groupable item or a grouping item for our requirements but we can use COUNT to aggregate some information about it in the grouping context of PROJ_ID:

```
SELECT
    PROJ_ID,
    SUM(PROJECTED_BUDGET) AS TOTAL_BUDGET,
    COUNT(DEPT_NO) AS NUM_DEPARTMENTS
FROM PROJ_DEPT_BUDGET
WHERE FISCAL_YEAR = 1994
    GROUP BY PROJ_ID;

PROJ_ID    TOTAL_BUDGET    NUM_DEPARTMENTS
```

=====	=====	=====
GUIDE	650000.00	2
HWRII	520000.00	3
MAPDB	111000.00	3
MKTPR	1480000.00	5
VBASE	2600000.00	3

The LIST() function

The LIST() function takes a singular value expression and returns all the found values, aggregated into a comma-separated list, as a text BLOB. An optional second argument allows a different separator string to be specified.

Syntax The syntax pattern is:

```
LIST ( [ ALL | DISTINCT ] <value-expression> [, <separator-value> ] )
```

The **<value-expression>** argument is not optional. Numeric and date/time values are implicitly converted to strings during evaluation. Text BLOBs are eligible.



*If the **<value-expression>** is a BLOB that is not a TEXT sub-type, the output will be a BLOB of that sub-type, not TEXT.*

By default, all of the values returned by the **<value-expression>** are included in the list. Where the **<value-expression>** is a string type, DISTINCT can be specified to exclude duplicates. DISTINCT does not work if the **<value-expression>** is a BLOB.

The default separator is a comma. That can be overridden by including the optional second argument, **<separator-value>**. It can be a string literal of any character set in all contexts. With v.2.1.4 and higher versions, it can be any string expression, including a prepared parameter that is place-holding a VARCHAR or CHAR type, a local variable in PSQL or it could even be a BLOB if, for example, you wanted to use a glyph as the separator.

The output order of the listed values depends on the order of the set from which the list is extracted, i.e., there is no way to specify an output order explicitly.

Examples The first example extracts a list of customer names from the CUSTOMER table, using a derived table as its **<value-expression>**. The list is displayed in a BLOB viewer.

```
SELECT LIST(CL.CUSTOMER, ';' )
FROM
  (SELECT CUSTOMER FROM CUSTOMER ORDER BY 1) AS CL
```

Figure 22.1 LIST() output snapshot 1:

```
CLIST
3D-Pad Corp.; Anini Vacation Rentals; Buttle, Griffith and Co.; Central Bank; DT System
ns, LTD.; Dallas Technologies; DataServe International; Dynamic Intelligence Corp; Dyn
```

In the next example, we make use of the feature whereby a TEXT BLOB can be cast as a string type, to show how handy this function can be when we exploit its aggregating talents:

```
SELECT
  DEPT_NO,
  CAST(LIST(LAST_NAME) AS VARCHAR(500)) AS ALIST
```

```
FROM EMPLOYEE
GROUP BY 1;
```

A snapshot of the results in a list viewer is shown in Figure 22.2.

Figure 22.2 LIST() output snapshot 2

DEPT NO	ALIST
000	Lee,Bender
100	MacDonald,Yanowski
110	Baldwin,Leung
115	Ichida,Yamamoto
120	Bennet,Reeves,Stansbury
121	Osborne
123	Glon
125	Ferrari
130	Lambert,Weston
140	Sutherland
180	Johnson,Nordstrom
600	Nelson,Brown
621	Young,Ramanathan,Bishop,Green
622	Forest,Burbank,Guckenheimer
623	Young,De Souza,Phong,Parker,Johnson
670	O'Brien,Cook
671	Papadopoulos,Fisher,Page
672	Williams,Montgomery
900	Hall,Steadman

CHAPTER

23

VIEWS AND OTHER RUN-TIME SET OBJECTS

Between the abstract layer of persistent tables and the dynamic DML layer that enables clients and procedural modules to manipulate the abstract into dynamic sets of information lie the various forms of virtual set objects that are standardised in form and function, to a greater or lesser degree.

Forms and Functions

In this chapter, we examine the various forms of virtual set object supported in Firebird, viz.,

View—a persistent definition that is defined by a `SELECT` statement over one or more tables but stores no data. A view is defined in DML and is globally accessible to all users with the appropriate privileges. Views are available in all versions.

Derived table—A `SELECT` expression, enclosed in brackets and named, specifying a set that can be used in `FROM` and `JOIN` clauses as though it were a real table, with restrictions. The set lives as long as the execution of the main query that uses it. Derived tables are supported in all versions from the “2” series onward.

Common table expression (CTE)—a view-like structure that is defined in the preamble of the main query statement that will use it. It could be described as “a derived table on steroids”: it has all the capabilities of a derived table and more. It can be self-referencing and supports up to 1024 levels of recursion for retrieving “flattened” records from tables or views designed for storing hierarchical (“tree”) data. The set lives as long as the execution of the main query that uses it. CTEs are supported from v.2.1 onward.

Other virtual run-time set objects, touched on in this chapter but described in detail elsewhere are

Global temporary table (GTT)—like a view, a GTT is a persistent definition, defined in DDL and stored without data. It is different from a view, in that it is defined not by a `SELECT` statement but by a specialised table definition. An instance of a GTT is

created on demand at run-time and populated by the calling application. The instance is private to the client connection or user transaction that instantiates it and resides in the server resources (memory, disk) isolated for that connection. GTTs are supported from v.2.1 onward.

Selectable stored procedure—a procedural language (PSQL) module designed to generate a run-time set of virtual data that has been retrieved and reprocessed. In all versions of Firebird, the module is called by clients from a SELECT statement, often parameterized to pass input variables to it. From the “2” series onward, a PSQL block can be defined by the client, for one-off usage, in the DML statement EXECUTE BLOCK.

External virtual table (EVT)—an external virtual table (EVT) is a persistent table definition that gets its data from some external data source rather than from the database. The results of a query on an EVT are treated exactly the same way as the results of any other query, and look exactly as if they came from a database table. External tables are supported in all versions.

Views

In SQL-89 and SQL-92 terms, a view is a standard table type, also referred to as *viewed tables* or *virtual tables*. It is characterized as virtual because, instead of storing a table object and allocating pages for storing data, the Firebird engine stores a metadata object description. It comprises a unique identifier, a list of column specifications and a compiled SELECT statement for retrieving the described data into those columns at run time.

What is a view?

At its simplest, a view is a table specification that stores no data. It acts as a filter on both the columns and the rows of the underlying tables referenced in the view—a “window” through which actual data are exposed. The query that defines the view can be from one or more tables or other views in the current database. It behaves in many ways like a persistent table and encapsulates some special extensions to link it with the underlying tables.

By nature, a view is a persistent object and, as such, must exist in the metadata of a database in order to be used. This part of the book is about data manipulation language (DML), of course, and a view is defined by a DML statement—SELECT. Here, then, we are interested in both the data definition language (DDL) for creating, altering and dropping a view and the DML that defines it.

Once a view definition is committed, you query the view as if it were an ordinary table—perform joins, order and group output, specify search conditions, subquery it, derive run-time columns from its virtual data, process a named or unnamed cursor selected from it and so on.

Many views can be “updated”—thereby modifying the state of the underlying persistent tables—or can be made updatable through triggers. When changes to the tables’ data are committed, the data content of the view changes with them. When a view’s data changes are committed, underlying tables’ data change accordingly.

Keys and indexes

Views can not have keys or indexes. The underlying tables—known as *base tables*—will be used as the sources of indexes when the optimizer constructs query plans. The topic of query plans for queries involving views is quite complicated. It is discussed later in this chapter, in the topic [Using query plans for views](#).

Row ordering and grouping

From the “2” series onward, a view definition can include an ordering specification. In prior versions, an exception is thrown if an ORDER BY clause is included. Consequently, it does not make sense in v.1.5.x to use the FIRST and/or SKIP quantifiers for the defining SELECT statement, since they operate on ordered sets.

A grouped query specification (using a legal GROUP BY clause) is fine.

Other SELECT expressions

FIRST/SKIP and ROWS syntaxes, UNIONs and PLAN clauses can be used in view specifications from the “2” series onward, just as they would be used in regular SELECT expressions.

Creating a View

The DDL statement for defining the query specification that will be transformed into a view object is CREATE VIEW. Although it defines a table—albeit a virtual one—and optionally allows custom names to be declared for columns, the syntax does not include any data definitions for the columns. Its structure is formed around the column list of a SELECT statement and the tables specified in the FROM and, optionally, JOIN clauses of the statement. The data types and other attributes of columns are inherited from the tables’ definitions.

All styles of joined and union sets that are supported in queries are supported in view definitions. However, in versions prior to v.2.5, it is not possible to define a view that selects from the output set of a stored procedure or derived table.

The CREATE VIEW statement

The syntax for creating a view is:

```
CREATE VIEW view-name
  [(view-column-name [, view-column-name [...]])]
AS
  <select-specification> [WITH CHECK OPTION];
```

view name—uniquely identifies the view as an object in the database. The name cannot be the same as the name of any other view, table or stored procedure.

Inferred column names

As Firebird has advanced through versions, so has its ability to infer column names from the metadata of the base objects. At v.2.5, the circle is virtually complete, with automatic inference of column names for views over selectable stored procedures and grouped selects.

Specifying view column names

Specifying the list of column names for the view is optional if there are no duplicate names in the column list. The names of the underlying columns of persistent tables will be used by default if there are no duplicate names.

In the case where a join would result in duplicated column names, it becomes mandatory to use a complete list of names and rename columns to avoid the duplication.

This rather ugly example demonstrates how duplication of column names could occur:

```
CREATE VIEW VJOB_LISTING
AS
SELECT
    E.*,
    J.JOB_CODE,
    J.JOB_TITLE
FROM EMPLOYEE E
JOIN JOB J
    ON E.JOB_CODE = J.JOB_CODE ;
```

```
ISC ERROR CODE:335544351
unsuccessful metadata update
STORE RDB$RELATION_FIELDS failed
attempt to store duplicate value (visible to active transactions) in unique index
"RDB$INDEX_15"
```

Index RDB\$INDEX_15 is a unique index on the relation name and the field name. The JOB_CODE column from the EMPLOYEE table was already stored for VJOB_LISTING, hence the exception.

It is necessary to name all of the columns in this view:

```
CREATE VIEW VJOB_LISTING (
    EMP_NO, FIRST_NAME, LAST_NAME,
    PHONE_EXT, HIRE_DATE, DEPT_NO,
    EMP_JOB_CODE, /* alternative name */
    JOB_GRADE, JOB_COUNTRY, SALARY, FULL_NAME,
    JOB_JOB_CODE, /* alternative name */
    JOB_TITLE)
AS
SELECT
    E.*,
    J.JOB_CODE,
    J.JOB_TITLE
FROM EMPLOYEE E
JOIN JOB J
    ON E.JOB_CODE = J.JOB_CODE ;
```

A list is mandatory also if the column list contains any fields created from expressions. For example, this fails:

```
CREATE VIEW VJOB_ALT NAMES
AS
SELECT JOB_CODE || 'for ' || JOB_TITLE AS ALTNAME
```

```
FROM JOB;
```

```
ISC ERROR CODE:335544569
```

```
Invalid command
```

```
must specify column name for view select expression
```

This succeeds:

```
CREATE VIEW VJOB_ALT NAMES
```

```
(ALTNAME)
```

```
AS
```

```
SELECT JOB_CODE || ' for ' || JOB_TITLE
```

```
FROM JOB;
```

The view's column name list specification must correspond in order and number to the columns listed in the `SELECT` statement.

UNIONS

In versions prior to v.2.5, a column list is required when the view set is being defined by a `UNION`. From v.2.5, it is optional—the engine will autodetect the column names from the first `SELECT` expression.

The SELECT specification

The `SELECT` specification is an ordinary `SELECT` statement that can incorporate joins, expression fields, grouping specifications and search conditions—but not ordering conditions in versions prior to the “2” series.

The output list in the `SELECT` clause automatically defines the types, degree and, unless explicitly specified, the names of the views columns.

A `SELECT DISTINCT` query is valid, if required.

The `FROM` clause—along with any `JOIN` clauses or subqueries—defines the base tables of the view.



*`SELECT * FROM <relation>` is valid, although not recommended for views if effective self-documentation is in your landscape. When it is used, the column order will follow that of the base table. It is important to remember that if you need to use the column-naming clause (see *Specifying column names*, above).*

A `WHERE` clause can be included if you want to specify search conditions.

A valid `GROUP BY` clause with an optional `HAVING` clause can be included.

Defining computed columns

The same rules that apply to any expression used for defining a run-time field for a query apply to run-time columns for the view specification. The output is almost parallel to a computed column in a table. However, a computed column has its own, distinct effects in a view:

- it forces the view column-list to become mandatory
- it makes the query non-updatable

Suppose you want to create a view that assigns a hypothetical 10% salary increase to all employees in the company. The next example creates a read-only view that displays all of the employees and their possible new salaries:

```
CREATE VIEW RAISE_BY_10
```

```
(EMPLOYEE, NEW_SALARY)
AS
SELECT EMP_NO, SALARY * 1.1 FROM EMPLOYEE;
```

WITH CHECK OPTION

The WITH CHECK OPTION phrase is an optional syntax item used only in view specifications. It affects updatable views that have been defined with a WHERE clause. Its effect is to block any update operation that would result in a violation of a search condition in the WHERE clause.

Suppose you create a view that allows access to information about all departments with budgets between \$10,000 and \$500,000. The view, V_SUB_DEPT, could be defined as follows:

```
CREATE VIEW V_SUB_DEPT (
    DEPT_NAME,
    DEPT_NO,
    SUB_DEPT_NO,
    LOW_BUDGET)
AS SELECT
    DEPARTMENT,
    DEPT_NO,
    HEAD_DEPT,
    BUDGET
FROM DEPARTMENT
WHERE BUDGET BETWEEN 10000 AND 500000
WITH CHECK OPTION;
```

A user with INSERT privileges on the view can insert new data into the DEPARTMENT, DEPT_NO, HEAD_DEPT AND BUDGET columns of the base table through this view. WITH CHECK OPTION ensures that all budget values entered through the view fall within the range prescribed by the view.

The following statement inserts a new row for the Publications Department through the V_SUB_DEPT view:

```
INSERT INTO V_SUB_DEPT (
    DEPT_NAME,
    DEPT_NO,
    SUB_DEPT_NO,
    LOW_BUDGET)
VALUES ('Publications', '999', '670', 250000);
```

But this statement will fail, because the value of LOW_BUDGET is outside the range prescribed for its target, the underlying column, BUDGET:

```
INSERT INTO V_SUB_DEPT (
    DEPT_NAME,
    DEPT_NO,
    SUB_DEPT_NO,
    LOW_BUDGET)
VALUES ('Publications', '999', '670', 750000);
```

```
ISC ERROR CODE:335544558
```

Operation violates CHECK constraint on view or table V_SUB_DEPT

A view WITH CHECK OPTION clause can be useful when you want to provide users with an updatable view but want to prevent them from updating certain columns. Just include a search condition for each column you want to protect. Its usefulness is limited somewhat by the fact that a view cannot be defined with replaceable parameters.

How views can be useful

The data requirements of an individual user or user group are often quite consistent. Views provide the means to create custom versions of the underlying tables to target clusters of data that are pertinent to specific users and their tasks. To summarize the benefits of views:

- Simplified, re-useable data access paths—views enable you to encapsulate a subset of data from one or more tables to use as a foundation for future queries
- Customized access to the data—views provide a way to tailor the database output so that it is task-oriented, suits the specific skills and requirements of users and reduces the volume of data moving across networks.
- Data independence—views can shield user applications from the effects of changes to database structure. For example, if the DBA decides to split one table into two, a view can be created that joins the two new tables. Applications can continue to query the view as if it were still a single, persistent table.
- Data security—views enable access to sensitive or irrelevant portions of tables to be restricted. For example, a user might be able to look up job information through a view over an Employee table, without seeing associated salary information.

Some simple view specifications

A view can be created from virtually any SELECT query specification. For example:

A vertical subset of columns from a single table

The JOB table in the employee.fdb database has 8 columns: JOB_CODE, JOB_GRADE, JOB_COUNTRY, MIN_SALARY, MAX_SALARY, JOB_REQUIREMENT and LANGUAGE_REQ.

The following view returns a list of salary ranges (subset of columns) for all jobs (all rows) in the JOB table:

```
CREATE VIEW JOB_SALARY_RANGES
AS SELECT
    JOB_CODE,
    MIN_SALARY,
    MAX_SALARY
FROM JOB;
```

A horizontal subset of rows from a single table

The next view returns all of the columns in the JOB table, but only the subset of rows where the MAX_SALARY is less than \$15,000:

```
CREATE VIEW LOW_PAYING_JOBS
AS
```

```
SELECT * FROM JOB
WHERE MAX_SALARY < 15000;
```

A combined vertical and horizontal subset

This view returns only the JOB_CODE and JOB_TITLE columns and only those jobs where MAX_SALARY is less than \$15,000:

```
CREATE VIEW ENTRY_LEVEL_JOBS
AS SELECT
    JOB_CODE,
    JOB_TITLE
FROM JOB
WHERE MAX_SALARY < 15000;
```

A subset of rows and columns from multiple tables

The next example shows a view that joins the JOB and EMPLOYEE tables. EMPLOYEE contains 11 columns: EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, HIRE_DATE, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY, SALARY, FULL_NAME. It returns two columns from the JOB table, and two columns from EMPLOYEE, filtering so that records for workers whose salary is \$15,000 or more are suppressed:

```
CREATE VIEW ENTRY_LEVEL_WORKERS
AS SELECT
    E.JOB_CODE,
    J.JOB_TITLE,
    E.FIRST_NAME,
    E.LAST_NAME
FROM JOB J
JOIN EMPLOYEE E
ON J.JOB_CODE = E.JOB_CODE
WHERE E.SALARY < 15000;
```

Read-only and updatable views

When a DML operation is performed on a view, the changes can be passed through to the underlying tables from which the view was created only if certain conditions are met. If a view meets these conditions, it is updatable. If it does not meet these conditions, it is read-only, and writes to the view cannot be passed through to the underlying tables.

Values can only be inserted through a view for those columns named in the view. Firebird stores NULL into any unreferenced columns. A view will not be updatable if any non-nullable base columns are absent from the view.

A read-only view can be made updatable by means of *triggers*.

Read-only views

A view will be read-only if its SELECT statement has **any** of the following characteristics:

- specifies a row quantifier other than ALL (i.e. DISTINCT, FIRST, SKIP)
- contains fields defined by subqueries other expressions

- contains fields defined by aggregating functions and/or a GROUP BY clause
- includes UNION specifications
- joins multiple tables
- does not include all NOT NULL columns from the base table
- selects from an existing view that is not updatable
- selects from a stored procedure

Making read-only views updatable

Many non-updatable views can be made into updatable views using triggers that will perform the correct writes to the base tables when a DELETE, UPDATE or INSERT operation is requested for a view.

The following script creates two tables, creates a view that is a join of the two tables, and then creates three triggers—one each for DELETE, UPDATE and INSERT—that will pass all updates on the view through to the base tables.

```
CREATE TABLE Table1 (
    ColA INTEGER NOT NULL,
    ColB VARCHAR(20),
    CONSTRAINT pk_table PRIMARY KEY(ColA)
);
COMMIT;
--
CREATE TABLE Table2 (
    ColA INTEGER NOT NULL,
    ColC VARCHAR(20),
    CONSTRAINT fk_table2 FOREIGN KEY REFERENCES Table1(ColA)
);
COMMIT;
--
CREATE VIEW TableView AS SELECT
    Table1.ColA,
    Table1.ColB,
    Table2.ColC
FROM Table1, Table2
    WHERE Table1.ColA = Table2.ColA;
COMMIT;
--
SET TERM ^;
CREATE TRIGGER TableView_Delete FOR TableView
ACTIVE BEFORE DELETE AS
BEGIN
    DELETE FROM Table1
        WHERE ColA = OLD.ColA;
    DELETE FROM Table2
        WHERE ColA = OLD.ColA;
END ^
--
```

```

CREATE TRIGGER TableView_Update FOR TableView
ACTIVE BEFORE UPDATE AS
BEGIN
    UPDATE Table1
        SET ColB = NEW.ColB
        WHERE ColA = OLD.ColA;
    UPDATE Table2
        SET ColC = NEW.ColC
        WHERE ColA = OLD.ColA;
END ^
--
CREATE TRIGGER TableView_Insert FOR TableView
ACTIVE BEFORE INSERT AS
BEGIN
    INSERT INTO Table1 values (NEW.ColA,NEW.ColB);
    INSERT INTO Table2 values (NEW.ColA,NEW.ColC);
END ^
COMMIT ^
SET TERM ;^

```



Often, you will be able to create one conditional trigger for each phase, using the “multi-action” syntax, e.g.,

```

CREATE TRIGGER ...
ACTIVE BEFORE | AFTER INSERT OR UPDATE OR DELETE
AS...

```

For details, see [Multi-action triggers](#) in Chapter 30.

When defining triggers for views, take care to ensure that any triggers on the view do not create a conflict or an unexpected condition with regard to triggers defined for the base tables. The trigger event for the view precedes that for the table, respective of phase (BEFORE/AFTER).

For example, if you have a BEFORE INSERT trigger on the base table that fetches a fresh generator value for the primary key if its new.value is null and the view trigger includes an INSERT statement, omit the primary key from the view trigger in order to pass NULL to the new.value of the primary keys and allow the table trigger to do its work.

Gotcha!

An implementation bug that is present until v.2.0 means that, for a view that is non-updatable because it is missing base columns that are non-nullable, attempts to insert will fail, even with an INSERT trigger that looks as though it should work. It happens because, in the older versions, the NOT NULL constraint checking is happening before the trigger has a chance to correct it.

For the full information about writing triggers, refer to Chapter 30.

Hopeless cases

Not all views can be made updatable by defining triggers for them. For example, this handy little read-only view reads a context variable from the server; but regardless of the triggers you define for it, all operations except SELECT will fail:

```

CREATE VIEW SYSTRANS
(CURR_TRANSACTION) AS
    SELECT CURRENT_TRANSACTION FROM RDB$DATABASE;

```

Naturally updatable views

A view is naturally updatable if both of the following conditions are met:

- the view specification is a subset of a single table or another updatable view.
- all base table columns excluded from the view definition are nullable

The following statement creates a naturally updatable view:

```
CREATE VIEW EMP_MNGRS (FIRST, LAST, SALARY) AS SELECT
    FIRST_NAME,
    LAST_NAME,
    SALARY
FROM EMPLOYEE
WHERE JOB_CODE = 'Mngr'
WITH CHECK OPTION;
```

Because the WITH CHECK OPTION clause was included in this specification, applications will be prevented from changing the JOB_CODE—even if there is no violation of the foreign key constraint on this column in the base table.

Changing the behavior of updatable views

Alternative behavior for naturally updatable views can be specified using triggers. For a particular phase (BEFORE/AFTER) of an operation, the view triggers fire before the base table's triggers. With care and skill, it is thus possible to use views to pre-empt the normal trigger behavior of the base table under planned circumstances.

It is also possible to create havoc with badly-planned view triggers. Test, test, test!

Modifying a view

The terms *updatable* and *read-only* refer to how the *data* in the base tables can be accessed, not to whether the view definition can be modified. Prior to v.2.5, no syntax is available to modify a view definition. RECREATE VIEW is available, to attempt to drop the existing view if it exists already, before creating a new one of the same name, behaving exactly like CREATE VIEW. If the existing view is in use or has dependencies, RECREATE VIEW will fail.

v.1.0.x RECREATE VIEW is not supported in v.1.0.x. It is necessary to drop any dependencies, drop the view and then create a new one using CREATE VIEW.

From v.2.5 onward, you can use ALTER VIEW OR CREATE OR ALTER VIEW to modify a view definition. Both follow the same syntax as CREATE VIEW.

- ALTER VIEW enables a view definition, with all of its dependencies, to be altered in one step. There is no need to drop the old version and create the new one from scratch.
- Consistently with CREATE OR ALTER TABLE, the CREATE OR ALTER VIEW statement allows the view definition will be altered (as with ALTER VIEW) if it exists, or created if it does not exist.

Syntax create [or alter] | alter } view <view_name> [(<field list>)]
as <select statement>

Example CREATE TABLE USERS (
ID INTEGER,
NAME VARCHAR(20),

```
PASSWD VARCHAR(20);
COMMIT;
CREATE VIEW V_USERS AS
    SELECT NAME FROM USERS;
COMMIT;
ALTER VIEW V_USERS (ID, NAME) AS
    SELECT ID, NAME FROM USERS;
COMMIT;
```

Dropping a view

The DROP VIEW statement enables a view’s owner to remove its definition from the database. It does not affect the base tables associated with the view.

Syntax `DROP VIEW view-name;`

The DROP VIEW operation will fail if you are not logged in as the owner of the view or if the view is used by another object, such as a another view, a stored procedure, a trigger on another view or a table or a CHECK constraint definition.

The following statement removes a view definition:

```
DROP VIEW SUB_DEPT;
```

Privileges for Views

Because a view is a database object, it requires specific user privileges in order to be accessed. Through granting privileges to a view, it is possible to provide users with very finely-grained access to certain columns and rows from tables, whilst denying them access to other, more sensitive data stored in the underlying tables. In that case, the view is granted privileges to the tables and the users are granted privileges to the views.

Owner privileges

The user that creates the view will be its owner. In order to create the view, the user must have the appropriate privileges on the base tables:

- Some views are read-only by nature (see below). To create a read-only view, the creator needs SELECT privileges for any underlying tables.
- For views that are updatable, the creator needs ALL privileges to the underlying tables.

Additionally, the owners of the base tables and other views accessed by the view must grant all required privileges for accessing those objects, and for modifying them through the view if required, to the view itself. That is, privileges ON those base objects must be granted TO the view.

The owner of the view has all privileges for it, including the ability to grant privileges on it to other users, to triggers, and to stored procedures.

User privileges

The creator of the view must grant the appropriate privileges to users, stored procedures and other views that need to access the view. A user can be granted privileges to a view without having access to its base tables.

In the case of updatable views, INSERT, UPDATE and DELETE privileges must be assigned to any users who need to perform DML on underlying tables through the view. Conversely, grant the users only SELECT privileges if your intention is to provide a read-only view.

If a user already has the required rights on the view's base objects, it will automatically have the same rights on the view.



The less direct a user's privileges are, the more secure the base objects are. However, the potential multiplicity of the hierarchies of privileges can cause problems if the chain gets broken by revoking privileges from the view's owner. Considering the attractions of views as a mechanism for protecting data from being seen by the wrong eyes, it behoves the system administrator to maintain reliable documentation of all privileges granted.

For a detailed description of SQL privileges, refer to Chapter 37, **Database-level Security**.

Using views in SQL

In SQL a view behaves in many ways just like a regular table. You can select from it, with or without ORDER BY, GROUP BY and WHERE clauses.

If it is naturally updatable, or has been made updatable by triggers, and the appropriate SQL privileges exist, you can perform both positioned and searched update, insert and delete operations on it, which will operate on the underlying table.

You can create views of views.

You can process the output of a selection on a view in PSQL modules.

You can perform a JOIN between a view and other views and tables. In some cases, you can join views with selectable stored procedures.

For a simple illustration, we'll create a view and a stored procedure on the Employee table and join them.

The view:

```
CREATE VIEW V_EMP_NAMES
AS SELECT
  EMP_NO,
  LAST_NAME,
  FIRST_NAME
FROM EMPLOYEE ^
COMMIT ^
```

The procedure:

```
CREATE PROCEDURE P_EMP_NAMES
RETURNS (
  EMP_NO SMALLINT;
  EMP_NAME VARCHAR(35))
AS
BEGIN
  FOR SELECT EMP_NO, FIRST_NAME || ' ' || LAST_NAME
  FROM EMPLOYEE
  INTO :EMP_NO, :EMP_NAME
  DO
    SUSPEND;
```

```
END ^
COMMIT ^
```

A query that joins them:

```
SELECT
  V.EMP_NO,
  V.LAST_NAME,
  V.FIRST_NAME,
  P.EMP_NAME
FROM V_EMP_NAMES V
JOIN P_EMPNAMES P
  ON V.EMP_NO = P.EMPNO ^
```

v.1.0.x Firebird 1.0.x has a known issue with a view that is defined from a UNION. It will misbehave badly if such a view is used in a subquery. For example, the following query will crash the server:

```
SELECT 1 FROM Table1
WHERE EXISTS (
  SELECT FIELD1 FROM UNION_VIEW
  WHERE <search-conditions> )
```

Using query plans for views

Views may present some difficulty for users of the PLAN feature. Ordinarily, users may treat a view the same as a table. However, if you want to define a custom plan, you need to be aware of the indexes and structures of the base table(s) participating in the view.

The optimizer treats a view reference as if the base tables used in creating the view were inserted into the FROM list of the query.

Suppose a view is created as:

```
CREATE VIEW V_PROJ_LEADERS (
  PROJ_ID,
  PROJ_TITLE,
  LEADER_ID,
  LEADER_NAME)
AS SELECT
  P.PROJ_ID,
  P.PROJ_NAME,
  P.TEAM_LEADER,
  E.FULL_NAME,
FROM PROJECT P
JOIN EMPLOYEE E
  ON P.TEAM_LEADER = E.EMPNO;
```

A simple query on the view

```
SELECT * FROM V_PROJ_LEADERS;
```

outputs this plan:

```
PLAN JOIN (V_PROJ_LEADERS P NATURAL,V_PROJ_LEADERS E INDEX (RDB$PRIMARY7))
```

Notice that the optimizer accesses the indexes of the base tables (through the aliases P and E) to evaluate the best way to retrieve the view. It is the SELECT specification of the CREATE VIEW declaration that determines the logic for executing the join.

The next query is a little more complex. This time, the view is joined to the table EMPLOYEE_PROJECT, an intersection of the primary keys of the EMPLOYEE and PROJECT tables. From there, it is joined back into the EMPLOYEE table, to provide a denormalized listing that includes the names of the members of all of the projects commanded by the view:

```
SELECT
  PL.*,
  EMP.LAST_NAME
FROM V_PROJ_LEADERS PL
JOIN EMPLOYEE_PROJECT EP
  ON PL.PROJ_ID = EP.PROJ_ID
JOIN EMPLOYEE EMP
  ON EP.EMP_NO = EMP.EMP_NO;
```

```
PLAN JOIN (EMP NATURAL,EP INDEX (RDB$FOREIGN15),PL P INDEX (RDB$PRIMARY12),PL E INDEX
(RDB$PRIMARY7))
```

This time, the foreign key index on the EMPLOYEE_PROJECT (aliased as EP) column EMP_NO is used to select the project members' names from the second “hit” on EMPLOYEE. As before, the join inside the view uses the primary key of the EMPLOYEE to search for the TEAM_LEADER matches.

If you decide you need to write a custom plan for a query that works on a view, familiarity with the view definition is essential for making your own estimations of indexes and access methods.

For more information about plans, see [Query Plans and the Optimizer](#) in Chapter 19.

Derived Tables

From the “2” series onward, DSQL supports derived tables as defined by SQL200n standards. A derived table is a form of “virtual table” that is returned to a FROM clause as a set derived from a dynamic subquery expression. Derived tables can be nested, if required, to build complex queries and they can be involved in joins as though they were normal tables or views.

Syntax SELECT
 <select list>
FROM
 (select-expression)
 [[AS] derived-table-alias [(column-alias [, column-alias ...])]

Virtually anything that is part of a SELECT expression can be used in the **select-expression**, viz., it can

- contain aggregate functions, subqueries and joins, and can itself be used in aggregate functions, subqueries and joins. However, if the derived table is involved in an inner join and contains a subquery, then no join order can be made.

- have WHERE, ORDER BY and GROUP BY, FIRST, SKIP or ROWS clauses and more.
- be a call to a selectable stored procedure
- be a union. It can also be used in a union

Rules for Derived Tables

- Every column in the derived table must have a name, even those that are not referred to outside that derived table.
- Expression fields and constants can be defined AS <alias> in the normal way or left unnamed and included in the complete column list.
- The list of column aliases is usually optional. If it is used:
 - it must contain aliases for all of the columns
 - the number of columns in the column list should be the same as the number of columns from the query expression
 - the aliases must be in the same left-to-right order as the corresponding columns in the select-expression.
- Column names must be unique. If the query would result in duplicate names, the duplicates must be aliased and the column list must be used.

Examples Simple derived table:

```
SELECT * FROM
  (SELECT
    RDB$RELATION_NAME,
    RDB$RELATION_ID
    FROM RDB$RELATIONS) AS R (RELATION_NAME, RELATION_ID)
```

Nesting and joining: aggregate on a derived table which also contains an aggregate:

```
SELECT
  DT.FIELDS,
  Count(*)
FROM
  (SELECT
    R.RDB$RELATION_NAME,
    Count(*)
    FROM RDB$RELATIONS R
    JOIN RDB$RELATION_FIELDS RF
      ON (RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME)
    GROUP BY R.RDB$RELATION_NAME) AS DT (RELATION_NAME, FIELDS)
  GROUP BY DT.FIELDS
```

UNION and ORDER BY

```
SELECT DT.* FROM
  (SELECT
    R.RDB$RELATION_NAME,
    R.RDB$RELATION_ID
    FROM RDB$RELATIONS R
```



```

UNION ALL
SELECT
    R.RDB$OWNER_NAME,
    R.RDB$RELATION_ID
FROM RDB$RELATIONS R
ORDER BY 2) AS DT
WHERE DT.RDB$RELATION_ID <= 4

```

When to Use a Derived Table

Derived tables may come in handy to simplify a more complex query or to avoid the need to write a stored procedure to obtain the logic that may elude you in DSQL because of language rules. The optimizer can handle a derived table very efficiently.

The following example illustrates how two derived tables extracted from the same SALES table solve the problem of getting two incompatible groupings from the same data. The SALES table contains one row for each sale made by a sales representative. We want the ranked total of sales for the top 5 sales reps but we also want to know the number of distinct customers each one dealt with to obtain these totals.

```

SELECT
    DT1.SALES_REP,
    E.LAST_NAME || ', ' || E.FIRST_NAME AS FULL_NAME,
    DT1.TOTAL_SALES,
    DT2.CUSTOMERS
FROM (SELECT
    S1.SALES_REP,
    SUM(S1.TOTAL_VALUE)
    FROM SALES S1
    GROUP BY 1) AS DT1 (SALES_REP, TOTAL_SALES)
JOIN EMPLOYEE E
    ON (E.EMP_NO = DT1.SALES_REP)
JOIN (SELECT
    S2.SALES_REP,
    COUNT (DISTINCT S2.CUST_NO)
    FROM SALES S2 GROUP BY 1) AS DT2 (SALES_REP, CUSTOMERS)
    ON (DT2.SALES_REP = DT1.SALES_REP)
ORDER BY 3 DESC ROWS 5

```

Common Table Expressions

Like a derived table, a common table expression (CTE) defines a transient set that can be used anywhere within the scope of its defining query as though it were a table or view. CTE support is available in versions from 2.1 onward.

A CTE is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. It also resembles a view, in several ways. The rules and structure of its definition in the preamble to the main query are very similar to the

CREATE VIEW syntax. Like a view, it can be self-referencing and can be referenced more than once in the same query, using multiple aliases.

A CTE can spawn UNION sets that are instances of itself, providing a means to operate recursively on the set it defines. However, it cannot embed another CTE.

Often, a CTE lets you compose awkward queries more directly and lucidly than you might otherwise using multiple, re-entrant outer joins or subqueries. When grouping is involved, a CTE, like a derived table, enables you to extract incompatible aggregations, something that is otherwise not possible unless you write a stored procedure for it.

In extreme cases where recursive retrieval is required, you might still need a stored procedure. For example, although you can define multiple CTEs together for one main statement, it is not possible to nest one CTE within another. For sets that are extracting and summarising at many levels you will need the nesting logic of PSQL looping. However, where the retrieval can be achieved with a CTE, the consumption of memory and CPU cycles is significantly reduced.



You can use a CTE in PSQL within any level of FOR...DO looping.

Syntax for a CTE

The basic syntax pattern is:

```
WITH [RECURSIVE]
    <cte-name1> [ ( <column-list1> ) ] AS <select-expr1>
    [, <cte-name2> [ ( <column-list2> ) ] AS <select-expr2>
    [... , <cte-nameN> [ ( <column-listN> ) ] AS <select-exprN> ] ]
SELECT <main-query> | other DML operations
```

WITH is the keyword that marks the preamble to the definition of one or more CTEs. A single WITH clause may define more than one common table expression.

RECURSIVE is the optional keyword that signals to the engine that the upcoming **<main-query>** is going to be a recursive operation.



*It's perhaps stating the obvious to note that, if **RECURSIVE** is not present, the processing won't be recursive!*

<cte-name1>, **<cte-name2>** and any others are the distinct names for each CTE being defined. CTE definitions must be separated by commas.

<column-list1>, **<column-list2>** and any others are the distinct names for the list of named columns for each CTE being defined. The column-lists are optional except when the engine is unable to infer a distinct column name for each column from the set specified by the **<select-expr>**. If any column names are likely to be duplicated or are not able to be inferred, the column-list must be used, with all specified columns named in specification order and any duplications resolved by renaming.

AS <select-expr1>, **AS <select-expr2>** and any others are the SELECT expressions that define the contents of the respective **<cte-name>** elements. The keyword AS is required. It can be any valid SELECT expression except another CTE (WITH .. SELECT .. construct).

<main-query> is the main SELECT statement. It can refer to any of the CTEs defined in the preamble, including re-using them with distinct alias names. Care is needed to avoid looping. Test!

The main query is not limited to SELECT statements. It can also be an update, merge, insert or delete statement, wherever it is valid to use a SELECT expression. It can even be a subquery expression. For these purposes, the CTE preamble construct must be entirely enclosed in parentheses.

A non-recursive CTE

Here we use a CTE as a means to “pivot” the department budget rows so that the budget figures for each year, by department, appear as columns in the output. In this example, we take the figures for just two of the years in the records, making re-entrant left joins to instances of the CTE for each column.

```
Example1 WITH DEPT_BUDGET_BY_YEAR (FISCAL_YEAR, DEPT_NO, BUDGET) AS
  (SELECT
    FISCAL_YEAR,
    DEPT_NO,
    SUM(PROJECTED_BUDGET)
  FROM PROJ_DEPT_BUDGET
  GROUP BY 1, 2)
--
SELECT
  D.DEPT_NO,
  D.DEPARTMENT,
  DBBY1994.BUDGET AS BUDGET_94,
  DBBY1995.BUDGET AS BUDGET_95
FROM DEPARTMENT D
LEFT JOIN DEPT_BUDGET_BY_YEAR DBBY1994
  ON D.DEPT_NO = DBBY1994.DEPT_NO
  AND DBBY1994.FISCAL_YEAR = 1994
LEFT JOIN DEPT_BUDGET_BY_YEAR DBBY1995
  ON D.DEPT_NO = DBBY1995.DEPT_NO
  AND DBBY1995.FISCAL_YEAR = 1995
WHERE EXISTS (
  SELECT B.* FROM PROJ_DEPT_BUDGET B
  WHERE D.DEPT_NO = B.DEPT_NO);
```

DEPT_NO	DEPARTMENT	BUDGET_94	BUDGET_95
=====	=====	=====	=====
000	Corporate Headquarters	100000.00	<null>
100	Sales and Marketing	1500000.00	3500000.00
621	Software Development	2340000.00	900000.00
DEPT_NO	DEPARTMENT	BUDGET_94	BUDGET_95
=====	=====	=====	=====
622	Quality Assurance	560000.00	<null>
623	Customer Support	80000.00	1200000.00

1. Courtesy of Paul Vinkenooog, adapted from an example in *Firebird 2.5 Language Reference Update*

670	Consumer Electronics Div.	20000.00	<null>
671	Research and Development	461000.00	<null>
672	Customer Services	100000.00	800000.00
110	Pacific Rim Headquarters	200000.00	1200000.00

A recursive CTE

A recursive CTE is a UNION of self-referencing sets, of which at least one must be non-recursive. The non-recursive sets precede the recursive ones and one of them is used as the “anchor” for the recursing sets.

The CTE preamble must begin with the keywords WITH RECURSIVE.

The recursive sets are linked to each other, and the first of those to its non-recursive predecessor, by the UNION ALL operator. The unions between nonrecursive members can be either UNION ALL or UNION DISTINCT.

A recursive CTE follows this execution flow:

The engine begins execution from a non-recursive member.

For each row evaluated, it starts executing each recursive set in turn, using the current values from the outer row as parameters.

If the currently executing instance of a recursing member produces no rows, execution loops back one level and gets the next row from the outer set.

Restrictions on recursing sets

Maximum depth of recursion is 1024 levels. The logic of recursion imposes some restrictions on recursing sets, that don’t apply to non-recursing ones:

- Each recursive union member may reference itself only once and only in a FROM clause.
- Aggregating operations (DISTINCT, GROUP BY, HAVING) and aggregate functions (SUM, COUNT, MAX, MIN, LIST) are not allowed.
- A recursive reference cannot participate in an outer join.

Example²

This time, using similar sets as in the non-recursive example, we gather up the parent departments for the outer (non-recursing) set and course through the children of each department to pick up the children and assemble a pivoted result showing the two budget years in “flattened” tree output that is friendly to “tree” GUI control processors:

```
WITH RECURSIVE
  DEPT_BUDGET_BY_YEAR (FISCAL_YEAR, DEPT_NO, BUDGET) AS
  (SELECT
    FISCAL_YEAR,
    DEPT_NO,
    SUM(PROJECTED_BUDGET)
  FROM PROJ_DEPT_BUDGET
  GROUP BY 1, 2),
  --
  -- the parent (controlling) set
  DEPARTMENT_TREE (DEPT_NO, HEAD_DEPT, DEPARTMENT, INDENT) AS
  (SELECT
```

2. Courtesy of Paul Vinkenoog, adapted from an example in *Firebird 2.5 Language Reference Update*

```

        DEPT_NO,
        HEAD_DEPT,
        DEPARTMENT,
        CAST ('' AS VARCHAR(255))
FROM DEPARTMENT
WHERE HEAD_DEPT IS NULL
--
UNION ALL
-- the recursive child sets
SELECT D.DEPT_NO,
       D.HEAD_DEPT,
       D.DEPARTMENT,
       TREE.INDENT || ' '
FROM DEPARTMENT D
JOIN DEPARTMENT_TREE TREE
  ON D.HEAD_DEPT = TREE.DEPT_NO
)
--
SELECT
  DTREE.DEPT_NO,
  DTREE.DEPARTMENT,
  DBBY1994.BUDGET AS BUDGET_94,
  DBBY1995.BUDGET AS BUDGET_95
FROM DEPARTMENT_TREE DTREE
LEFT JOIN DEPT_BUDGET_BY_YEAR DBBY1994
  ON DTREE.DEPT_NO = DBBY1994.DEPT_NO
 AND DBBY1994.FISCAL_YEAR = 1994
LEFT JOIN DEPT_BUDGET_BY_YEAR DBBY1995
  ON DTREE.DEPT_NO = DBBY1995.DEPT_NO
 AND DBBY1995.FISCAL_YEAR = 1995;

```

DEPT_NO	DEPARTMENT	BUDGET_94	BUDGET_95
000	Corporate Headquarters	100000.00	<null>
100	Sales and Marketing	1500000.00	3500000.00
180	Marketing	<null>	<null>
130	Field Office: East Coast	<null>	<null>
140	Field Office: Canada	<null>	<null>
110	Pacific Rim Headquarters	200000.00	1200000.00
115	Field Office: Japan	<null>	<null>
116	Field Office: Singapore	<null>	<null>
120	European Headquarters	<null>	<null>
121	Field Office: Switzerland	<null>	<null>
123	Field Office: France	<null>	<null>
125	Field Office: Italy	<null>	<null>
600	Engineering	<null>	<null>
620	Software Products Div.	<null>	<null>

621	Software Development	2340000.00	900000.00
622	Quality Assurance	560000.00	<null>
623	Customer Support	80000.00	1200000.00
670	Consumer Electronics Div.	20000.00	<null>
671	Research and Development	461000.00	<null>
672	Customer Services	100000.00	800000.00
DEPT_NO	DEPARTMENT	BUDGET_94	BUDGET_95
=====			
900	Finance	<null>	<null>

Other Virtual Set Objects

Firebird currently supports three other forms of derived set object: the *global temporary table*, the *selectable stored procedure* and the *external virtual table* (EVT).

Global temporary tables (GTTs)

The same structural features that you can apply to regular tables (indexes, triggers, field-level and table level constraints) can be applied to a global temporary table. However, a GTT cannot be linked in a referential relationship to a regular table nor to another GTT that has a different lifespan (see below), nor be referred to in a constraint or domain definition.

For more information, see [Global Temporary Tables](#) in Chapter 15, *Tables*.

Selectable stored procedures

Firebird's PSQL extensions provide syntax for defining a stored procedure that outputs a derived set of data from virtually anywhere: from the database, from context variables, even from input variables alone, from external tables or from any combination. SQL and DSQL SELECT syntax provides for these virtual tables to be retrieved just as though they were real tables.

The output set for a selectable stored procedure is defined as a set of output variables, using the RETURNS clause of a CREATE PROCEDURE statement. The output data are created by looping through a cursor set, defined by a SELECT statement, and reading the values of the specified columns into these output variables or into declared local variables. Within the loop, almost anything can be done to manipulate the data, including processing embedded loops. A selectable stored procedure can be called from (embedded in) another stored procedure and return values to its caller via the RETURNING_VALUES structure. Anything that can be selected, calculated or derived can be transformed to output.

As a simple illustration, the following stored procedure declaration sets up a loop and proceeds to pass processed rows, one at a time, to the output buffer:

```
CREATE PROCEDURE SHOW_JOBS_FOR_COUNTRY (  
    COUNTRY VARCHAR(15))  
RETURNS ( /* the virtual table */  
    CODE VARCHAR(11),
```

```

TITLE VARCHAR(25),
GRADE SMALLINT)
AS
BEGIN
FOR
  SELECT
    job_code,
    job_title,
    job_grade
  FROM job
  WHERE JOB_COUNTRY = :COUNTRY
    INTO :CODE, :TITLE, :GRADE
  DO BEGIN /* begin the loop */
    CODE = 'CODE: ' || CODE; /* mess about with the value a little */
    SUSPEND; /* this outputs one row per loop */
  END
END

```

When the stored procedure is compiled, it is ready for action. Retrieval of the set is by way of a slightly specialized SELECT statement which can, if required, take constant arguments as input parameters:

```
SELECT * FROM SHOW_JOBS_FOR_COUNTRY ('England');
```

CODE	TITLE	GRADE
=====	=====	=====
CODE: Admin	Administrative Assistant	5
CODE: Eng	Engineer	4
CODE: Sales	Sales Co-ordinator	3
CODE: SRep	Sales Representative	4

From the “2” series onward, a PSQL block can be defined by the client application, for one-off usage, in the DML statement EXECUTE BLOCK.

For details about creating and using stored procedures and EXECUTE BLOCK, refer to Part Six. [*Selectable Stored Procedures*](#) are discussed in detail in Chapter 29.

External virtual tables

An *external virtual table* (EVT) is a table that gets its data from some external data source rather than from the database. The results of a query on an EVT are treated exactly the same way as the results of any other query, and look exactly as if they came from a database table. This allows the integration of external data such as real-time data feeds, formatted data in operating system files, other databases (even non-SQL databases), and any other tabular data sources.

Firebird implements EVT's by means of the EXTERNAL FILE clause of the CREATE TABLE statement. External data are read from fixed format text records as though they were regular Firebird data columns.

Firebird external tables can also insert records into EVT's and delete records from them.



The “external virtual table” is a concept that is implemented in a multitude of ways in different database architectures. You would rarely—if ever—hear everyday Firebird developer refer to an external table as “an EVT”.

For more information about external tables, refer to the topic *Using External Files as Tables* in Chapter 15, **Tables**.

CHAPTER 24

INTERACTIVE SQL UTILITY (ISQL)

The *isql* utility, installed in the `/bin` directory beneath your Firebird root, provides a non-graphical interface to Firebird databases that is consistent on all server and client platforms.

isql accepts both DDL and DML statements, as well as a subset of SQL-like console commands not available in DSQL. It can be used both for creating and maintaining metadata and for querying and changing data. It includes several admin tools and the option to perform some database operations directly from a command shell or through a shell script or batch file.



Several other database management systems have adopted the “isql” name for their interactive query programs. Always run Firebird's isql program from its own directory or provide the absolute file path if this is a problem on your server.

Interactive Mode

Interactive SQL (*isql*) is a command-line program, available on all platforms, that can be run locally or from a remote client.

- From a remote client, a valid user name and password are always required to run *isql*.
- If you are connecting locally, you can set the operating system variables `ISC_USER` and `ISC_PASSWORD` and avoid the need to enter them on commands. For more information about these variables, refer to [ISC_USER and ISC_PASSWORD](#) in Chapter 33, **Configuring Firebird and Its Environment**.

Some additional command-line switches can be used when invoking the interactive shell. They are noted later in this chapter, in Table 24.1, [Switches for isql command-line options](#).

Default text editor

Some *isql* commands access your system's default text editor.

- On UNIX, Linux and some other POSIX platforms, the default editor is defined by one or the other of the two environment variables, EDITOR and VISUAL. The installation default is usually *vi*, *vim*, or *emacs* but you can set it to another preferred console (not X) text editor.
- On Windows, the default editor is *notepad.exe*. If you have another text editor that you prefer, you can define a system variable EDITOR for Firebird's use, that *isql* will recognise and try to use instead of the Windows default.



Use an editor that creates only plain text when invoked. Word processors and editors that open new files by default to anything but plain text are not suitable.

Starting isql

To start *isql*, open a command shell and cd to the /bin directory of your Firebird server or client installation. Use the following command pattern at the shell prompt and press the Enter key:

```
isql [<database_name>] [-u[ser] <user-name> -pas[sword] <password>]
```

- <database_name> is optional. If you include it, *isql* will open a connection to the database and start its shell already connected.



Use either the alias of the database or its absolute path. Include the hostname if you are invoking isql remotely, or locally using Superserver on POSIX.

- the switches `-user <user-name>` and `-password <password>` are optional when you are starting *isql* without a connection to the database and required if you are starting *isql* remotely. If the ISC_USER and ISC_PASSWORD environment variables are not set, they will also be required when you start *isql* locally.

Examples

On POSIX:

```
./isql
```

On Windows:

```
isql
```

starts the program.

```
./isql -user TEMPDBA -password osoweary [on POSIX]
```

```
isql -user TEMPDBA -password osoweary [on Windows]
```

starts the program and stores the supplied username and password without authenticating them.

```
isql hotchicken:/data/mydatabase.fdb -user TEMPDBA -password osoweary
```

starts the program on a Windows client and connects to the database on a POSIX server, provided the username and password are valid on the server.

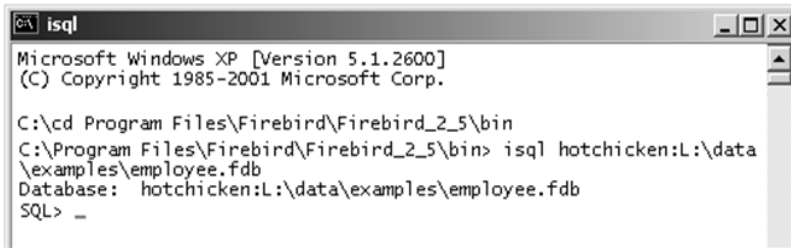
```
./isql /data/mydatabase.fdb
```

starts the program locally on a POSIX Classic server and connects to the database, provided the environment variables ISC_USER and ISC_PASSWORD are set and are available to your Linux user profile.

You are in the *isql* shell if you can see the **SQL>** prompt, . If there were errors in the command string, or authentication problems, you might also see some error messages or you might be still at the command line.

If you are logged in to the database when *isql* starts up, you will see a console display similar to Figure 24.1. The appearance of the surrounding shell depends on the operating system. The *isql* shell is the same on all platforms:

Figure 24.1 Console display when *isql* starts logged-in



If you didn't enter a database path or alias or you used a username and password that are not defined on the server, you will see something similar to Figure 24.2:

Figure 24.2 Console display when *isql* starts not logged-in



Connecting to a database

To connect to a database from the **SQL>** prompt in the *isql* shell, use the following examples as syntax patterns. Notice that the syntax and punctuation inside the *isql* shell are different to what are used when passing the connection parameters from the system shell:

```
CONNECT 'HOTCHICKEN:L:\DATA\EXAMPLES\EMPLOYEE.FDB'
USER 'SYSDBA' PASSWORD 'masterkey';
```

connects to a remote or local server named **HOTCHICKEN**.

```
CONNECT 'L:\DATA\EXAMPLES\EMPLOYEE.FDB';
```

connects to a local server where *isql* already knows your Firebird username and password—either because you entered them correctly when you started *isql* or because *isql* is executing in a shell that can see the environment variables **ISC_USER** and **ISC_PASSWORD**.

```
CONNECT 'HOTCHICKEN:EMP3' USER 'SYSDBA' PASSWORD 'masterkey';
```

is equivalent to the first example, using an alias stored in **aliases.conf** on the server, that points to the path.

```
CONNECT 'L:/DATA/EXAMPLES/EMPLOYEE.FDB';
```

is equivalent to the second example—slashes may be forward or backward in *isql*.

Server (host) and path names

On Windows, don't confuse server names and shared disk resource names. The client/server network layer does not recognize mapped drives or shares. A drive identifier must always point to the actual drive-letter of a hard-drive or partition on the server host machine.

User authentication

Regardless of whether you log in directly from the command-line or do so when connecting inside the *isql* shell, authentication will fail if the server does not recognise the username or the password. For example, you will see this if your **CONNECT** statement fails:

Figure 24.3 Failed authentication

```

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\>cd Program Files\Firebird\Firebird_2_5\bin
C:\Program Files\Firebird\Firebird_2_5\bin>isql
Use CONNECT or CREATE DATABASE to specify a database
SQL> CONNECT 'L:\data\examples\employee.fdb'
CON> user 'SYSDBA' password 'monsterkey';
Statement failed, SQLCODE = -902
Your user name and password are not defined. Ask your
database administrator to set up a Firebird login.
SQL> _

```

If this happens to you, double-check that you have spelt the username and password correctly; and that the password is correct for case. Passwords are case-sensitive; user names are not.

Using the Interface

Now you are logged in to a database, you can begin using *isql* to work with your data and metadata. Before getting into the work you can do with this utility, it will be useful to understand something about the interface and how it works with your work.

SET commands

Several commands are available inside *isql* that, in most cases, are not valid in any other environment. These are the commands using the **SET** verb, which can be used to set up various conditions within your *isql* session. We look at some of them in this section, while others are discussed a little later.

The continuation prompt

If you press Enter without remembering to finish a statement with a terminator, you will see the continuation prompt **CON>** instead of the *isql* prompt **SQL>**:

```
SQL> SHOW DATABASE
CON>
```

If it was a mistake, simply type the terminator character and press Enter again.

```
SQL> SHOW DATABASE
CON> ;
```

```
SQL>
```

However, you can use this feature to make your typing easier to read. For example:

```
SQL> CREATE TABLE ATABLE (
CON>   ID INTEGER NOT NULL,
CON>   DATA VARCHAR(20),
CON>   DATE_ENTERED DATE
CON>   DEFAULT CURRENT_DATE
CON> );
SQL>
```



*One good reason to use the continuation feature at times is that you can use the **OUTPUT** command to pipe your *isql* input to a file (q.v.). Since the output is saved exactly as you type it, any blank-space indenting will be preserved. Many Firebird users use *isql* as their only script editor!*

The terminator character

The default statement terminator is the semicolon (;), which is used for all of the examples in this chapter. You can change the terminator to any character or group of characters with the **SET TERM[INATOR]** command. For example, to change it to '!!', use this statement:

```
SQL> SET TERM !!;
SQL>
```

Now, if you try to use the semicolon as the terminator, *isql* will assume you have an unfinished statement:

```
SQL> SHOW DATABASE;
CON>
```

Then, if you supply the new terminator character to finish the statement, *isql* complains:

```
CON> !!
Command error: SHOW DATABASE;
SQL>
```

Transactions in isql

Transaction management in *isql* differs according to whether you issue a DDL statement, a **SHOW** command or other kinds of statements.

When *isql* starts, it starts a transaction in **SNAPSHOT** (concurrency) isolation with a lock resolution setting of **WAIT**. Unless you run DDL statements or **SHOW** commands, the transaction stays current—and thus uncommitted—until you issue a **COMMIT** or **ROLLBACK** statement.

You can start an explicit transaction by committing the current transaction and using a **SET TRANSACTION** statement to start a new one. For example, to start a **READ COMMITTED NO WAIT** transaction:

```
SQL> COMMIT;
SQL> SET TRANSACTION
CON> NO WAIT READ COMMITTED;
```

When you have finished your task, just issue a **COMMIT** statement as usual. The next statement will revert to the default configuration.

DDL statements

Each time you issue a DDL statement—those are the ones that define, modify or drop metadata objects—*isql* starts a special transaction for it and commits it immediately you press Enter. A new transaction is started immediately afterwards. You can change this automatic behavior by issuing the SET AUTODDL OFF command from the SQL prompt before you begin issuing your DDL statements:

```
SQL> SET AUTODDL OFF;
```

To switch back to autocommit mode for DDL statements:

```
SQL> SET AUTODDL ON;
```

For switching back and forth between autodddl *on* and *off*, a short version is available that simply sets autodddl off if it is on, and vice versa:

```
SQL> SET AUTO;
```



The autodddl feature works only with DDL statements.

SHOW commands

The SHOW commands query the system tables. Whenever you invoke a SHOW command, *isql* commits the existing transaction and starts a new one in READ COMMITTED isolation. This ensures that you always have an up-to-date view of metadata changes as soon as they occur.

Retrieving the line buffer

isql allows you to retrieve the line buffer, in a similar fashion to the way the *readline* feature works on POSIX platforms. Use the up and down arrow keys to “scroll through” the program’s command buffer, a line at a time, to retrieve copies of lines you typed previously.

Using warnings

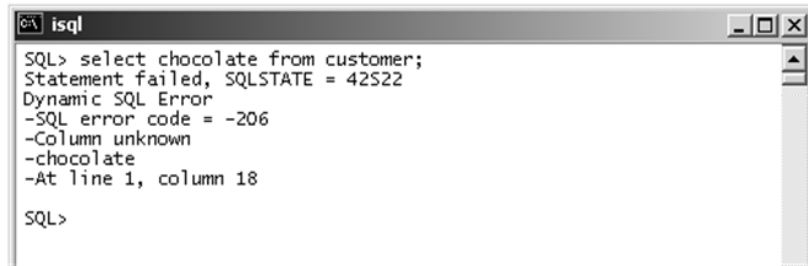
By default, *isql* issues warnings for certain conditions, for example:

- statements with no effect
- ambiguous join specifications in Firebird 1.0.x (in v.1.5 and higher, they will cause exceptions)
- expressions that produce different results in different versions of Firebird
- API calls that will be replaced in future versions
- when a database shutdown is pending

For toggling the display off and on during an interactive *isql* session, use SET WARNINGS or its shorthand counterpart, SET WNG.

Exception handling

SQL errors in are handled and delivered in much the same way as they are in any DSQL application—*isql* displays an error message consisting of the SQLSTATE code, the SQLCODE (SQL error code) and the text message from the Firebird status array:

Figure 24.4 Example of an error message in *isql*


```

C:\ isql
SQL> select chocolate from customer;
Statement failed, SQLSTATE = 42S22
Dynamic SQL Error
-SQL error code = -206
-Column unknown
-chocolate
-At line 1, column 18
SQL>

```

When an SQL error has occurred, all versions display the SQL error code. SQL errors with sub-zero SQLCODEs mean the statement has failed to execute. They are all listed in Appendix VII. You may also see one of the SQL warning or information messages, viz.

- 0 SUCCESS (successful execution).
- +1—99 SQLWARNING (system warning or information message).
- +100 NOT FOUND (indicates that no qualifying rows were found, or “end-of-file”, i.e. the end of the current active set of rows was detected).



The SQLSTATE codes appear in isql versions 2.5 and higher. A reference listing for them is in Appendix VIII.

Dialect in *isql*

If you start the *isql* client program and attach to a database without specifying a dialect, *isql* takes on the dialect of the database.

You can set the *isql* dialect yourself, in the following ways:

- When starting *isql*:
 `bin] isql -s n`
 where *n* is 1, 2, or 3.
 If you specify the dialect this way, *isql* retains that dialect after connection unless you explicitly change it.
- Within an *isql* session or in a SQL script
 `SET SQL DIALECT n;`
 isql continues to operate in that dialect unless it is explicitly changed.

The dialect can not be set as a parameter of a CREATE DATABASE statement.



When you create a database interactively using isql, the database will be in the dialect that is current in isql at the time the CREATE DATABASE statement is issued. You need to watch this if you had a dialect 1 database open previously, because isql stays in dialect 1 after it disconnects from the dialect 1 database.

Dialect effects

A dialect 1 client processes all commands according to the expectations of the ancient legacy InterBase 5 language and syntax, with certain variations. The effects may also show

- some variations according to which dialect is in force. For example, if you create a table that specifies a column of type DATE, you will see an info message telling you “DATE data type is now called TIMESTAMP”.
- In a dialect 2 client, elements that have different interpretations in dialect 1 and 3 are all flagged with warnings or errors, to assist in migrating databases to dialect 3
 - A dialect 3 client parses all statements according to native Firebird SQL semantics: double quotes are delimited identifiers and are not recognized as string delimiters, the DATE data type is date-only and exact numerics with precision greater than 9 are stored as BIGINT

v.1.0.x `BIGINT=(NUMERIC(18,0))`

SELECT statements

The console output from SELECT statements pours forth until there is no more data to be fetched. Columns are extended to the maximum defined width and there is no way to shorten or hide them. In practice, SELECT * queries with no WHERE clause will be not be much use to you.

Sometimes you will find it useful to use the OUTPUT command to pass the fetched data directly to a text file, where you can use a suitable editor to view it.



From the “2” series onward, isql displays CHAR and VARCHAR types defined in character set OCTETS (alias BINARY) in hexadecimal format.

BLOBs

By default, *isql* does not display the contents of BLOB columns, just their hexadecimal blob_ids. You can use the command SET BLOBDISPLAY / SET BLOB to change this behaviour.

You can use the blob_id with BLOBDUMP to dump the contents of a BLOB to a file and with BLOBVIEW to view (but not edit) a text BLOB in the default editor.

Stopping query output

From v.2.1 onward, output from a SELECT in an interactive *isql* session on any platform can be stopped using Ctrl-C. It does not cancel the query but it does stop the avalanche of data arriving on your screen or pouring into your output file.

Interactive Commands

- You can enter three kinds of commands or statements interactively at the `SQL>` prompt:
- SQL data definition (DDL) statements, such as CREATE, ALTER, DROP, GRANT and REVOKE. These statements create, modify or remove metadata and objects, or control user access permission (privileges) to the database.
 - SQL data manipulation (DML) statements such as SELECT, INSERT, UPDATE and DELETE. The output of SELECT statements can be displayed or directed to a file (see the OUTPUT command).
 - *isql* commands, which fall into three main categories:
 - General commands (for example, commands to read an input file, write to an output file, or end an *isql* session)

- SHOW commands (to display metadata or other database information)
- SET commands (to modify the isql environment)

Scripts

While it is possible to build a database by submitting and committing a series of DDL statements during an interactive *isql* session, this is an ad hoc approach that leaves you with no documentation of what you did and potential holes in your QA review process.

It is very good practice to use a script to create your database and its objects. A script for creating and altering database objects is sometimes known as a *schema script*, a *data definition file* or just a *DDL script*.

In *isql* you will find utility commands both for creating scripts from your interactive command stream (OUTPUT) and for streaming one or a sequence of script files into the application (INPUT) through either the interactive interface or the command shell.

The topic of schema scripting is covered in detail later in this chapter, in the section Creating and Running Scripts.

General *isql* Commands

The general isql commands perform a variety of useful tasks, including reading, recording and processing schema scripts and executing shell commands. They are: BLOBDUMP, BLOBVIEW, EDIT, EXIT, HELP, INPUT, OUTPUT, QUIT and SHELL.

BLOBDUMP

stores BLOB data into a named file

```
BLOBDUMP blob_id filename ;
```

Arguments

blob_id	Identifier consisting of two hex numbers separated by a colon (:). The first number is the ID of the table containing the BLOB column, the second is a sequenced instance number. To get the blob_id, issue any SELECT statement that selects a column of BLOB data. The output will show the hex blob_id above or in place of the BLOB column data, depending on whether SET BLOB[DISPLAY] is ON or OFF.
filename	Fully qualified filesystem name of the file which is to receive the data.

```
Example      SQL> BLOBDUMP 32:d48 IMAGE.JPG ;
```

BLOBVIEW

displays BLOB data in the default text editor.

```
BLOBVIEW blob_id ;
```

Argument

blob_id Identifier consisting of two hex numbers separated by a colon (:). See BLOBDUMP for instructions on how to determine the blob_id you are looking for. In current versions, BLOBVIEW does not support on-line editing of the BLOB. It may be introduced in a future release.

Example SQL> BLOBVIEW 85:7 ;



BLOBVIEW may return an “Invalid transaction handle” error after you close the editor. It is a bug. To correct the situation, start a transaction manually, with

SQL> SET TRANSACTION;

EDIT

allows editing and re-execution of the previous isql command or of a batch of commands in a source file.

SQL> EDIT [filename];

Argument

filename Optional, fully qualified filesystem name of file to edit.

Example SQL> EDIT /usr/mystuff/batch.sql

EDIT can also be used to open the previous statements in your editor:

SQL> SELECT EMP_CODE, EMP_NAME FROM EMPLOYEE ;

SQL> EDIT ;

Press Enter to display the “scroll” from your *isql* session in your text editor. Edit it, save it if you wish, and exit. The edited batch of commands will be re-executed in your *isql* shell when you exit the editor.

EXIT

commits the current transaction without prompting, closes the database and ends the isql session. If you need to roll back the transaction instead of committing it, use QUIT instead.

SQL> EXIT ;

EXIT takes no arguments.

HELP

displays a list of *isql* commands with descriptions. You can combine it with OUTPUT to print the list to a file.

SQL> HELP ;

Example SQL> OUTPUT HELPLIST.TXT ;

SQL> HELP ;

SQL> OUTPUT ; /* toggles output back to the monitor */

HELP takes no arguments.

INPUT

reads and executes a block of commands from a named text file (SQL script). Input files can embed other INPUT commands, thus providing the capability to designed chained or structured suites of DDL scripts. To create scripts, use a text editor or build them interactively, use the OUTPUT or EDIT commands.

```
SQL> INPUT filename ;
SQL> EDIT [filename];
```

Argument

filename	Fully qualified filesystem name of file containing SQL statements and commands to be opened executed, statement by statement.
----------	---

Example SQL> INPUT /data/schemascripts/myscript.sql ;

In a script:

```
...
CREATE EXCEPTION E010 'This is an exception.';
COMMIT;
-- TABLE DEFINITIONS
INPUT '/data/schemascripts/tabledefs.sql';
-- CONSTRAINT DEFINITIONS
INPUT 'data/schemascripts/constraintdefs.sql';
...
```

OUTPUT

redirects output to a disk file or (back) to the standard output device (monitor). Use SET ECHO commands to include or exclude commands:

- SET ECHO ON to output both commands and data
- SET ECHO OFF to output data only

```
SQL> OUTPUT [filename];
```

Argument

filename	Optional, fully qualified filesystem path to a file containing SQL statements and commands. If no file name is given, results appear on the standard monitor output, i.e., output-to-file is switched off
----------	---

Example SQL> OUTPUT d:\data\employees.dta ;
 SQL> SELECT EMP_NO, EMP_NAME FROM EMPLOYEE ; /* output goes to file */
 SQL> OUTPUT ; /* toggles output back to the monitor */

Tip *If you are using OUTPUT to build scripts, it will be necessary to edit them to remove any stray interactive isql commands. However, when you “replay” output in isql using INPUT, isql usually just ignores the echoed interactive commands.*

QUIT

Rolls back the current transaction and closes the isql shell.

```
SQL> QUIT ;
```

QUIT takes no arguments. If you need to commit the transaction instead of rolling it back, use EXIT instead.

SHELL

gives temporary access to a command-line shell without committing or rolling back any transaction.

```
SQL> SHELL [operating system command];
```

Argument

operating system command	Optional, a command or call that is valid in command shell from which <i>isql</i> was launched. The command will be executed and control returned to <i>isql</i> . If no command is specified, <i>isql</i> opens an interactive session in the command shell. Typing <i>exit</i> returns control to <i>isql</i> .
--------------------------	---

Example SQL> SHELL dir /mydir | more ;

The example will display the contents of the directory /mydir and return control to *isql* when the display completes or if the *more* utility is terminated by Ctrl-C.

SHOW Commands

SHOW commands are used to display metadata, including tables, indexes, procedures, triggers and privileges. They can list the names of all objects of the specified type or supply detailed information about a particular object named in the command.

The SHOW commands are (approximately) the interactive equivalent of the command-line -extract, -x or -a options (q.v.). However, although you can use the OUTPUT command to send the output of the SHOW commands to a file, the saved text is not ready to use as a schema script without editing. Use the command-line options if obtaining a schema script is your goal.

Each SHOW command runs in its own READ COMMITTED statement, ensuring that each call returns the most up-to-date view of the state of the database.

SHOW CHECK

displays the names and sources for all user-defined CHECK constraints defined for a specified table.

```
SQL> SHOW CHECK tablename ;
```

Argument

tablename	Name of a table that exists in the attached database
-----------	--

Example ...
SQL> SHOW CHECK JOB ;
CONSTRAINT INTEG_12
CHECK (min_salary < max_salary)

SHOW COLLATIONS

From v.2.1 onward, lists all the user-declared character set/collation pairs in the database.



Even if you have no user-declared character sets, you can still list out all of the installed character sets using

```
SELECT * FROM RDB$CHARACTER_SETS;
```

SHOW DATABASE

displays information about the attached database (file name, page size and allocation, sweep interval, transaction numbers, Forced Writes status, default character set). SHOW DB is a shorthand version of the command.

```
SQL> SHOW DATABASE | DB ;
```

SHOW DATABASE takes no arguments.

Figure 24.5 SHOW DATABASE output

```
isql
SQL> SHOW DATABASE;
Database: hotchicken:emp3
      Owner: SYSDBA
PAGE_SIZE 8192
Number of DB pages allocated = 288
Sweep interval = 20000
Forced Writes are ON
Transaction - oldest = 112
Transaction - oldest active = 761
Transaction - oldest snapshot = 761
Transaction - Next = 768
ODS = 11.2
Default Character set: NONE
SQL>
```

V.1.5.X For the old versions, use SHOW VERSION to inspect the on-disk structure.

SHOW DOMAIN[S]

displays domain information.

```
SQL> SHOW { DOMAINS | DOMAIN name };
```

Variations

SHOW DOMAINS Lists the names of all the domains declared in the database

SHOW DOMAIN *name* Displays definition of the named single domain

Examples

```
SQL> SHOW DOMAINS ;
D_CURRENCY D_NOTES
D_BOOLEAN D_PHONEFAX
... ..
SQL> SHOW DOMAIN D_BOOLEAN ;
D_BOOLEAN SMALLINT NOT NULL
DEFAULT 0
CHECK (VALUE IN (0,1))
```

SHOW EXCEPTION[S]

displays information about user-defined exceptions.

```
SQL> SHOW { EXCEPTIONS | EXCEPTION name };
```

Variations

SHOW EXCEPTIONS	Lists the names and texts of all exceptions declared in the database
SHOW EXCEPTION <i>name</i>	Displays text of the named single exception

Examples ...

```
SQL> SHOW EXCEPTIONS ;
Exception Name Used by, Type
=====
BAD_WIZ_TYPE UPD_FAVEFOOD, Stored procedure
Invalid Wiz type, check CAPS LOCK
...
SQL> SHOW EXCEPTION BAD_WIZ_TYPE ;
Exception Name Used by, Type
=====
BAD_WIZ_TYPE UPD_FAVEFOOD, Stored procedure
Invalid Wiz type, check CAPS LOCK
```

SHOW FUNCTION[S]

displays information about external functions declared in the attached database.

```
SQL> SHOW { FUNCTIONS | FUNCTION name };
```

Variations

SHOW FUNCTIONS	Lists the names of all external functions declared in the database
SHOW FUNCTION <i>name</i>	Displays the declaration of the named external function

Examples ...

```
SQL> SHOW FUNCTIONS ;
ABS MAXNUM
LOWER SUBSTRLEN
...
SQL> SHOW FUNCTION maxnum ;
Function MAXNUM:
Function library is /usr/firebird/udf/ib_udf.so
Entry point is FN_MAX
Returns BY VALUE DOUBLE PRECISION
Argument 1: DOUBLE PRECISION
Argument 2: DOUBLE PRECISION
```

SHOW GENERATOR[S]

displays information about generators and sequences declared in the attached database.

```
SQL> SHOW { GENERATORS | GENERATOR name };
```

Variations

SHOW GENERATORS	Lists the names of all generators declared in the database, along with their current values
SHOW GENERATOR <i>name</i>	Displays the declaration of the named generator, along with its current value

Examples

```
...
SQL> SHOW GENERATORS ;
Generator GEN_EMPNO, Next value: 1234
Generator GEN_JOBNO, Next value: 56789
Generator GEN_ORDNO, Next value: 98765
...
SQL> SHOW GENERATOR gen_ordno ;
Generator GEN_ORDNO, Next value: 98765
```

SHOW GRANT

displays privileges and role ownership information about a named object in the attached database; or displays user membership within roles.

```
SQL> SHOW GRANT { object | rolename };
```

Argument Options

SHOW GRANT <i>object</i>	Takes the name of an existing table, view or procedure in the current database
SHOW GRANT <i>rolename</i>	Takes the name of an existing role in the current database. Use SHOW ROLES to list all the roles defined for this database.

Examples

```
...
SQL> SHOW GRANT JOB ;
GRANT SELECT ON JOB TO ALL
GRANT DELETE, INSERT, SELECT, UPDATE ON JOB TO MANAGER
SQL> SHOW GRANT DO_THIS ;
GRANT DO_THIS TO MAGICIAN
```

SHOW INDEX (SHOW INDICES)

displays information about a named index, about indices for a specified table or about indices for all tables in the attached database. The command can be abbreviated to SHOW IND.

```
SQL> SHOW {INDICES | INDEX [{ index | table }]};
```

Variations

SHOW INDEX	Prints details of all indexes in the database
SHOW INDEX <i>index</i>	Takes the name of an existing index in the current database and prints its details
SHOW INDICES <i>table</i>	Takes the name of an existing table in the current database and prints details of all its indexes

Examples

```
...
SQL> SHOW INDEX ;
RDB$PRIMARY1 UNIQUE INDEX ON COUNTRY(COUNTRY)
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)
CUSTREGION INDEX ON CUSTOMER(COUNTRY, CITY)
RDB$FOREIGN23 INDEX ON CUSTOMER(COUNTRY)
...
SQL> SHOW IND COUNTRY ;
RDB$PRIMARY20 UNIQUE INDEX ON CUSTOMER(CUSTNO)
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)
```



For information about the current states of indexes in the database, use *gstat -i*. Use of the *gstat* utility is described in Chapter 38, **Monitoring and Logging Features**, in the section Collecting Database Statistics—*gstat*.

SHOW PROCEDURE[S]

lists all procedures in the attached database, with their dependencies; or displays the text of the named procedure with the declarations and types (input/output) of any arguments. The command can be abbreviated to SHOW PROC.

```
SQL> SHOW {PROCEDURES | PROCEDURE name } ;
```

Variations

SHOW PROCEDURES	Lists out all procedures by name, together with their dependencies.
SHOW PROCEDURE <i>name</i>	For the named procedure, lists the source, dependencies and arguments.

Examples

```
SQL> SHOW PROCEDURES ;
Procedure Name Dependency Type
=====
ADD_EMP_PROJ EMPLOYEE_PROJECT Table
UNKNOWN_EMP_ID Exception
DELETE_EMPLOYEE DEPARTMENT Table
EMPLOYEE Table
EMPLOYEE_PROJECT Table
...
SQL> SHOW PROC ADD_EMP_PROJ ;
Procedure text:
=====
BEGIN
```



```

BEGIN
INSERT INTO EMPLOYEE_PROJECT (
EMP_NO, PROJ_ID)
VALUES (
:emp_no, :proj_id) ;
WHEN SQLCODE -530 DO
EXCEPTION UNKNOWN_EMP_ID;
END
RETURN ;
END
=====
Parameters:
EMP_NO INPUT SMALLINT
PROJ_ID INPUT CHAR(5)

```

SHOW ROLE[S]

displays the names of SQL roles for the attached database.

```
SQL> SHOW ROLES ;
```

SHOW ROLES takes no arguments.

Examples

```

...
SQL> SHOW ROLES ;
MAGICIAN MANAGER
PARIAH SLEEPER
...

```

To show user membership within roles, use **SHOW GRANT <rolename>**.

SHOW SQL DIALECT

displays the SQL dialects of the client and of the attached database, if there is one.

```
SQL> SHOW SQL DIALECT;
```

Example

```

...
SQL> SHOW SQL DIALECT;
Client SQL dialect is set: 3 and database SQL dialect is: 3

```

SHOW SYSTEM

displays the names of system tables, system views (if any) and, from V.2.0, pre-defined UDFs, for the attached database. It can be abbreviated to **SHOW SYS**.

```
SQL> SHOW SYS [ TABLES ] ;
```

The command takes no arguments. TABLES is an optional keyword that does not affect the behavior of the command.

Examples

```

...
SQL> SHOW SYS ;
RDB$CHARACTER_SETS RDB$CHECK_CONSTRAINTS
RDB$COLLATIONS RDB$DATABASE
...

```

For more detailed information about the system tables, see Appendix V.

SHOW TABLE[S]

lists all tables or views, or displays information about the named table or view.

```
SQL> SHOW { TABLES | TABLE name };
```

Variations

SHOW TABLES	Lists out names of all tables and views in alphabetical order, by name
SHOW TABLE <i>name</i>	Shows details about the named table or view. If the object is a table, the output contains column names and definitions, PRIMARY KEY, FOREIGN KEY and CHECK constraints, and triggers. If the object is a view, the output contains column names and the SELECT statement that the view is based on.

Examples

```
...
SQL> SHOW TABLES ;
COUNTRY CUSTOMER
DEPARTMENT EMPLOYEE
EMPLOYEE_PROJECT JOB
...
SQL> SHOW TABLE COUNTRY ;
COUNTRY COUNTRYNAME VARCHAR(15) NOT NULL
CURRENCY VARCHAR(10) NOT NULL
PRIMARY KEY (COUNTRY)
SQL> show table employee;
EMP_NO                (EMPNO) SMALLINT Not Null
FIRST_NAME            (FIRSTNAME) VARCHAR(15) Not Null
LAST_NAME             (LASTNAME) VARCHAR(20) Not Null
PHONE_EXT             VARCHAR(4) Nullable
HIRE_DATE             TIMESTAMP Not Null DEFAULT 'NOW'
DEPT_NO              (DEPTNO) CHAR(3) Not Null
                     CHECK (VALUE = '000' OR (VALUE > '0' AND VALUE <
= '999') OR VALUE IS NULL)
JOB_CODE              (JOBCODE) VARCHAR(5) Not Null
                     CHECK (VALUE > '99999')
JOB_GRADE             (JOBGRADE) SMALLINT Not Null
                     CHECK (VALUE BETWEEN 0 AND 6)
JOB_COUNTRY           (COUNTRYNAME) VARCHAR(15) Not Null
SALARY               (SALARY) NUMERIC(10, 2) Not Null DEFAULT 0
                     CHECK (VALUE > 0)
FULL_NAME             Computed by: (last_name || ', ' || first_name)
CONSTRAINT INTEG_28:
    Foreign key (DEPT_NO) References DEPARTMENT (DEPT_NO)
CONSTRAINT INTEG_29:
```

```

Foreign key (JOB_CODE, JOB_GRADE, JOB_COUNTRY)    References JOB (JOB_CODE, JO
B_GRADE, JOB_COUNTRY)
CONSTRAINT INTEG_27:
    Primary key (EMP_NO)
CONSTRAINT INTEG_30:
    CHECK ( salary >= (SELECT min_salary FROM job WHERE
                        job.job_code = employee.job_code AND
                        job.job_grade = employee.job_grade AND
                        job.job_country = employee.job_country) AND
            salary <= (SELECT max_salary FROM job WHERE
                        job.job_code = employee.job_code AND
                        job.job_grade = employee.job_grade AND
                        job.job_country = employee.job_country))

```

Triggers on Table EMPLOYEE:

SET_EMP_NO, Sequence: 0, Type: BEFORE INSERT, Active

SAVE_SALARY_CHANGE, Sequence: 0, Type: AFTER UPDATE, Active

SQL>

See also SHOW VIEWS (below).

SHOW TRIGGER[S]

displays all triggers defined in the database, along with the table they depend on; or, for the named trigger, displays its sequence, type, activity status (active/inactive) and PSQL definition. It can be abbreviated to SHOW TRIG.

```
SQL> SHOW {TRIGGERS | TRIGGER name } ;
```

Variations

SHOW TRIGGERS Lists out all table names with their trigger names alphabetically

SHOW TRIGGER *name* For the named trigger, identifies the table it belongs to, displays the header parameters, activity status and PSQL source of the body.

Examples

```

SQL> SHOW TRIGGERS ;
Table name Trigger name
=====
EMPLOYEE SET_EMP_NO
EMPLOYEE SAVE_SALARY_CHANGE
CUSTOMER SET_CUST_NO
SALES POST_NEW_ORDER
SQL> SHOW TRIG SET_CUST_NO ;
Trigger:
SET_CUST_NO, Sequence: 0, Type: BEFORE INSERT, Active
AS
BEGIN
    new.custno = gen_id(cust_no_gen, 1);
END

```

SHOW VERSION

displays information about the software versions of *isql* and the Firebird server program, and the on-disk structure of the attached database. It can be abbreviated to SHOW VER.

SQL> SHOW VERSION ;

The command takes no arguments.

Example ...

```
SQL> SHOW VER ;
ISQL Version: WI-V2.5.0.26074 Firebird 2.5
Server version:
Firebird/x86/Windows NT (access method), version "WI-V2.5.0.26074 Firebird 2.5"
Firebird/x86/Windows NT (remote server), version "WI-V2.5.0.26074 Firebird 2.5/XNet (DEV)/P12"
Firebird/x86/Windows NT (remote interface), version "WI-V2.5.0.26074 Firebird 2.5/XNet (DEV)/P12"
on disk structure version 11.2
```

SHOW VIEW[S]

lists all views, or displays information about the named view.

See also SHOW TABLES.

SQL> SHOW { VIEWS | VIEW name } ;

Variations

SHOW VIEWS	Lists out the names of all views in alphabetical order
SHOW VIEW <i>name</i>	The output displays column names and the SELECT statement that the view is based on.

Example

```
SQL> SHOW VIEWS ;
PHONE_LIST CUSTOMER
...
SQL> SHOW VIEW PHONE_LIST;
EMP_NO                (EMPNO) SMALLINT Not Null
FIRST_NAME            (FIRSTNAME) VARCHAR(15) Not Null
LAST_NAME             (LASTNAME) VARCHAR(20) Not Null
PHONE_EXT             VARCHAR(4) Nullable
LOCATION               VARCHAR(15) Nullable
PHONE_NO              (PHONENUMBER) VARCHAR(20) Nullable
View Source:
==== =====
SELECT
    emp_no, first_name, last_name, phone_ext, location, phone_no
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

SET Commands

The SET commands enable you to view and change things about the *isql* environment. Some are available in scripts.

Many of the SET commands act as “toggles” that switch the feature on if it is off, or off if it is on. Using SET on its own will display the state of all the toggle settings.

SET AUTODDL | SET AUTO

specifies whether DDL statements are committed automatically after being executed, or committed only after an explicit COMMIT.

Available in scripts.

```
SQL> SET AUTODDL [ON | OFF] ; /* default is ON */
```

Options

SET AUTODDL ON	Toggles automatic commit on for DDL
SET AUTODDL OFF	Toggles automatic commit off for DDL
SET AUTO	With no argument, simply toggles AUTODDL on and off.

Example

```
...
SQL> SET AUTODDL OFF ;
SQL> CREATE TABLE WIZZO (x integer, y integer) ;
SQL> ROLLBACK; /* Table WIZZO is not created */
...
SQL>SET AUTO ON ;
SQL> CREATE TABLE WIZZO (x integer, y integer) ;
SQL> /* table WIZZO is created */
```

SET BAIL

When set ON, causes script execution to end and return on the first error encountered. It can be set either as a SET BAIL command in a script or passed as the `-b[ail]` switch when executing an input script from a command-line call to *isql*. If a “bail-out” occurs, script execution ends and a fail code—merely a non-zero number—is returned to the operating system. The error message will appear at the console or in the output file, according to other options passed and, in some cases, can return the line number where the error occurred.

- DML errors will be caught either at prepare time or at execution, according to the type of error that occurs.
- DDL errors will be caught at prepare time or at execution unless AUTODDL is off. In that case, the error will occur when the script issues an explicit COMMIT. In the latter case, it may be difficult to detect which statement actually triggers the bail-out.

Options

SET BAIL ON	Enables bail out on error
SET BAIL OFF	Disables bail out on error
SET BAIL	With no argument, simply toggles BAIL on and off.

Example `isql -b -i my_fb.sql -o results.log -m -m2`

Note *isql will accept a SET BAIL command during an interactive session but it achieves nothing, even if the session is processing an INPUT script and encounters an error.*

SET BLOBDISPLAY | SET BLOB

specifies both sub_type of BLOB to display and whether BLOB data should be displayed.

`SQL> SET BLOBDISPLAY [n |ALL |OFF] ;`

SET BLOB is a shortened version of the same command.

Options and Arguments

<i>n</i>	BLOB SUB_TYPE to display. Default: <i>n</i> = 1 (text). Positive numbers are system-defined; negative numbers are user-defined.
SET BLOB[DISPLAY] ALL	Display BLOB data of any sub_type
SET BLOB[DISPLAY] OFF	Toggles display of BLOB data off. The output shows only the BlobID (two hex numbers separated by a colon (:). The first number is the ID of the table containing the BLOB column. The second is a sequenced instance number.

Example ...

```
SQL> SET BLOBDISPLAY OFF ;
SQL> SELECT PROJ_NAME, PROJ_DESC FROM PROJECT ;
SQL> /* rows show values for PROJ_NAME and Blob ID */
...
SQL>SET BLOB 1 ;
SQL> SELECT PROJ_NAME, PROJ_DESC FROM PROJECT ;
SQL> /* rows show values for PROJ_NAME and Blob ID */
SQL> /* and the blob text appears beneath each row */
```

SET COUNT

toggles off/on whether to display the number of rows retrieved by queries.

`SQL> SET COUNT [ON | OFF] ;`

Options

SET COUNT ON	Toggles on display of “rows returned” message
SET COUNT OFF	Toggles off display of “rows returned” message (default)

Example ...

```
SQL> SET COUNT ON ;
SQL> SELECT * FROM WIZZO WHERE FAVEFOOD = 'Pizza' ;
SQL> /* displays the data, followed by */
...
40 rows returned
```

SET ECHO

toggles off/on whether commands are displayed before being executed. Default is ON but you might want to toggle it to OFF if sending your output to a script file.

```
SQL> SET ECHO [ON | OFF] ; /* default is ON */
```

Options

SET ECHO ON	Toggles on command echoing (default)
SET ECHO OFF	Toggles off command echoing

Example Script wizzo.sql:

```
...
SET ECHO OFF;
SELECT * FROM WIZZO WHERE FAVEFOOD = 'Pizza' ;
SET ECHO ON ;
SELECT * FROM WIZZO WHERE FAVEFOOD = 'Sardines' ;
EXIT;
...
SQL > INPUT wizzo.sql ;
WIZTYPE FAVEFOOD
=====
alpha Pizza
epsilon Pizza
SELECT * FROM WIZZO WHERE FAVEFOOD = 'Sardines' ;
WIZTYPE FAVEFOOD
=====
gamma Sardines
lamda Sardines
```

SET HEAD[ING]

toggles off/on the displaying of column headers in output from SELECT statements, useful if you are passing the output to a file. By default, headings are ON. SET HEAD and SET HEADING are both valid. This command can be used in scripts.

SET NAMES

specifies the character set that is to be active in database transactions. This is very important if your database's default character set is not NONE. If the client and database character sets are mismatched, you risk transliteration errors and storing wrong data if you use isql for performing updates or inserts or for searches (including searched updates and deletes).

SET NAMES is available in scripts.

```
SQL> SET NAMES character-set ;
```

Argument

character-set Name of the character set to activate. Default: NONE

Example In script:

```
...
SET NAMES ISO8859_1 ;
CONNECT 'HOTCHICKEN:/usr/firebird/examples/employee.gdb' ;
```

SET PLAN

specifies whether to display the optimizer's query plan.

```
SQL> SET PLAN [ON | OFF ];
```

Options

- SET PLAN ON Turns on display of the query plan (default)
- SET PLAN OFF Turns off display of the query plan.
- SET PLAN With no arguments, can be used as an ON/OFF toggle.

Example In a script:

```
...
SET PLAN ON ;
SELECT JOB_COUNTRY, MIN_SALARY
FROM JOB
WHERE MIN_SALARY > 50000
AND JOB_COUNTRY = 'Sweden';
...
SQL> INPUT iscript.sql
PLAN (JOB INDEX (RDB$FOREIGN3,MINSALX,MAXSALX)
JOB_COUNTRY MIN_SALARY
=====
Sweden 120550.00
```

SET PLANONLY

specifies to prepare SELECT queries and display just the plan, without executing the actual query.

```
SQL> SET PLANONLY ON | OFF;
```

The command works as a toggle switch. The argument is optional.

SET ROWCOUNT

specifies a limit to be the number of rows returned by a query .

The following statement will stop returning rows after the 100th row of any query in the session:

```
SQL> SET ROWCOUNT 100;
```


SET SQLDA_DISPLAY

Shows the information for raw SQLVARs in the input SQLDA parameters of INSERTs, UPDATEs and DELETEs. Each SQLVAR represents a field in the XSQLDA, the main structure used in the FB API to talk to clients transferring data into and out of the server.

Options

SET SQLDA_DISPLAY ON Turns on display of the SQLVARs.

SET SQLDA_DISPLAY OFF Turns off display of the SQLVARs (default).

In versions prior to v.2.0 it is available only in DEBUG builds.

SET SQL DIALECT

Sets the Firebird SQL dialect to which the client session is to be changed. If the session is currently attached to a database of a different dialect to the one specified in the command, a warning is displayed and you are asked whether you want to commit existing work (if any).

SQL> SET SQL DIALECT *n* ;

Argument

n Dialect number. *n* = 1 for Dialect 1, 2 for Dialect 2, 3 for Dialect 3

Example SQL> SET SQL DIALECT 3 ;

SET STATS

Specifies whether to display performance statistics following the output of a query.

SQL> SET STATS [ON |OFF];

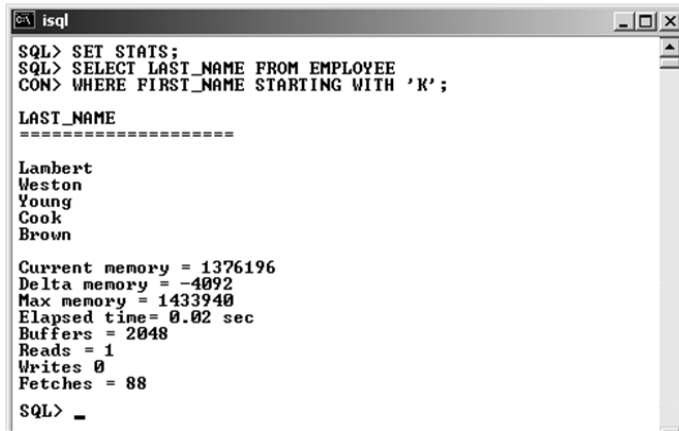
Options

SET STATS ON Turns on display of performance statistics.

SET STATS OFF Turns off display of performance statistics. Default.

SET STATS You can omit ON | OFF and use just SET STATS as a toggle.

Figure 24.6 SET STATS example



```

isql
SQL> SET STATS;
SQL> SELECT LAST_NAME FROM EMPLOYEE
CON> WHERE FIRST_NAME STARTING WITH 'K';

LAST_NAME
=====
Lambert
Weston
Young
Cook
Brown

Current memory = 1376196
Delta memory = -4092
Max memory = 1433940
Elapsed time= 0.02 sec
Buffers = 2048
Reads = 1
Writes 0
Fetches = 88
SQL> _
  
```



After issuing *SET STATS ON*, call *COMMIT*; to prompt *isql* to display the information about memory and buffer usage.

SET STATISTICS

SET STATISTICS is an SQL (not ISQL) command that you can use in *isql*—as well as in other programs—to recompute the selectivity of an index. It is mentioned here because, not surprisingly, people often get it confused with SET STATS. To find out why selectivity can be important in very dynamic tables, refer to the topic [Dropping an Index](#) in Chapter 16.

The syntax for the command, available only to the user that owns the index, is

```
SET STATISTICS index-name
```

SET TERM

Specifies the character which will be used as the command or statement terminator, from the next statement forward. Available in scripts. See the notes about this command earlier in this chapter.

```
SQL> SET TERM string ;
```

Argument

string

Character or characters which will be used as statement terminator. The default is “;”

Example

```

...
SET TERM ^^;
CREATE PROCEDURE ADD_WIZTYPE (WIZTYPE VARCHAR(16), FAVEFOOD VARCHAR(20))
AS
BEGIN
    INSERT INTO WIZZO(WIZTYPE, FAVEFOOD)
    VALUES ( :WIZTYPE, :FAVEFOOD) ;
END ^^
SET TERM ;^^
...
  
```

SET TRANSACTION

The command is used in some DSQL environments, including *isql*, to start a new transaction without using the specialised API call. The full range of parameters is available.

SET TRANSACTION [options];

Options

SET TRANSACTION

read write read only	Transactions are READ WRITE by default. Use READ ONLY to specify a read-only transaction.
wait no wait	By default, a transaction will wait for access to a record if it encounters a lock conflict with another transaction. Specify NO WAIT to have the lock conflict raised immediately.
ignore limbo	From v.2.0, causes the records created by limbo transactions to be ignored. In general, it has little purpose in isql, being a facility intended for use by the <i>gfx</i> utility for resetting databases affected by an uncompleted two-phase transaction.
isolation level xxx	Can be READ COMMITTED, SNAPSHOT (the default) or SNAPSHOT TABLE STABILITY.
lock timeout n	From v.2.0, a timeout period of n seconds can be specified as the length of time the transaction is to wait for access to a record.
no auto undo	From v.2.0, prevents the transaction from building an undo log that would be used to undo work in the event of its being rolled back.
[no] record version	Only if isolation level is READ COMMITTED, specifies whether the transaction is to be allowed to prevail over other transactions that already have pending changes to a record (RECORD VERSION) or is to experience a conflict under those conditions (NO RECORD VERSION, the default).
reserving <list of tables>	Reserves locks on the tables in the comma-separated list

Example SET TRANSACTION WAIT SNAPSHOT NO AUTO UNDO LOCK TIMEOUT 10

For more information about transactions and their configuration, refer to Part V.

SET TIME

Specifies whether to display the time portion of a DATE value (Dialect 1 only).

SQL> SET TIME [ON|OFF];

Options

SET TIME ON	Toggles on time portion display in Dialect 1 DATE value. Default.
SET TIME OFF	Toggles off time portion display in Dialect 1 DATE value

```
Example      SQL> SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;
              HIRE_DATE
              -----
              16-MAY-2014
              ...
              SQL>SET TIME ON ;
              SQL> SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;
              HIRE_DATE
              -----
              16-MAY-2014 18:20:00
```

SET WARNINGS | SET WNG

Specifies whether warnings are to be output..

```
SQL> SET WARNINGS [ON |OFF ];
```

Options

- SET WARNINGS ON Switches on display of warnings if it was switched off, or if the session was started with the -nowarnings option.
- SET WARNINGS OFF Switches off display of warnings if it is currently on.
- SET WNG Alternative on/off toggle

Exiting an interactive *isql* session

To exit the *isql* utility and roll back all uncommitted work, enter:

```
SQL> QUIT;
```

To exit the *isql* utility and commit all work, enter:

```
SQL> EXIT;
```

Command Mode *isql*

Although *isql* has some handy interactive features, it is not restricted to being run in its own shell. Many of the interactive commands are available as command-line switches, making it potentially a powerful tool for automating and controlling changes. Some *isql* utilities—such as metadata extraction—are available only from the command shell. Operations on input and output files need not be interactive: in fact, calling *isql* with the -i[nput] and -o[utput] switches will not invoke the interactive shell.

Commands execute and, upon completion, return control automatically to the command shell. Calls to *isql* can also be batched inside shell scripts, cron scripts or batch files.

Operating *isql* in command mode

Open a command shell and change to the Firebird /bin directory of your Firebird server or client installation. Use the following syntax pattern for *isql* calls:

```
isql [options] [database_name] [-u[ser] <user-name> -pas[sword] <password>]
```

The <options> are noted in Table 24.1 below.

For SYSDBA operations, you can set the operating system variables ISC_USER and ISC_PASSWORD and avoid the need to enter them on commands. For non-SYSDBA activities, you will always need the owner credentials for operations on the database and/or objects.

The default statement terminator is the semicolon. You can change it to any character or group of characters with a command-line option (`-t[terminator]`).



If your command is connecting to a database for the purpose of running a script and its default character set is not NONE, you need to include the SET NAMES command in your script.

You can set the SQL dialect from the command line when invoking *isql*:

```
isql -s n ;
```

where *n* is 1, 2, or 3.

Quick help

If you call *isql* from the command line with an unknown switch—a question mark, for example—it will display a quick reference list of all its options and arguments:

```
opt/firebird/bin] isql -?
Unknown switch: ?
usage: isql [options] [<database>]
-a(all) extract metadata incl. legacy non-SQL tables
-b(ail) bail on errors (set bail on)
... etc.
```

Command-line switches

Only the initial characters in an option are required. You can also type any portion of the text shown here in square brackets, including the full option name. For example, specifying `-n`, `-no`, or `-noauto` have the same effect.

Table 24.1 Switches for isql command-line options

Option	Description
<code>-a</code>	Extracts all DDL for the named database, including non-DDL statements
<code>-b[ail]</code>	Since v.2.0, instructs <i>isql</i> to bail out on error, returning an error code (a non-zero integer) to the operating system. To understand how it works, please refer to the notes for SET BAIL .
<code>-d[atabase] name</code>	Used with the <code>-x</code> (extract) switch; changes the CREATE DATABASE statement that is extracted to a file. It should be a fully-qualified file path. Without the <code>-d</code> switch, CREATE DATABASE appears as a C-style comment, using the database name specified in the command line.
<code>-c[ache]</code>	Set number of cache pages used for this connection to the database. You can use this switch to override the cache size that is currently set for the database, for the duration.
<code>-e[cho]</code>	Displays (echoes) each statement before executing it

Option	Description
<code>-ex[tract]</code>	Extracts DDL for the named database; displays DDL to the screen unless output is redirected to a file
<code>-i[nput] <i>file</i></code>	Reads commands from an input file instead of from keyboard input. The file argument must be a fully qualified file path. Input files can contain -input commands that call other files, enabling execution to branch and then return. <i>isql</i> commits work in the current file before opening the next.
<code>-m[erge_stderr]</code>	Merges <code>stderr</code> output with <code>stdout</code> . Useful for capturing output and errors to a single file when running <i>isql</i> in a shell script or batch file. It has no counterpart in interactive <i>isql</i> .
<code>-m2</code>	Since v2.0, sends the statistics and plans to the same output file as the other output, when an output file is in effect. It has no counterpart in interactive <i>isql</i> .
<code>-n[oa]utocommit</code>	Turns off automatic committing of DDL statements. By default, DDL statements are committed automatically in a separate transaction
<code>-now[arnings]</code>	Displays warning messages if and only if an error occurs (by default, <i>isql</i> displays any message returned in a status vector, even if no error occurred).
<code>-o[utput] <i>file</i></code>	Writes results to an output file instead of to standard output. The file argument must be a fully qualified file path.
<code>-pas[sword] <i>password</i></code>	Used with <code>-user</code> to specify a password when connecting to a remote server or when required for a local operation. For access, both password and user must represent valid entries in the security database on the server.
<code>-pag[elength] <i>n</i></code>	In query output, prints column headers every <i>n</i> lines instead of the default 20
<code>-q[uiet]</code>	Suppresses the “Use CONNECT or CREATE DATABASE...” message when the database path is not supplied on the command line.
<code>-r[ole] <i>rolename</i></code>	Passes role <i>rolename</i> , case-insensitively, in upper case, along with user credentials on connection to the database
<code>-r2 <i>rolename</i></code>	From v.2.0, passes a case-sensitive <i>rolename</i> (exactly as typed) along with user credentials on connection to the database
<code>-s[ql_dialect] <i>n</i></code>	Interprets subsequent commands as dialect <i>n</i> until end of session or until dialect is changed by a SET SQL DIALECT statement. Refer to the topic Setting dialect in <i>isql</i> , earlier in this chapter.
<code>-t[erminator] <i>x</i></code>	Changes the end-of-statement symbol from the default semicolon (;) to <i>x</i> , where <i>x</i> is a single character or any sequence of characters
<code>-u[ser] <i>user</i></code>	Used with <code>-password</code> ; specifies a user name when connecting to a remote server. For access, both password and user must represent a valid entry in the security database
<code>-x</code>	Same as <code>-extract</code>
<code>-z</code>	Displays the software versions of <i>isql</i> , server and client.

Extracting Metadata

From the command line, you can use the `-extract` option to output the DDL statements that define the metadata for a database. You can use the resulting text file to:

- Examine the current state of a database's system tables before planning alterations. This is especially useful when the database has changed significantly since its creation.
- Create a database with schema definitions that are identical to the extracted database.
- Open in your text editor to make changes to the database definition or create a new database source file.

All reserved words and objects are extracted into the file in uppercase unless the local language uses a character set that has no upper case. The output script is created with a COMMIT statement following each set of commands, so that tables can be referenced in subsequent definitions. The output file includes the name of the object and the owner, if one is defined.

The optional `–output` flag reroutes output to a named file.

Syntax pattern

```
isql [[-extract | -x][ -a] [[-output | -o] outputfile]] database
```

The `–x` option can be used as an abbreviation for `–extract`. The `–a` flag directs *isql* to extract all database objects.

The output file specification, *outputfile*, must be a fully-qualified path placed after the `–output` switch directly. The path and name, or the alias, of the database being extracted can be at the end of the command.



The `–extract` function is not always as smart as it should be about dependencies. It is sometimes necessary to edit the output file to rectify creation order.

Using isql `–extract`

The following statement extracts the SQL schema from the database `employee.fdb` to a schema script file called `employee.sql`:

```
isql -extract -output /data/scripts/employee.sql /data/employee.fdb
```

This command is equivalent:

```
isql -x -output /data/scripts/employee.sql /data/employee.fdb
```

Objects and items not extracted

- System tables and views, system triggers
- External function and BLOB filter code—it's not part of the database.
- Object ownership attributes

Using isql `–a`

The `–(e)x(tract)` option extracts metadata for SQL objects only. If you wish to extract a schema script that includes declarations—such as `DECLARE EXTERNAL FUNCTION` and `DECLARE FILTER`—use the `–a` option.

For example, to extract DDL statements, external function declarations and BLOB filter declarations from database `employee.fdb` and store in the file `employee.sql`, enter:

```
isql -a -output /data/scripts/employee.sql /data/employee.fdb
```

Creating and Running Scripts

With Firebird, as with all true SQL database management systems, at some level you build your database and its objects—the metadata or schema of a database—using statements from a specialised subset of SQL statements known as Data Definition Language, or DDL. Even highly-featured graphical tools ultimately process your design in DDL statements.

Running a batch of DDL statements that has been scripted into in a text file is much closer to the “coal face”. A script, or a set of scripts, can be processed by calling *isql* directly at the command line with an *-input* argument amongst its switches.

About Firebird scripts

A script for creating and altering database objects is sometimes referred to as a *data definition file* or, more commonly, a *DDL script*. A DDL script can contain certain *isql* statements, specifically some of the SET <parameter> commands. COMMIT is also a valid statement in a script.

Other scripts can be written for inserting basic or “control” data into tables, updating columns, performing data conversions and other maintenance tasks involving data manipulation. These are known as DML scripts (for Data Manipulation Language).

DDL and DML commands can be mixed in a script but, as a rule of thumb, good reason exist not to do it. Script processing allows “chaining” of scripts, linking one script file to another by means of the *isql* INPUT <filespec> statement. DDL and DML statements should be separated each into their own sequenced script files.

Script statements are executed in strict order. Use of the SET AUTODDL command enables you to control where statements or blocks of statements will be committed. It is also an option to defer committing the contents of a script until the entire script has been processed.

Why use scripts?

It is very good practice to use DDL scripts to create your database and its objects. Some of the reasons include

- Self-documentation. A script is a plain text file, easily handled in any development system, both for updating and reference. Scripts can—and should—include detailed comment text. Alterations to metadata can be signed and dated manually.
- Control of database development and change. Scripting all database definitions allows schema creation to be integrated closely with design tasks and code review cycles.
- Repeatable and traceable creation of metadata. A completely reconstructable schema is a requirement in the quality assurance and disaster recovery systems of many organizations.
- Orderly construction and reconstruction of database metadata. Experienced Firebird programmers often create a set of DDL scripts, designed to run and commit in a specific order, to make debugging easy and ensure that objects will exist when later, dependent objects refer to them.

What is in a DDL script?

A DDL script consists of one or more SQL statements to CREATE, ALTER, or DROP a database or any other object. It can include data manipulation language (DML) statements, although it is recommended to keep DDL and DML statements in separate scripts.



It is quite common to include (INPUT) one or more scripts amongst a chain of DDL scripts, containing INSERT statements to populate some tables with static control data. You might, for example, post statements to insert the initial rows in a table of Account Types.

The complex procedure language (PSQL) statements defining stored procedures and triggers can also be included. PSQL blocks get special treatment in scripts with regard to statement terminator symbols (see below).

Comments

A script can also contain comments, in two varieties.

Block comments

Block comments in DDL scripts use the C convention:

```
/* This comment can span multiple
   lines in a script */
```

A block comment can occur on the same line as a SQL statement or *isql* command and can be of any length, as long as it is preceded by */** and followed by **/*.

In-line comments

The */* ... */* style of comment can also be embedded inside a statement as an in-line comment, e.g.

```
CREATE TABLE USERS1 (
  USER_NAME VARCHAR( 128 ) /* security user name */
  , GROUP_NAME VARCHAR( 128 ) /* not used on Windows */
  , PASSWD VARCHAR( 32 ) /* will be stored encrypted */
  , FIRST_NAME VARCHAR( 96 ) /* Defaulted */
  , MIDDLE_NAME VARCHAR( 96 ) /* Defaulted */
  , LAST_NAME VARCHAR( 96 ) /* Defaulted */
  , FULL_NAME VARCHAR( 290 ) /* Computed */
) ;
```

One-line comments

In Firebird scripts you can use an alternative convention for commenting a single line: the double hyphen, e.g.,

```
-- don't change this!
```

The *--* commenting convention can be used anywhere on a line, to “comment out” everything from the marker to the end of the current line. For example:

```
CREATE TABLE MET_REPORT (ID BIGINT NOT NULL, -- COMMENT VARCHAR(40), invisible
  WEATHER_CONDITIONS BLOB SUB_TYPE TEXT, LAST_REPORT TIMESTAMP);
```

v.1.0.x This double-hyphen style of comment is not implemented in the old v.1.0 series.

isql statements

The *isql* commands SET AUTODDL, SET SQL DIALECT, SET TERM, SET BAIL and INPUT are valid statements in a Firebird script.

Terminator symbols

All statements that are to be executed in the script must end with a terminator symbol.

- The default symbol is the semicolon (;).
- The default terminator can be overridden for all statements except procedure language statements (PSQL) by issuing a SET TERM command in the script.

Terminators and procedure language (PSQL)

For its internal statements, PSQL does not permit any terminator other than the default semi-colon (;). This restriction is necessary because CREATE PROCEDURE, RECREATE PROCEDURE, ALTER PROCEDURE, CREATE TRIGGER and ALTER TRIGGER, together with their subsequent PSQL statements, are complex statements in their own right—statements within a statement. The compiler needs to see semi-colons in order to recognize each distinct PSQL statement.

Thus, in scripts, it is necessary to override the terminator being used for script commands before commencing to issue the PSQL statements for a stored procedure or a trigger. After the last END statement of the procedure source has been terminated, the terminator should be reset to the default using another SET TERM statement.

Examples ...

```
CREATE GENERATOR GEN_MY_GEN ;
SET TERM ^^;
CREATE TRIGGER BI_TABLEA_0 FOR TABLEA
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
    IF (NEW.PK IS NOT NULL) THEN
        NEW.PK = GEN_ID(GEN_MY_GEN, 1);
    END ^^
SET TERM ;^^
```

...

Any string may be used as an alternative terminator, for example:

```
...
SET TERM @!#;
CREATE PROCEDURE...
AS
BEGIN
    ... ;
    ... ;
END @!#
SET TERM ;@!#
/**/
COMMIT;
/**/
SET TERM +;
CREATE PROCEDURE...
AS
BEGIN
    ... ;
```

```

... ;
END +
SET TERM ;+
/**/
COMMIT;

```

The SQL statement silently fails if significant text follows the terminator character on the same line. Whitespace and comments can safely follow the terminator, but other statements can not.

For example, in the following sequence, the COMMIT statement will not be executed:

```
ALTER TABLE ATABLE ADD F2 INTEGER; COMMIT;
```

whereas this one is fine:

```
ALTER TABLE ATABLE ADD F2 INTEGER;      /* counter for beans */
COMMIT;
```

Managing errors

Several options can help with pinpointing where an error occurs. The `-o[utput]` switch with its *file* argument enables *isql* to pass output to a log, while `-e[cho]` is the means of passing the commands and statements to your log and `-m[erge]` includes the error messages.

The *isql* script processor counts lines and can deliver the line number of the offending line and, if applicable, the position of the error in that line. What constitutes a “line”, and therefore its number, can be variable, being governed by the underlying I/O layer. By and large, the start of a new line is recognised as the first non-whitespace character after a hard line-break, although *isql* itself will force a break if a line exceeds the limit of 32,767 bytes.

The line counting works correctly with nested scripts, using a separate line counter for each input file.

The position of the beginning of a keyword or phrase within the line that throws a DSQL parsing error is referred to as the “column”. Thus, a message that reports “an error at line *m* column *n*” is referring to the *n*th character in the line. White space characters are counted.

Normal behaviour is for the script to continue executing whether errors occur or not. If you are echoing all the output, you can certainly discover what errors occurred but not always where they occurred.

Including the `-b[ail]` option causes script execution to stop as soon as an error occurs, which means you can finger your errors explicitly.



*You can alternatively build your script to include a **SET BAIL** command and manage the bail-out behaviour internally.*

Basic steps

The basic steps for using script files are:

1 Create the script file

Use any suitable plain text editor. At the learning stage, you might wish to follow each DDL statement with a COMMIT statement, to ensure that an object will be visible to subsequent statements. As you become more practised, you will learn to commit statements in blocks, employing SET AUTODDL ON and SET AUTODDL OFF as a means of controlling interdependencies and testing/debugging scripts.



Ensure that every script ends with a carriage return and at least one blank line.

2 Execute the script

Use the INPUT command in an *isql* session or the Execute button (or equivalent) in your database management tool.

isql on POSIX:

```
SQL> INPUT /data/scripts/myscript.sql;
```

isql on Win32:

```
SQL> INPUT d:\data\scripts\myscript.sql;
```

3 View output and confirm database changes

External tools and Firebird *isql* versions vary in the information they return when a script trips up on a bad command and what they leave to you to put right. In an *isql* script, bear in mind that scripts containing SET AUTODDL commands and COMMIT statement will have accomplished work that cannot be rolled back.

Creating a script file

You can create DDL scripts in several ways, including:

- in an interactive *isql* session using the OUTPUT command to pass a series of DDL statements to a file
- in a plain ASCII text editor that formats line breaks according to the rules of the operating system shell in which the DDL script will be executed
- using one of the many specialized script editor tools that are available in third-party database design or administration tools for Firebird.
- using a CASE tool which can output DDL scripts according to the Firebird conventions

You can use any text editor to create a SQL script file, as long as the final file format is plain text (ASCII) and has lines terminated according to the rules of the operation system you intend to run the script on:

- on Windows the line terminator is carriage return + line feed (ASCII 13 followed by ASCII 10)
- on Linux/UNIX it is the solitary line feed or “new line” character (ASCII 10)
- on Mac OSX it is new line (ASCII 10) and on native Macintosh it is carriage return (ASCII 13)



*Don't overlook the *isql*'s metadata extract tools—they may prove useful for providing you with templates for your own scripts.*

Some editing tools provide the capability to save in different text formats. It may prove useful to be able to save Linux-compatible scripts on a Windows machine, for example. However, take care that you use an editor that is capable of saving all characters in the character sets that will be used in the database. Also be sure to avoid an editor that saves text with any enhancements whatsoever.

A complete schema script file must begin with either a CREATE DATABASE statement or, if the database already exists, a CONNECT statement (including username and password in single quotes) that specifies the database on which the script file is to operate. The CONNECT or CREATE keyword must be followed by a complete, absolute

database file name and directory path in single quotes or, if appropriate, an alias that has already been set up in `aliases.conf`.

// Do not use database aliases in scripts that create a database.

For example:

```
SET SQL DIALECT 3 ;
CREATE DATABASE 'd:\databases\MyDatabase.fdb'
    PAGE_SIZE 8192
    DEFAULT CHARACTER SET ISO8859_1
    USER 'SYSDBA' PASSWORD 'masterkey';
```

or

```
CONNECT 'd:\databases\MyDatabase.gdb' USER 'SYSDBA' PASSWORD 'masterkey' ;
```

Committing work in scripts

Scripts do not run atomically. It is important to be aware of the effects of including commands and statements in your scripts that cause work to be committed. There is no hard and fast rule that says it is either “good” or “bad” to run scripts that cannot be reversed. Often, for example, you will have dependencies between DDL objects that will require committing in strict order. In script sequences where objects are created in one script and populated in another, errors will abound if commits are not carefully attended to.

DDL statements

Statements in DDL scripts can be committed in one or more of the following ways:

- by including COMMIT statements at appropriate points in the script to ensure that new database objects are available to all subsequent statements that depend on them.
- by including this statement at the beginning of the script:

```
SET AUTODDL ON ;
```

To turn off automatic commit of DDL in an isql script, use

```
SET AUTODDL OFF ;
```

The ON and OFF keywords are optional: the abbreviation SET AUTO can be used as a two-way switch. For clarity of self-documentation, it is recommended that you use SET AUTODDL with the explicit ON and OFF keywords.

Autocommit in isql

If you are running your script in *isql*, changes to the database from data definition (DDL) statements—for example, CREATE and ALTER statements—are automatically committed by default. This means that other users of the database see changes as soon as each DDL statement is executed.

Some scripting tools deliberately turn off this autocommitting behavior when running scripts, since it can make debugging difficult. Make sure you understand the behaviour of any third-party tool you use for scripts.

DML statements

Changes made to the database by data manipulation (DML) statements—INSERT, UPDATE and DELETE—are not permanent until they are committed. Explicitly include COMMIT statements in your script to commit DML changes.

To undo all database changes since the last COMMIT, use ROLLBACK. Committed changes cannot be rolled back.

Executing scripts

DDL scripts can be executed in an interactive isql session using the INPUT command, as described in the summary above. Many of the third-party tools have the ability to execute and even to intelligently debug scripts in a GUI environment.

Chaining scripts

A stable suite of scripts can be “chained” together using the isql INPUT statement as the last statement in the preceding script. For example, to chain a script named `leisurestore_02.sql` to the end of one named `leisurestore_01.sql`, end the script this way:

```
...
COMMIT;
-- chain to CREATE TABLE statements
INPUT 'd:\scripts\leisurestore_02.sql' ;
-- don't forget the carriage return!
```

The master script approach

The INPUT statement is not restricted to being the last in a script. Hence, another useful approach to maintaining script suites is to have one “master” script that inputs each of the subsidiary scripts in order. It has the benefit of making it easy to maintain large suites and you can include comments to indicate to others the contents of each input script.

Managing your schema scripts

Keeping a well-organised suite of scripts that precisely reflects the up-to-date state of your metadata is a valuable practice that admirably satisfies the most rigorous quality assurance system. The use of ample commentary within scripts is highly recommended, as is archiving all script versions in a version control system.

Disaster recovery

The most obvious purpose of such a practice is to provide a “fallback of last resort” for disaster recovery. If worst comes to worst—a database is ruined and backups are lost—metadata can be reconstructed from scripts. Surviving data from an otherwise unrecoverable database can be reconstituted by experts and pumped back.



A script alone will not save you. It is strongly recommended that a complete “recovery from basic ingredients” plan be tested and documented before a rollout.

Development control

Normally, more than one developer will work on the development of a database during its life cycle. Developers notoriously abhor writing system documentation! Keeping an annotated script record of every database change—including those applied interactively using *isql* or a third-party tool—is a painless and secure solution that works for everybody.

Metadata extraction

Several admin tools for Firebird, including *isql*, are capable of extracting metadata from a database for saving as a script file. For *isql* instructions, refer to the topic [Extracting Metadata](#) in this chapter. While metadata extraction is a handy adjunct to your scripting, there are good reasons to treat these tools as “helpers” and make a point of maintaining your main schema scripts manually:

- Metadata extraction tools generate only the current metadata. There is no history of changes—dates, reasons or authors.
- Some tools, including some versions of *isql*, are known to generate metadata in the wrong sequence for dependency, making the scripts useless for regenerating the database without editing. Such a task is between tedious and impossible, depending on how well the repairer knows the metadata.
- Even moderately-sized databases may have an enormous number of objects, especially where the system design makes intensive use of embedded code modules. Very large scripts are prone to failure due to various execution or resource limits. Large, poorly organized scripts are also confusing and annoying to work with as documentation.

Manual scripting

The author strongly advocates maintaining fully annotated schema scripts manually and splitting the mass into separate files. The following sample suite of scripts records and regenerates a database named `leisurestore.fdb`:

Table 24.2 Sample suite of schema scripts

File	Contents
<code>leisurestore_01.sql</code>	CREATE DATABASE statement, CREATE DOMAIN, CREATE GENERATOR and CREATE EXCEPTION definitions
<code>leisurestore_02.sql</code>	All CREATE TABLE statements, including UNIQUE constraints, ALTER TABLE statements adding all primary keys as named PRIMARY KEY constraints
<code>leisurestore_03.sql</code>	ALTER TABLE statements adding FOREIGN KEY constraints
<code>leisurestore_04.sql</code>	CREATE INDEX statements
<code>leisurestore_05.sql</code>	CREATE TRIGGER statements
<code>leisurestore_06.sql</code>	CREATE PROCEDURE statements
<code>leisurestore_07.sql</code>	DML script that inserts rows into static (control) tables
<code>leisurestore_08.sql</code>	GRANT statements (security script)
<code>leisurestore_09.sql</code>	Recent changes, in correct sequence for dependency
<code>leisurestore_10.sql</code>	QA scripts (test data)



The
Firebird Book
A Reference for Database Developers

SECOND EDITION

PART V



Transactions

CHAPTER 25

OVERVIEW OF FIREBIRD TRANSACTIONS

In a client/server database such as Firebird, client applications never touch the data that are physically stored in the pages of the database. Instead, the client applications conduct conversations with the database management system—“the server”—by packaging requests and responses inside transactions. This topic visits some of the key concepts and issues of transaction management in Firebird.

- A data management system that is being updated by multiple users concurrently is vulnerable to a number of data integrity problems if work is allowed to overlap without any control. In summary, they are:
- Lost updates, that would occur when two users have the same view of a set and one performs changes, closely followed by the other, who overwrites the first user's work.
- Dirty reads, that permit one user to see the changes that another user is in the process of doing, with no guarantee that the other user's changes are final.
- Non-reproducible reads, that allow one user to select rows continually while other users are updating and deleting. Whether this is a problem depends on the circumstances. For example, a month-end financials process or a snapshot inventory audit will be skewed under these conditions, whereas a ticketing application needs to keep all users' views synchronized to avoid double booking.
- Phantom rows, which arise when one user can select some but not all new rows written by another user. Again, this may be acceptable in some situations but will skew the outcomes of some processes.
- Interleaved transactions. These can arise when changes in one row by one user affect the data in other rows in the same table or in other tables being accessed by other users. They are usually timing-related, occurring when there is no way to control or predict the sequence in which users will perform the changes.

To solve these problems, Firebird applies a management model that isolates every task inside a unique context that prescribes the outcome if work within that task would be at risk of overlapping work being performed by other tasks. The state of the database can not change if there is any conflict.



Firebird does not permit dirty reads. In certain conditions, by design, it allows non-reproducible reads.

The ACID Properties

It is now more than 20 years since two researchers, Theo Haërder and Andreas Reuter, published a review paper describing requirements for maintaining the integrity of a database in a parallel updating environment. They distilled the requirements into four precepts defined as *atomicity*, *consistency*, *isolation* and *durability*—abbreviated in the acronym ACID¹. Over the succeeding years, the ACID concept evolved as a benchmark for the implementation of transactions in database systems.

Atomicity

A transaction—also known as a *unit of work*—is described as consisting of a collection of data-transforming actions. To be “atomic”, the transaction must be implemented in such a way that it provides the “all-or-nothing illusion that either all of these operations are performed or none of them is performed”². The transaction either completes as a whole (commits) or aborts (rolls back).

Consistency

Transactions are assumed to perform correct transformations of the abstract system state—that is, the database must be left in a consistent state after a transaction completes, regardless of whether it commits or rolls back. The transaction concept assumes that programmers have mechanisms available to them for declaring points of consistency and validating them. In SQL, the standards provide for triggers, referential integrity constraints and check constraints to implement those mechanisms at the server.

Isolation

While a transaction is updating shared data, the system must give each transaction the illusion that it is running in isolation: it should appear that all other transactions ran either before it began or after it committed. While data are in transition from a starting consistent state to a final consistent state, they may be inconsistent with database state, but no data may be exposed to other transactions until the changes are committed.

The next chapter explores the three *levels of isolation* for transactions that Firebird implements, along with the options available to respond to conflicts and prevent the work of one transaction from interfering with the work of another.

1. Theo Haërder and Andreas Reuter, “*Principles of Transaction-Oriented Database Recovery*,” ACM Computing Surveys 15(4) (1983): 287–317.

2. Andreas Reuter and Jim Gray, “*Transaction Processing Concepts and Techniques*” (San Francisco, CA: Morgan Kaufmann, 1993).

Durability

Once a transaction commits, its updates must be “durable”—that is, the new state of all objects made visible to other transactions by the commit will be preserved and irreversible, regardless of events such as hardware failure or software crashing.

Context of a Transaction

A transaction encompasses a complete conversation between a client and the server. Each transaction has a unique context that causes it to be isolated it from all other transactions in a specific way. The rules for the transaction’s context are specified by the client application program, by passing transaction *parameters*. A transaction starts when a client signals the beginning of the transaction and receives a transaction handle back from the server. It remains active until the client either commits the work (if it can) or rolls it back.

One transaction, many requests

In Firebird, every operation requested by a client must occur in the context of an active transaction. One transaction can entail one or many client requests and server responses within its bounds. A single transaction can span more than a single database, encompassing reads from and writes to multiple databases in the course of a task. Tasks that create databases or change their physical structure—single or batched DDL statements—entail transactions too.

The role of the client

Clients initiate all transactions. Once a transaction is under way, a client is responsible for submitting requests to read and write and also for completing—committing or rolling back—every transaction it starts.

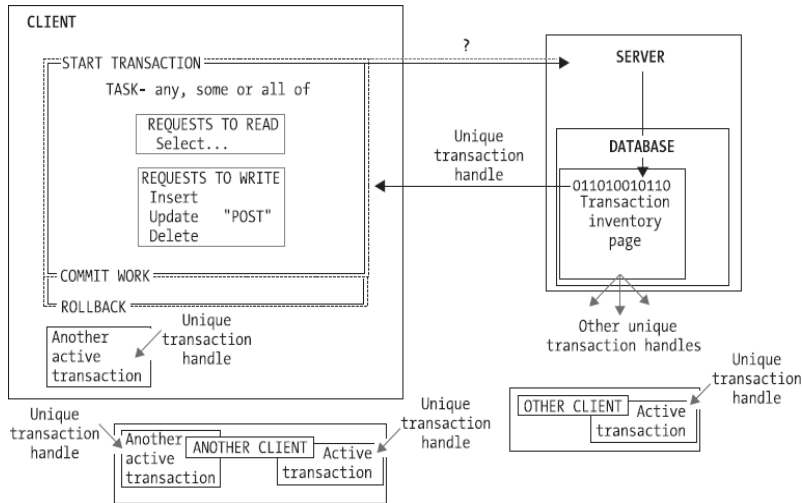
A single client connection can have multiple transactions active.

The role of the server

It is the job of the server to account for each transaction uniquely and keep each transaction’s view of database state consistent with its context. It has to manage all requests to change the database so that conflicts are handled appropriately and any risk of breaking the consistency of database state is pre-empted.

Figure 25.1 broadly illustrates a typical read-write transaction context from start to finish. It shows how the server manages many concurrent transactions independently of all others, even if the other transactions are active in the same client application.

Figure 25.1 Transaction context and independence



In this illustration, the application may use the transaction to select (read) sets of rows, any of which may be chosen by the user for updating or deleting. In the same transaction, new rows might be inserted. Update, insert or delete requests are “posted” to the server, one by one, as the user performs her task.

Atomicity of the model

The capability for a client application to issue multiple, reversible requests inside a single Firebird transaction has obvious advantages for tasks that need to execute a single statement many times with a variety of input parameters. It inherently provides atomicity for tasks where a group of statements must be executed and, finally, either committed as a single job or rolled back entirely if an exception occurs.

Transactions and the MGA

MGA—an acronym for multi-generational architecture—refers to the architectural model underlying state management in a Firebird database.

In the MGA model, each row stored in database holds the unique transaction ID of the transaction that writes it. If another transaction posts a change to the row, the server writes a new version of the row to disk, with the new transaction ID, and converts an image of the older version into reference (known as the delta) to this new version. The server now holds two “generations” of the same row.

Post vs COMMIT

The term “post” has probably been borrowed from older desktop accounting software, as an analogy to the posting of journals in accounting systems. The analogy is useful for distinguishing the two separate operations of writing a reversible change to the database (by executing a statement) and committing all of the changes executed by one or several statements. Posted changes are invisible beyond the boundaries of their transaction

context and can be rolled back; committed changes are permanent and become visible to transactions that are subsequently started or updated.

If any conflict occurs upon posting, the server returns an exception message to the client and the commit fails. The application must then deal with the conflict according to its kind. The solution to an update conflict is often to roll back the transaction, causing all of the work to be “undone” as an atomic unit. If there is no conflict, the application can proceed to commit the work when it is ready.



*Calls from the client to **COMMIT** or to **ROLLBACK** are the only possible ways to end a transaction. A failure to commit does not cause the server to roll back the transaction.*

Rollback

Rollback never fails. It will undo any changes that were requested during the transaction: the change that caused the exception as well as any that would have succeeded had the exception not occurred.

Some rolled-back transactions do not leave row images on the disk. For example, any rollbacks instigated by the server to undo work performed by triggers simply vanish; and the record images created by inserts are normally expunged during rollback, with reference to an auto-undo log that is maintained for inserts. If a single transaction inserts a huge number of records, the auto-undo log is abandoned, with the result that a rollback will leave the images on disk. Other conditions that may cause inserted record images to remain on disk include server crashes during the operation or use of a transaction parameter that explicitly requests “no auto-undo”.

Row locking

Under MGA, the existence of a pending new version of a row has the effect of locking it. In most conditions, the existence of a committed newer version blocks a request to update or delete the row as well—a *locking conflict*.

On receiving a request to update or delete, the server inspects the states of any transactions that own newer versions of the row. If the newest of those owner transactions is active or has been committed, the server responds to the requesting transaction according to the context—isolation level and lock resolution parameters—of the requesting transaction.

If the newest version’s transaction is active, the requesting transaction will, by default, wait (parameter *WAIT*) for it to be completed (committed or rolled back) and then the server will allow it to proceed. However, if *NO WAIT* was specified, it returns a conflict exception to the requesting transaction.

If the newest version’s transaction is committed and the requesting transaction is in *SNAPSHOT* (i.e. concurrency) isolation, the server refuses the request and reports a lock conflict. If the transaction is in *READ COMMITTED* isolation, with the default *RECORD_VERSION* setting, the server allows the request and writes a new record version on behalf of the transaction.



*Other conditions are possible when the requesting transaction is in **READ COMMITTED**. The outcome of any transaction request may also be affected by unusual, pre-emptive conditions in the context of the transaction that owns the pending change. For more information about these variations, refer to the next chapter, where the transaction parameters are explored in detail.*

Optimistic locking

Firebird does not use conventional two-phase locking at all for the most usual transaction contexts. Hence, all normal locking is at row-level and is said to be *optimistic*—each row is available to all read-write transactions until a writer creates a newer version of it.

On committing

Upon a successful commit, the old record version referenced by the delta image becomes an obsolete record.

- If the operation was an update, the new image becomes the latest committed version and the original image of the record, with the ID of the transaction that last updated it, is made available for garbage collection.
- If the operation was a deletion, a “stump” replaces the obsolete record. A sweep or a backup clears this stump and releases the physical space on disk that was occupied by the deleted row.

For information about sweeping and garbage collection, refer to the section *Management Tools* in Chapter 35, *Configuring and Managing Databases*, especially the topics *Garbage collection* and *Sweeping*. See also Appendix XI, *Healthcare for Databases*.

In summary, under normal conditions:

- any transaction can read any row that was committed before it started
- any read-write transaction can request to update or delete a row.
- a request (post) will usually succeed if no other read-write transaction has already posted or committed a change to a newer version of the record. Read-committed transactions are usually allowed to post changes overwriting versions committed by newer transactions.
- if a post succeeds, the transaction has a “lock” on the row. Other readers can still read the latest committed version but none will succeed in posting an update or delete statement for that row

Table-level locks

A transaction can be configured to lock whole tables. There are two acceptable ways to do this in DSQL: by isolating the transaction in *SNAPSHOT TABLE STABILITY* (a.k.a consistency) mode, forced repeatable read) or by *table reservation*. It should be emphasized that these configurations are for when unusually pre-emptive conditions are required. They are not recommended in Firebird for everyday use.

Pessimistic locking

An unacceptable way to impose a table-level lock is to apply a statement-level *pessimistic lock* that affects the whole table. A statement such as the following can do that:

```
SELECT * FROM ATABLE
[FOR UPDATE] WITH LOCK;
```

It is not, strictly speaking, an issue of transaction configuration. However, the configuration of the transaction that owns sets retrieved with this explicit pessimistic lock is important. The issue of pessimistic locking is discussed in Chapter 27, *Programming with Transactions*.

Inserts

There are no deltas or locks for inserts. If another transaction has not pre-empted inserts by a table-level lock, an insert will always succeed if it does not violate any constraints or validation checks.

Transaction “Aging” and Statistics

When a client process secures a transaction handle, it acquires a unique internal identifier for the transaction and stores it on a transaction inventory page (TIP).

Transaction ID and age

Transaction IDs (TIDs) are 32-bit integers that are generated in a single-stepping series. A new or freshly-restored database begins the series at 1. Transaction age is determined from the TID: lowest is oldest.

TIDs and their associated state data are stored on *transaction inventory pages* (TIPs). On the database header page, system accounting maintains a set of fields containing TIDs that are of interest to it, viz. the oldest interesting (OIT), the oldest active (OAT) and the number to be used for the next transaction. A “snapshot” TID is also written each time the OAT increases, usually the same TID as the OAT, or close to it—or so you hope. It is a record of the OAT before the latest garbage collection began.

Getting the Transaction ID

The context variable `CURRENT_TRANSACTION`, supported since v.1.5, returns the TID of the transaction that requests it. It can be used in SQL wherever it is appropriate to use an expression. For example, to store the TID into a log table, you could use it like this:

```
INSERT INTO BOOK_OF_LIFE
(TID, COMMENT, SIGNATURE)
VALUES
(CURRENT_TRANSACTION,
 'This has been a great day for transactions',
 CURRENT_USER);
```

It is important to remember that TIDs are cyclic. Because the numbering series will be reset after each restore, they should *never* be used for primary keys or unique constraints in persistent tables.

Transaction ID overflow

The transaction id is a 32-bit integer. If a series hit its 4 Gb limit and rolls over, bad things will happen. When the last transaction ends, the system transaction will not work and metadata updates will be impossible. Garbage collection will stop. User transactions will not start.

At 100 transactions per second, it takes one year, four months, eleven days, two hours, and roughly 30 minutes to roll over the TID.

Backing up and restoring into a fresh database resets the TID. Until recently, a neglected database would have had other trauma before the TID series exhausted itself. Now, with

larger page sizes, larger disks and reduced need to watch the size of database files, the risk of blowing a transaction ID series is more apparent.

“Interesting transactions”

Server and client transaction accounting routines use TIDs to track the states of transactions. Housekeeping routines take the age of transactions into account when deciding which old record versions are “interesting” and which are not. Uninteresting transactions can be flagged for removal. The server remains interested in every transaction that not been hard-committed by a COMMIT statement.

Active, limbo, rolled back and dead transactions are all “interesting”. Transactions that have been committed using COMMIT WITH RETAIN (a.k.a. soft commit or CommitRetaining) remain active until they are “hard-committed” and are thus stay interesting. Conditions can develop in which interesting transactions become “stuck” and progressively inhibit performance.

If neglected, stuck transactions will become the source of serious performance degradation. A stuck OIT will cause the number of transaction inventory pages to grow. The server maintains a bitmapped working table of the transactions stored in the TIPs. The table is copied and written to the database at the start of each transaction. As it becomes bloated by stuck transactions, it uses progressively more memory resources and memory becomes fragmented from constant reallocation of the resources.

Oldest Interesting Transaction (OIT)

The Oldest Interesting Transaction (OIT) is the lowest-numbered transaction in the TIPs (transaction inventory pages) that is in the state of having been involved with records that are in transactions which are as yet not hard-committed.

Oldest Active Transaction (OAT)

The Oldest Active Transaction (OAT) is the lowest-numbered transaction in the TIPs that is active. A transaction is active as long as it is not hard-committed, not rolled back and not *in limbo*.



“Limbo transactions” cannot occur unless the application is running multi-database transactions.

Read-only transactions

A read-only transaction remains active (and interesting in some ways) until it is committed. However, an active read-only transaction that is in the recommended READ COMMITTED isolation level (see Chapter 26) never gets stuck and will not interfere with system housekeeping.



Don't confuse a read-only transaction with a read-write transaction that is in the process of passing output to a user interface from a SELECT statement. Even if the application makes the output set read-only, the underlying SELECT statement inside a read-write transaction can be—and often is—the cause of system slow-down.

Background garbage collection

Record versions from rolled back transactions are marked for purging from the database when they are found in the course of normal data processing—as a rule, if a row is accessed by a statement, any uninteresting old record versions that are eligible for removal will be tagged for collection by the garbage collector (GC).



*In the case of a Superserver installed with “native” or “mixed” GC enabled in the server’s **GCPolicy** configuration parameter, the GC is a worker thread that works in the background. In all other cases, tagging and GC are performed cooperatively: that means that each client process or thread that touches a table will perform any outstanding GC before proceeding with the client request.*

Some corners of the database might not be visited for a long time by any statements, so there is no guarantee that all of the record versions created by a rolled-back transaction will be tagged and, eventually, removed. As long as the record versions remain, the interesting transaction must remain “interesting”, to preserve the state of the database.



A full database scan—by a backup, typically—will perform garbage collection but it can not change the state of transactions. That is the job of the full GC ritual that is performed by a sweep. A newly restored database has no garbage.

Rolled-back transactions

Transactions in the rolled-back state are not garbage-collected. They remain interesting until a database sweep tags them as “committed” and releases them for garbage collection. In systems with low contention, a periodic manual sweep may be all that’s needed to deal with them.

Some systems that are not well-designed to deal with conflicts demonstrate high levels of rollback and tend to accumulate interesting transactions faster than the automatic database housekeeping procedures can cope. Such systems should be subjected to scheduled manual sweeps often if automatic sweeping seems not to manage well.

“Dead” transactions

Transactions are said to be “dead” if they are in an active state but there is no connection context associated with them. Transactions typically “die” in client applications that do not take care to end them before disconnecting users. Dead transactions are also part of the flotsam left behind when client applications crash or network connections are broken abnormally.

The server can not tell the difference between a genuinely active transaction and a dead one. As long as the cleanup following connection timeout is able to proceed and the regular garbage collection kicks in in a timely fashion, dead transactions need not be a problem. The timeout cleanup will roll them back and their garbage will eventually be dealt with normally.

Frequent, large accumulations of dead transactions can be a problem. Because they are rolled back, too many of them remain interesting for too long. In a system where user behavior, frequent crashes, power cuts or network faults generate huge numbers of dead transactions, dead-transaction garbage will become a big problem for performance.

Limbo transactions

Limbo transactions—which are discussed in more detail in Chapter 26—happen in the course of a failed two-phase COMMIT operation across multiple databases. The system recognizes them as a special case for human intervention, since the server itself cannot

determine whether it is safe to either commit them or roll them back without causing inconsistency in a different database.

The only way to resolve a limbo transaction is to run the *gfix* tool over the database with the appropriate switches to achieve the desired outcome. (See *Limbo transactions* in Chapter 35, *Configuring and Managing Databases*.) Resolution changes the state of a limbo transaction to either “committed” or “rolled back”. From that point, it is managed just like any other committed or rolled-back transaction.

Keeping the OIT and the OAT moving

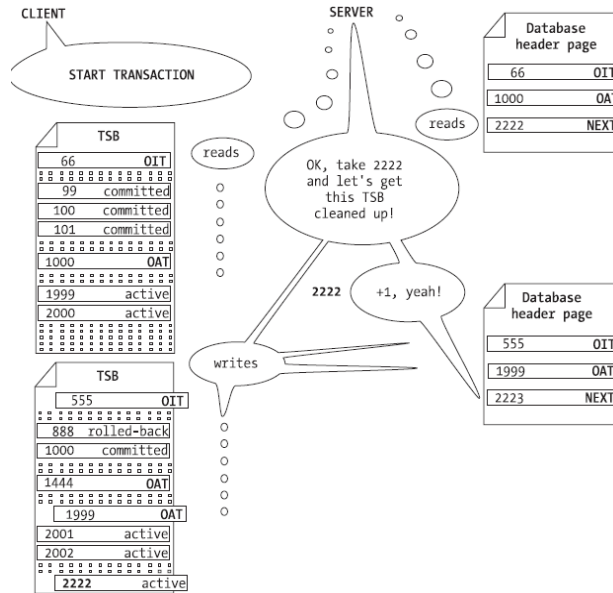
The admonition to “keep the OIT and OAT moving” is a catch-cry around all support solutions where performance is the problem. Time spent understanding the life-cycle of transactions in Firebird's multi-generational architecture will be one of your best investments for future work with Firebird and other open source databases that have MGA.

To begin understanding the interaction between the client processes that “own” the transactions, on the one hand, and the transaction inventory accounting maintained in the database, on the other, it is useful to be acquainted with the way the server and transactions interact in the mechanism.

The transaction state bitmap (TSB)

The internal *transaction state bitmap* (TSB) is a table of transaction IDs and their states, maintained by the server, that is initialized when an initial attachment is made to a database. In logical terms, the TSB tabulates each transaction found in the TIPs that is newer than the OIT. Whilst there are attachments to the database, the server process maintains the TSB dynamically, adding newer TIDs, updating states and sloughing off the TIDs that become uninteresting. It writes updates to the TIPs without reading them again from disk.

Figure 25.2 illustrates the steps. Each time the server processes a request for a transaction handle, it reads the database header page to acquire the TID for the next transaction. The TSB is updated and updates are written to the transaction inventory pages in the database.

Figure 25.2 Interaction of the client/server process and the TSB

“Moving the OIT (and/or the OAT) forward” is Firebird-speak for the evolving increase in the values of the OIT and OAT in the bitmap and in the database header page as older transactions get committed and are eliminated from the TSB and newer transactions become identified as “oldest”. Updating the database header page with the latest OIT, OAT and Next Transaction values is part of this dynamic ecology.

If the new transaction is a *SNAPSHOT* transaction, it will have its own copy of the TSB to maintain a consistent view of database state as it was when it began. A *READ COMMITTED* transaction always refers to the latest global TSB to get access to versions committed after it began.

Conditions for updating OIT and OAT

Each time the server starts another transaction, it tests the state of the TIDs it is holding in the TSB, eliminates those whose state has changed to “committed” and re-evaluates the OIT and OAT values. It compares them with those stored in the database page and, if necessary, includes them in the data accompanying the write of the new “Next transaction” ID it sends to the database header.

The OAT will keep moving forward if transactions are kept short enough to prevent active transactions and garbage from committed newer transactions from piling up too much. The OIT will keep moving forward as long as client processes are committing substantially more work than they are rolling back or losing as the result of crashes. Under these conditions, database performance will be in good shape.

A stuck OAT is worse than a stuck OIT. In a well-performing system, the difference between the OAT and the newest transaction should be a reasonably steady approximation of the (number of client processes running) times (the average number of transactions running per process). Sweep in off-work hours or keep automatic sweeping in force or do both.

In Chapter 27, *Programming with Transactions*, you will find some client application strategies for optimizing the progress of the OIT and OAT.

The “gap”

The “gap” is another piece of Firebird-speak. It refers to the difference between the OIT and the OAT or, more specifically, the Oldest Snapshot. The gap will be small in comparison with overall transaction throughput or, ideally, zero. In these conditions, it can be a reasonable assumption that there are no transactions hanging around that are causing the TSB to bloat and the transaction inventory pages to proliferate outlandishly.

It is not the gap itself that impedes performance. The gap is an indicator of the volume of overhead that sub-optimal transaction management is adding to database activity—over-use and fragmentation of memory, excessive numbers of page-reads during searches and new page-allocations during updates and inserts. Solving and avoiding problems of degrading performance is all about controlling and reducing the gap.

Sweeping vs garbage collection

Garbage collection (GC) is a continual background process that is a function of the normal course of record retrieval and record version checking that is performed for each transaction. When obsolete record versions are found with TIDs lower than the OAT, one of two things will happen:

- on a Classic or Superclassic server they are removed immediately. This is referred to as *cooperative garbage collection*, because each transaction and each server instance participates in clearing garbage left behind by others.
- on Superserver they are “tagged” in an internal list of items for removal by the garbage collection thread. When a GC worker thread is woken up, it will deal with the items in this list and update it.

Sweeping performs this task, too, but, unlike GC, it can also deal with one category of interesting transactions: those that are in “rolled-back” state. It can also remove the stumps of deleted records and release the space for re-use.

The gap is important for automatic sweeping because the sweep interval setting of the database governs the maximum size the gap is allowed to be, before a sweep is triggered off. Automatic sweeping is a backstop that is rarely or never triggered in well-managed databases because the gap never reaches the threshold number.

By default, each database is created with a sweep interval—“maximum gap size”—of 20,000. It can be varied up or down, if necessary, or disabled altogether by setting the interval to 0.

It should be stressed that a gap that is consistently smaller than the sweep interval is not an indication that the database never needs sweeping! All databases must be swept—it is a question of human management whether they are swept automatically, or manually with *gfx*, or both, should the need arise. Think of automatic sweeping as a safety net, not as a substitute for sensible database management.

Transaction statistics

Firebird has some useful utilities for querying how well your database is managing the gap between OIT and OAT. You can use either to inspect the values on the database header page.

gstat

The command-line tool *gstat*, used with the **-header** switch, reveals a number of database statistics, including the current transaction IDs of OIT, OAT, Oldest Snapshot and the next new transaction. To use *gstat*, log in as SYSDBA on the host machine in a command shell and go to your Firebird bin directory. Type

Windows

```
gstat -h <path-to-database> -user sysdba -password whatever
```

POSIX

```
./gstat -h <path-to-database> -user sysdba -password whatever
```

Following is an extract from near the top of the output:

```
Oldest transaction 10075
Oldest active 100152
Oldest snapshot 100152
Next transaction 100153
```

- Oldest transaction is the OIT.
- Oldest active is obviously the OAT.
- Oldest snapshot is usually the same as OAT—it gets written when the OAT moves forward. It is the actual TID that the garbage collector reads as a signal that there is some garbage that it can handle.

For details of *gstat* usage, refer to [*Collecting Database Statistics—gstat*](#) in Chapter 38, **Monitoring and Logging Features**.

isql

You can get a similar view of the database header statistics in an *isql* session, using the **SHOW DATABASE** command. For details, see [*SHOW DATABASE*](#) in the previous chapter.



Many of the third-party Firebird admin tools provide equivalent reports.

What the statistics can tell you

It is a “given” that no garbage collection will ever be performed on obsolete record versions left behind by interesting transactions. However, the Oldest Snapshot marks the boundary where the GC stops looking for committed transactions. Any garbage from that transaction number upwards will be unattended, regardless of whether the newer transactions are interested or not.

If the gap between the OAT and the Next Transaction indicates a much higher number of transactions than you can account for by an enumeration of logged-in users and their tasks, you can be certain that a lot of garbage is missing the dustman. As that gap keeps growing, database performance becomes more and more sluggish. Servers have been known to throw an ‘Out of memory’ error or just crash because the TSB exhausted memory or caused too much fragmentation for the system’s memory management services to handle. Low-end servers—especially those being used to deliver other services—may not even have enough resources left to record the evidence in the log.

If gaps—either between the OIT and the OAT or between the OAT and the Next transaction—seem to be a problem in your system, you can learn a lot about the effects of

sweeping the database and improving your client applications if you keep logs of the statistics.



To obtain a text file of the simple statistics, simply pipe the output of `gstat -h` to a text file according to the rules of your command shell. In the `isql` shell, use the output `<filename>` command to begin piping the output from the `SHOW DATABASE` command.

Trace logs

If you are running Firebird 2.5 or higher over a database of ODS 11.2 or higher, you may also gain insight into problem activity by running traces over a succession of sample periods. For details about running traces and setting up the trace logs, refer to [Trace and Audit Services](#) in Chapter 38, **Monitoring and Logging Features**.

mon\$ tables

When you need to delve deeper to look for the sources of problem transactions, querying the monitoring tables can lead you where you want to go, if your database has an ODS of 11.1 or higher. For details of *mon\$* tables usage, refer to [Monitoring Database Activity](#) in Chapter 38, **Monitoring and Logging Features**.

CHAPTER 26

CONFIGURING TRANSACTIONS

A transaction is a “bubble” around any piece of work affecting the state of a database or, indeed, of more than one database. A user process starts a transaction and the same user process finishes it. Because user processes come in many shapes and sizes, every transaction is configurable. The mandatory and optional properties of a transaction are important parts of any transaction context.

Anatomy of a Transaction

A transaction starts inside the context of a client session. A DSQL statement—`SET TRANSACTION`—could be used by a suitable client application to start the default transaction once it has established a connection.

`SET TRANSACTION` is no longer used by most types of application, being most suited to applications written in the mostly disused ESQL (Embedded SQL) that provided a superset of SQL for old InterBase applications, that could be precompiled with a client application written in C. These days, virtually all client applications written for Firebird use instead the transaction-related API functions exposed by the Firebird client libraries, wrapped in one way or another in host language structures (classes, components, etc.). All of the parameters for `SET TRANSACTION` have counterparts in API functions and their argument structures—and more, since `SET TRANSACTION` comes with limitations.

One application that can use `SET TRANSACTION` dynamically is *isql* in its interactive mode, providing a useful environment for familiarising yourself with the various combinations of its parameters. It affects only the transactions in which DML statements run, since they use the default transaction. *isql* runs DDL in separate transactions.

The Default Transaction

In DSQL, the only transaction that can be started with `SET TRANSACTION` is the default one that is started automatically for the application if the application does not start

It can only be a single-database transaction and it is not possible to start multiple transactions with `SET TRANSACTION`.

Syntax `SET TRANSACTION` has a few mandatory parameters and some optional ones. Used without any explicit parameters, it starts the transaction with default values in the mandatory parameters and no action for the optional ones.

Note *The syntax pattern describes the options for DSQL. Embedded SQL (ESQL) uses `SET TRANSACTION` differently and has a slightly different parameter set.*

Pattern:

```
SET TRANSACTION
[READ WRITE | READ ONLY]
[ [ISOLATION LEVEL] {
    SNAPSHOT [TABLE STABILITY] | READ COMMITTED [[NO] RECORD_VERSION]
} ]
[WAIT | NO WAIT]
[LOCK TIMEOUT <seconds>]
[NO AUTO UNDO]
[IGNORE LIMBO]
[RESERVING table1 [, table2 ...]]
[FOR [SHARED | PROTECTED] {READ | WRITE}]
```

Default settings

If `SET TRANSACTION` is requested without any parameters, the default transaction is configured with `SNAPSHOT` isolation in `READ WRITE` mode and the `WAIT` lock resolution policy and no lock timeout.

About Concurrency

The term *concurrency* broadly refers to the state in which two or more tasks are running inside the same database at the same time. In these conditions, the database is sometimes said to be supporting parallel tasks.

Inside a transaction's own “bubble”, the owning process will be allowed to perform any operation that

- 1 is consistent with its own current view of the database and
- 2 would not, if committed, interfere with the consistency of any other active transaction's current view of the database

The engine's interpretation of consistency is governed by the transaction's configuration. That, in turn, is governed by the client application. Each transaction can be configured by means of a constellation of parameters that enable the client process to predict, with absolute assurance, how the database engine will respond if it detects a potential inconsistency.

Factors affecting concurrency

The four configurable parameters that affect concurrency are:

- isolation level
- lock resolution policy (a.k.a. blocking mode)
- access mode
- table reservation

For one level of isolation (READ COMMITTED) the current *states of the record versions* are also considered.

Isolation level

Firebird provides three outright levels of transaction isolation to define the depth of consistency the transaction requires. At one extreme, a transaction can get exclusive write access to an entire table whilst, at the other, the uncommitted transaction becomes current with every external change of database state.



No Firebird transaction will ever see any uncommitted changes pending for other transactions.

In Firebird, isolation level can be READ COMMITTED, SNAPSHOT or SNAPSHOT TABLE STABILITY. Within READ COMMITTED, two sub-levels are available: RECORD_VERSION and NO_RECORD_VERSION.

Standard levels of isolation

The SQL standard for transaction isolation is sympathetic to the two-phase locking mechanism that most RDBM systems use to implement isolation. As standards go, it is quite idiosyncratic in comparison with many of the other standards. It defines isolation not so much in terms of ideals as in terms of the phenomena each level allows (or denies). The phenomena with which the standard is concerned are:

Dirty read—occurs if the transaction is able to read the uncommitted (pending) changes of others.

Non-repeatable read—occurs if subsequent reads of the same set of rows during the course of the transaction could be different to what was read in those rows when the transaction began.

Phantom rows—phantoms occur if a subsequent set of rows read during the transaction differs from the set that was read when the transaction began. The phantom phenomenon happens if the subsequent read includes new rows inserted and/or excludes rows deleted that were committed since the first read.

Table 26.1 shows the four standard isolation levels recognized by the standard, with the phenomena that govern their definitions.

Table 26.1 SQL Standard isolation levels and governing phenomena

Isolation level	Dirty Read	Non-repeatable Read	Phantoms
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	Disallowed	Allowed	Allowed
REPEATABLE READ	Disallowed	Allowed	Allowed
SERIALIZABLE	Disallowed	Disallowed	Disallowed



READ UNCOMMITTED is not supported in Firebird at all.
READ COMMITTED conforms with the standard. At the two deeper levels, the nature of MGA prevails over the two-phase locking limitations implied by the standard. Mapping to the standard governance of REPEATABLE READ and SERIALIZABLE is not possible.

READ COMMITTED

The shallowest level of isolation is READ COMMITTED. It is the only level whose view of database state changes during the course of the transaction since, every time a commit version of a record it is accessing changes, the newly committed record version replaces the version the READ COMMITTED transaction began with. Inserts committed since this transaction began are made visible to it.

By design, READ COMMITTED isolation allows non-repeatable reads and does not prevent the phenomenon of phantom rows. It is the most useful level for high-volume, real-time data entry operations because it reduces data contention, but it is unsuitable for tasks that need a reproducible view.

Because of the transient nature of Read Committed isolation, the transaction (“ThisTransaction”) can be configured to respond more conservatively to external commits and other pending transactions:

- with RECORD_VERSION (the default flag) the engine lets ThisTransaction read the latest committed version. If ThisTransaction is in READ WRITE mode, it will be allowed to overwrite the latest committed version, even if its own TID is newer than the TID on the latest committed version.
- with NO_RECORD_VERSION the engine effectively mimics the behavior of systems that use two-phase locking to control concurrency. It blocks ThisTransaction from writing a new version of the row if there is a pending update on the latest current version. Resolution depends on the lock resolution setting:
 - with WAIT, ThisTransaction waits until the other transaction either commits or rolls back its change. Its change will then be allowed if the other transaction rolled back or if its own TID is newer than the other transaction's TID. It will fail with a lock conflict if the other transaction's TID was newer.
 - with NOWAIT, ThisTransaction immediately receives a lock conflict notification.



The transaction can “see” all of the latest committed versions that its settings allow it to read, which is fine for insert, update, delete and execute operations. However, any output sets selected in the transaction must be requeried by the client application to get the updated view.

SNAPSHOT (Concurrency)

The “middle” level of isolation is SNAPSHOT, alternatively termed *Repeatable Read* or *Concurrency*. However, Firebird's SNAPSHOT isolation does not accord exactly with Repeatable Read as defined by the standard. It isolates the transaction's view from row-level changes to existing rows. However, because the MGA architecture, by nature, completely isolates snapshot transactions from new rows committed by other transactions, by denying SNAPSHOT transactions access to the global transaction state bitmap (TSB, see previous chapter), there is no possibility of SNAPSHOT transactions seeing phantom rows. Hence, a Firebird SNAPSHOT transaction provides a deeper level of isolation than the SQL's REPEATABLE READ.

Yet SNAPSHOT is not identical to SERIALIZABLE, because other transactions can update and delete rows that are in the purview of the snapshot transaction, provided they post first.

The transaction is guaranteed a non-volatile view of the database that will be unaffected by any changes committed by other transactions before it completes. It is a useful level for historical tasks like reporting and data export, which would be inaccurate if not performed over a completely reproducible view of the data.

SNAPSHOT is the default isolation level for the isql query tool and for many component and driver interfaces.

SNAPSHOT TABLE STABILITY (Consistency)

The “deepest” level of isolation is SNAPSHOT TABLE STABILITY, alternatively termed *Consistency* because it is guaranteed to fetch data in a non-volatile state which will remain externally consistent throughout the database as long as the transaction lasts. Read-write transactions can not even read tables that are locked by a transaction with this isolation level.

The table-level locking imposed by this isolation comprises all tables accessed by the transaction, including those with referential constraint dependencies.

This level constitutes an aggressive extension that guarantees serialization in the strict sense that no other transaction can insert or delete—or indeed, change—rows in the tables involved if any transaction succeeds in acquiring a handle with this isolation. Conversely, the TABLE STABILITY transaction will not be able to acquire a handle if any read-write transaction is currently reading any table that is in its intended purview. In terms of the standard, it is unnecessary, since SNAPSHOT isolation already protects transactions from all three of the phenomena governed by the SQL standard SERIALIZABLE level.

A consistency transaction is also referred to as a *blocking transaction* because it blocks access by any other read-write to any of the records that it is accesses and to any records that depend on those records.



Because of its potential to lock up portions of the database to other users needing to perform updates, SNAPSHOT TABLE STABILITY must be used with extreme care. Attend carefully to the size of the affected sets, the effects of joins and table dependencies and the likely duration of the transaction.

Locking Policy

Lock policy determines behaviour in the event the transaction (“ThisTransaction”) tries to post a change that conflicts with a change already posted by another transaction. The options are WAIT and NO WAIT.

WAIT

WAIT (the default) causes the transaction to wait until rows locked by a pending transaction are released, before determining whether it can update them. At that point, if the other transaction has posted a higher record version, the waiting transaction will be notified that a lock conflict has occurred.

WAIT is often not the preferred blocking mode in high-volume, interactive environments because of its potential to slow down the busiest users and, in some conditions, to cause “livelocks” (see below).

WAIT is virtually pointless in SNAPSHOT isolation. Unless the blocking transaction eventually rolls back—the least likely scenario—the outcome of waiting is certain to be a lock conflict, anyway. In a READ COMMITTED transaction, the likelihood that the outcome of waiting would be a lock conflict is much reduced.

That is not to deny the usefulness of WAIT for some conditions. If the client application's exception handler handles conflicts arising from NO WAIT by continually retrying without pausing, the bottlenecking caused by repeated retries and failures is likely to be worse than if WAIT is specified, especially if the blocking transaction takes a long time to complete. By contrast, WAIT is potentially going to cause one exception, eventually handled by one rollback.

LOCK TIMEOUT

From the “2” series onward, a transaction can be optionally configured with a positive, non-zero integer value to specify how many seconds a WAIT should continue before excepting.



LOCK TIMEOUT and NO WAIT are mutually exclusive.

Where the likelihood of transactions colliding is high but transactions are short, WAIT is to be preferred because it guarantees that waiting requests will proceed in a FIFO sequence, rather than be required to take their chances with each repeated request. However, in user environments where a quick turnover can not be guaranteed, WAIT transactions are contraindicated because of their potential to hold back garbage collection.

NO WAIT

In a NO WAIT transaction, the server will notify the client immediately if it detects a new, uncommitted version of a row the transaction tries to change. In a reasonably busy multi-user environment, NO WAIT is sometimes preferable to the risk of creating bottle-necks of waiting transactions.

As a rule of thumb for SNAPSHOT transactions, throughput will be faster and interfaces more responsive if the client application chooses NO WAIT and handles lock conflicts through the use of rollback, timed retries or other appropriate techniques.

Access Mode

Access mode can be READ WRITE or READ ONLY. A READ WRITE transaction can select, insert, update and delete data. A READ ONLY transaction can only select data.

The default access mode is READ WRITE.



One of the benefits of a READ ONLY transaction is its ability to provide selection data for the user interface without tying up excessive resources on the server. Make sure your read-only transactions are configured with READ COMMITTED isolation, to ensure that garbage collection on the server can proceed past this transaction.

Table Reservation

Firebird supports a table locking mode to force full locks on one or more tables for the duration of a transaction. The optional `RESERVING <list of tables>` clause requests immediate full locks on all committed rows in the listed tables, enabling a transaction to guarantee itself exclusive access at the expense of any transactions that become concurrent with it.

Unlike the normal optimistic locking tactic, reservation (`RESERVING` clause with specified table arguments) locks all rows pessimistically—it takes effect at the start of the transaction, instead of waiting until the point at which an individual row lock is required.



You can reserve more than one table in a transaction.

Table reservation has three main purposes:

- to ensure that the tables are locked when the transaction begins, rather than when they are first accessed by a statement, as is the case when `TABLE STABILITY` isolation is used for table-level locking. The lock resolution mode (`WAIT/NOWAIT`) is applied during the transaction request and any conflict with other transactions having pending updates will result in a `WAIT` for the handle, or a denial of the handle in the `NOWAIT` case. This feature of table reservation is important because it greatly reduces the possibility of deadlocks.
- to provide dependency locking, i.e. the locking of tables which might be affected by triggers and integrity constraints. Dependency locking is not normally in Firebird. However, it will ensure that update conflicts arising from indirect dependency conflicts will be avoided.
- to strengthen the transaction's precedence with regard to one or more specific tables with which it will be involved. For example, a `SNAPSHOT` transaction that needs sole write access to all rows in a particular table could reserve it, while assuming normal precedence with regard to rows in other tables. This is a less aggressive way to apply table-level locking than the alternative, to use `TABLE STABILITY` isolation.

Uses for Table Reservation

When table-level locking is required, table reservation with `SNAPSHOT` or `READ COMMITTED` isolation is recommended in preference to using `SNAPSHOT TABLE STABILITY` when table-level locking is required. Table reservation is the less aggressive and more flexible way to lock tables pre-emptively. It is available for use with any isolation level. However, using it in combination with `SNAPSHOT TABLE STABILITY` is not recommended, because it has no effect in mitigating the tables that the transaction might access that are out of its scope.

Pre-emptive table locking is not for everyday use but it can be usefully employed for a task such as a pre-audit valuation or an “inventory freeze” report prior to a stocktake.

Parameters for table reservation

Each table reservation can be configured with distinct attributes to specify how multiple transactions should be treated when they request access to reserved table.



*A **SNAPSHOT TABLE STABILITY** transaction can not get any access to ANY table that is reserved by table reservation.*

The choices are:

[**PROTECTED** | **SHARED**] {**READ** | **WRITE**}

The **PROTECTED** attribute gives ThisTransaction an exclusive lock on the table it is reading and allows any other transaction that is in **SNAPSHOT** or **READ COMMITTED** isolation to read rows. Writes are restricted by one of the two modifiers:

- **PROTECTED WRITE** allows ThisTransaction to write to the table.
- **PROTECTED READ** disallows writing to the table by any transaction, including ThisTransaction itself.

The **SHARED** attribute lets any **SNAPSHOT** or **READ COMMITTED** transaction read the table and provides two options for concurrent updating by other transactions:

- **SHARED WRITE** allows any **SNAPSHOT** read-write or **READ COMMITTED** read-write transaction to update rows in the set but no transaction as long as no transaction has or requests exclusive write
- **SHARED READ** is the most liberal reserving condition. It allows any other read-write transaction to update the table.

Summary

Any other transaction can read a table reserved by ThisTransaction, provided there is no aspect of the other transaction’s configuration that gives it an exclusive **WRITE** right on the table, i.e. all can read if they are configured only to read and not to have any pre-emptive right to write. The following conditions will always block the other transaction from reading a table reserved by ThisTransaction:

- the other transaction is in **SNAPSHOT TABLE STABILITY** isolation
- the other transaction is configured to reserve the table **PROTECTED WRITE** (although it can read the table if ThisTransaction is reserving it with **SHARED READ**)
- the other transaction wants to reserve the table **SHARED WRITE** and ThisTransaction is reserving it with **PROTECTED READ** or **PROTECTED WRITE**

In case this is all too confusing still, in Figure 26.1, we’ll let some configured transactions speak for themselves,

Figure 26.1 Configuring table reservation

Other Optional Parameters

IGNORE LIMBO, supported from the “2” series onward, tells the engine to ignore “in limbo” transactions that have been left behind by a failed multi-database transaction.



This parameter is available to applications via the API, for all versions of Firebird, as the member `isc_tpb_ignore_limbo` in the transaction parameter block (TPB).

NO AUTO UNDO, supported from the “2” series onward, tells the engine not to keep the internal “undo” log. By default, the server maintains this *internal savepoint log* of inserted and changed rows in memory. In the normal course of events, log entries are erased as each transaction commits or rolls back. However, under certain conditions, the system will abandon the log and refer directly to the global transaction state bitmap (TSB) instead. The transition is likely to happen when a huge insert is rolled back or a transaction using many repeated user savepoints (q.v.) has exhausted the capacity of the log.



This parameter is available to applications via the API, for all versions of Firebird, as the member `isc_tpb_no_auto_undo` in the transaction parameter block (TPB).

Record Versions

When an update request is successfully posted to the server, Firebird creates and writes to disk a reference linking the original row image as seen by the transaction—sometimes called a delta—and a new version of the row, incorporating the requested changes. The original and new row images are referred to as record versions.

When a new record version is created by a newer transaction than the one that created the “live” version, other transactions will not be able to update or delete the original unless the owner-transaction of the new version rolls back. The versioning process is described in detail in the previous chapter.

Until the transaction is eventually committed, it does not touch the “live” version again until the Commit occurs. Within its own context, it treats the posted version as if it were the latest committed version. Meanwhile, other transactions continue to “see” the latest committed version. In the case of SNAPSHOT transactions that started before This Transaction, the latest committed version that they see may be older than the one seen by our transaction and by other transactions that either started later or are in READ COMMITTED isolation.

Dependent rows

If the table that has an update posted to it has foreign keys linked to it, the server creates deltas of the rows from those tables that “belong to” the updated row. Those dependent rows—and any that are dependent on them through foreign keys—are thus made inaccessible for update by other transactions too, for the duration of the transaction.

Locking and Lock Conflicts

With Firebird, locking is governed by the relative ages of transactions and the records managed by Firebird’s versioning engine. All locking applies at row level, except when a transaction is operating in SNAPSHOT TABLE STABILITY isolation or with a table reservation restriction that blocks write access.

Timing

The timing of the lock on the row in normal read-write activity is optimistic—no locking is in force on any row until the moment it is actually required. Until an update of the row is posted to the server, the row is free to be “won” by any read-write transaction.

Pessimistic locking

Pessimistic—or pre-emptive—locking can be applied to sets of rows or to entire tables. The table-locking options have already been explained (see [Table Reservation](#) and [SNAPSHOT TABLE STABILITY \(Consistency\)](#) isolation).

Row-level and set-level pessimistic locking is also an option where there is an application requirement to reserve a row or a small set in advance of actually posting an update or deletion. It is not a transaction setting.

Explicit row locking

Explicit pessimistic row locking is achieved with the optional WITH LOCK clause in the SELECT statement. It is restricted to “outer-level” SELECT statements that return output sets or define cursors. It cannot be applied to subqueries or virtual output sets that use SELECT expressions.

The abbreviated syntax for acquiring explicit pessimistic row locks is:

```

SELECT <output-list>
FROM <table-or-procedure-or-view>
[WHERE <search-conditions>]
[GROUP BY <grouping-specification>]
[UNION <select-expression> [ALL]]
[PLAN <plan-expression>]
[ORDER BY <column-list>]
[FOR UPDATE [OF col1 [,col2..]] [WITH LOCK]]

```

FOR UPDATE—which is not a locking instruction—requests that the output set be delivered to the client one row at a time, rather than as a batch. The optional phrase WITH LOCK is the element that forces the pre-emptive lock on a row as soon as the server outputs it from the server. Rows waiting to be output are not locked.

v.1.0.x Explicit row locking is not supported in v.1.0.x.

Dummy updates

The traditional way of achieving a pessimistic row lock with Firebird is the “dummy update”. It is a hack that takes advantage of record versioning. Simply put, the client posts an update statement for the row that doesn’t update anything: it just sets a column to its current value, causing the server to create a new record version and thus blocks other transactions from acquiring the row for updating or deletion.

The conditions where pessimistic locking might help and the recommended techniques are discussed in the next chapter, *Programming with Transactions*.

Lock conflicts

A lock conflict is triggered when concurrent transactions try to update or delete the same row during a time when their views of database state overlap. Lock conflicts are the planned outcome of Firebird's transaction isolation and record versioning strategy, protecting volatile data from uncontrolled overwriting by parallel operations on the same data.

The strategy works so well that there are really only two conditions that can cause lock conflicts:

- Condition 1: one transaction—“ThisTransaction”—has posted an update or deletion for a row that another transaction, which started before ThisTransaction locked that row, attempts to update or delete. The other transaction encounters a lock conflict and has two choices: either
 - it can roll back its attempt and try again later against the newest committed version, or
 - it can wait until ThisTransaction either commits or rolls back.
- Condition 2: ThisTransaction is blocking the whole table against writes, because it has the table isolated in SNAPSHOT TABLE STABILITY or by a PROTECTED table reservation, and another transaction tries to update or delete a row or to insert a new row.

Suppose ThisTransaction posts a change to a row. Another transaction comes along and requests to change or delete the same row. In SNAPSHOT isolation with WAIT, the other

transaction will keep waiting until `ThisTransaction` completes with either commit or rollback.

If `ThisTransaction` commits, then the other transaction will fail with an update conflict. The client that started the other transaction should have an exception handler that either rolls the transaction back and starts a new one to resubmit the request, or simply commits the transaction and exits.

Calling `COMMIT` in a lock conflict exception handler is not recommended, since it breaks the atomicity of the transaction—some work will complete, some will not and it will be impossible to predict database state afterwards. Rollback is almost always the right response to a lock conflict.

Unfortunately, Firebird tends to generalize all locking exceptions and report them as “deadlocks”. The normal case just described is not a deadlock

What is a deadlock?

A deadlock is just a nickname, borrowed from the sport of wrestling, for the condition where two transactions are contending to update rows in overlapping sets and one transaction does not take any precedence over the other.

For example, `ThisTransaction` has an update pending on Row X and wants to update Row Y, while the other transaction has an update pending on Row Y and wants to update Row X and both transactions are in `WAIT` mode. As in wrestling, the deadlock can only be resolved if one contender withdraws its hold. One transaction must roll back and let the other commit its changes.

Firebird allows the application to resolve the deadlock by scanning for deadlocks every few seconds. It will arbitrarily pick one transaction from the deadlock and deliver a deadlock exception.

Developers should not dread deadlock messages. On the contrary, they are the essence of isolating the work of multiple users in transaction contexts. You should anticipate them and handle them effectively in your client application.

When `ThisTransaction` is selected to resolve the deadlock, the application's exception handler should roll it back to allow the other transaction to resume and complete its work. The alternative—committing `ThisTransaction` in the exception handler—is not recommended, since `ThisTransaction` becomes non-atomic and the other transaction will fail with a lock conflict.

“Deadly embrace”

In rare cases, more than two transactions could be deadlocked in contention for the same overlapping sets. It is sometimes called “the deadly embrace”. The deadlock scan will fail one transaction (`ThisTransaction`), handing it over to the client for exception resolution, as before. However, this time, even if the client rolls back `ThisTransaction`, those other transactions are still deadlocked out there.

“Livelock”

The client might start a new transaction and retry, but the other contenders are still deadlocked, waiting for the next deadlock scan to extricate another of the contenders with a deadlock exception. As long as the retrying application keeps retrying with a `WAIT` transaction, it is just going to wait for some indefinite time for the other transactions to resolve the deadly embrace. For those futile retries, the transactions are said to be in a “livelock”.

In short, it is important in the first place to avoid transaction contexts that make it possible for a deadly embrace to occur. As added protection, exception handlers should be made capable of quickly dealing with deadlocks and ensuring that problem transactions finish cleanly and without delay.



In the “2” series clients, you have the option of setting a LOCK TIMEOUT in seconds. If your applications are getting deadlocks too often, experiment with a few low values first—say 1 to 5 seconds—and see whether it improves the situation.

CHAPTER 27

PROGRAMMING WITH TRANSACTIONS

The transaction is the starting point for anything a client application does in its dealings with the server. In this chapter we take a client's eye view of the various interfaces for starting, managing and ending transactions.

Many language and development environments can interface with Firebird. It is outside the scope of this manual to describe in detail how to manage Firebird transactions from each one.

The Language of Transactions

It is important to address the features Firebird implements for transactions, yet several transaction-related features are not fully implemented in dynamic SQL, but only through the API.

Of the few transaction-related SQL statements available in the DSQL subset, only COMMIT and ROLLBACK are available to every interface. The SET TRANSACTION statement, discussed in the previous chapter, implements a way to access the “default transaction” through DSQL in some environments but it does not provide a route for applications to take advantage of features such as multiple concurrent transactions or multi-database transactions.

The real power of the transaction is realised in the Firebird API. While direct API programming with Firebird is rare today, virtually every host language-specific driver or application interface layer for Firebird wraps function calls to the API. It is through these layers that applications are able to work fully with transactions.

Although this is not a chapter about the API¹, the following topics will, in places, allude to it in a more or less general way to give some appreciation of what passes across the interface when clients talk to servers about transactions.

1. For the enthusiast, Borland published the API Guide as part of its beta InterBase® 6.0 media kit in 2000. Versions of this volume are available in PDF format from some websites—just Google "APIGuide.pdf".

The API

The API presents a flexible, if exacting, interface environment of complex functions for C and C++ programmers to build client applications with the thinnest possible communication and connectivity layer. A group of API functions implements the equivalent SQL statements relating to transactions: for example, *isc_start_transaction()* for SET TRANSACTION, *isc_commit_transaction()* for COMMIT.

The API header file, *ibase.h*, declares the function prototypes, type definitions for each structure, parameter definitions and macros that are used by functions. It is deployed with Firebird in the */include* directory.

For several object-oriented development environments, such as Object Pascal, Borland C++, Java, PHP, Python and DBI-Perl, classes and components are available that encapsulate the Firebird API calls relating to transactions comprehensively. Custom drivers for standard SQL database connectivity interfaces—notably ODBC, JDBC and .NET—similarly surface the API.²

The API function that does the equivalent job to SET TRANSACTION is called *isc_start_transaction()*.

Starting a Transaction

Starting each transaction has three parts:

- 1 Creating (if necessary) and initializing the transaction handle.
- 2 Creating (if necessary) and populating a transaction parameter buffer (TPB) to carry the configuration data—this is optional.
- 3 Calling *isc_start_transaction()*.

Without the optional transaction parameter buffer (TPB), the client starts a transaction exactly like the default transaction that a bare SET TRANSACTION statement starts—with SNAPSHOT isolation in READ WRITE mode and the WAIT lock resolution policy and no lock timeout.

The transaction handle

Each time you want to call this function, you must have a long pointer variable—called a transaction handle—already declared in your application and initialized to zero. An application needs to provide one transaction handle for each concurrent transaction and you can "recycle" used handles by re-initializing them.

A transaction handle must be set to zero in order to initialize it before starting a transaction. A transaction will fail to start up if it is passed a non-zero handle.

2. For most of these implementations, someone has done the job of translating Firebird's C header file into the appropriate high-level language, in order to provide the Firebird (or InterBase®) API to that language. If you are planning to try your hand at direct-to-API programming yourself, it is well worth your while to search the Web for an existing header translation.

The transaction parameter buffer (TPB)

A TPB is a byte array (or vector) of constants, each representing a transaction parameter and starting with the prefix *isc_tpb_*. The first parameter is always the constant *isc_tpb_version3*, which defines the version of TPB structure. The subsequent array members are all constants that map to an equivalent SQL transaction attribute. Table 27.1 (below) shows each SQL transaction parameter and its equivalent TPB constant.

A typical TPB declaration in C looks like this:

```
static char isc_tpb[] =
{
    isc_tpb_version3,
    isc_tpb_write,
    isc_tpb_wait,
    isc_read_committed,
    isc_tpb_no_rec_version
};
```

This TPB is identical in its effect to

```
SET TRANSACTION
  READ WRITE
  WAIT
  READ COMMITTED
  NO RECORD_VERSION;
```

You can use the same TPB for all transactions that need the same characteristics. It is also fine to define a distinct TPB for each transaction. In language-specific interface layers, transaction classes typically create a TPB instance whenever they need one and surface the settings to the design or run-time interface as read-write properties (a.k.a. data members).

In many cases, you can recognize the TPB constant in the property name. For standard driver layers, such as JDBC or ODBC, the names of classes and members are more likely to be dictated by the standard and mapped to the attributes most nearly matching the prescribed functionality.

Table 27.1 SQL transaction attributes and equivalent TPB constants

Attribute type	SQL attribute	TPB constant
Access mode	READ ONLY	isc_tpb_read
	READ WRITE	isc_tpb_write
Isolation level	READ COMMITTED	isc_tpb_read_committed
	RECORD_VERSION	isc_rec_version
	NO RECORD_VERSION	isc_no_rec_version
	SNAPSHOT	isc_tpb_concurrency
Lock resolution policy	SNAPSHOT TABLE STABILITY	isc_tpb_consistency
	WAIT	isc_tpb_wait
	LOCK TIMEOUT n	isc_tpb_lock_timeout
Table reservation	NO WAIT	isc_tpb_nowait
	SHARED	isc_tpb_shared

Attribute type	SQL attribute	TPB constant
	PROTECTED	isc_tpb_protected
	READ	isc_tpb_lock_read
	WRITE	isc_tpb_lock_write
Disable auto-undo log	NO AUTO UNDO	sc_tpb_no_auto_undo
Ignore limbo transactions	IGNORE LIMBO	isc_tpb_ignore_limbo

For more detail about the parameters, refer to the previous chapter, *Configuring Transactions*.

Accessing the Transaction ID

The transaction ID (TID) of the current transaction (as deduced from the stored transaction state data) is available as a context variable. It is available in DSQL, triggers, stored procedures and *isql*. The variable is `CURRENT_TRANSACTION`³.

For example, to get it in a query, you could request:

```
SELECT CURRENT_TRANSACTION AS TRAN_ID FROM RDB$DATABASE;
```

To store it in a table:

```
INSERT INTO TASK_LOG (USER_NAME, TRAN_ID, START_TIMESTAMP)
VALUES (CURRENT_USER, CURRENT_TRANSACTION, CURRENT_TIMESTAMP);
```

Caveat

Transaction IDs are not stable enough to use as keys. The numbering series for TIDs gets reset to 1 when a database is restored.

Using the TID in applications

The TID from the server is not the same as the transaction handle that the Firebird client fetches back into your application. It is quite valid to associate a TID with a transaction handle, provided you remember to re-map the relationship between TID and handle each time you re-use a handle. Every use of a handle should be atomic in your application.

The TID can be a useful asset for tracking down applications and users responsible for long-running transactions in a system that continually exhibits degrading performance. A completely cyclic tracking routine could record the server timestamps when transactions both start and finish (commit or roll back). In particular, missing finish times will help to find users who habitually use the “reset” button to terminate tasks or application work-flows that are failing to commit work in a timely fashion.

From v.2.1 onward, the monitoring tables (MON\$ tables) retrieve useful information at transaction level, as well as about the owing connection and the statements that the transaction has in its command. Developers can write tools to watch transactions behaviour, since the MON\$ tables can be queried, just like regular user tables.

For more information, refer to *Monitoring Database Activity* in Chapter 38, *Monitoring and Logging Features*.

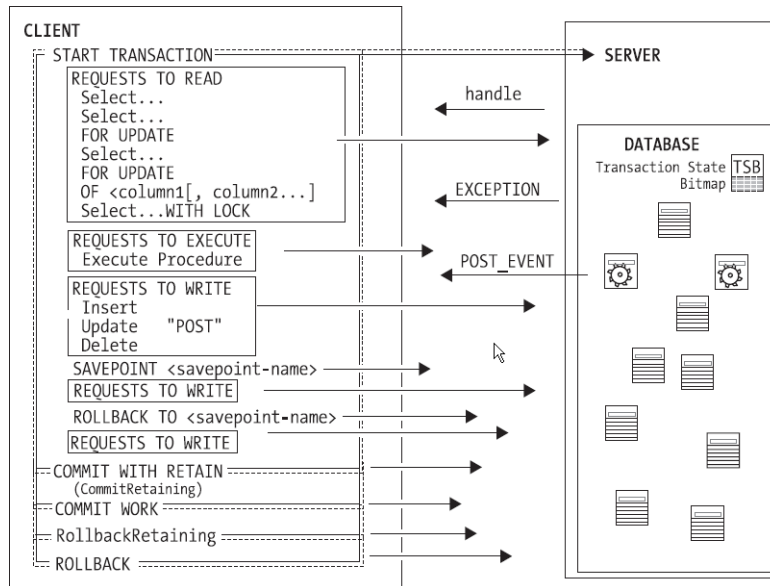
3. `CURRENT_TRANSACTION` is not available in Firebird 1.0.x.

Progress of a Transaction

At this point, you should be getting a sense of confidence about the point of operating a multi-user system as a series of atomic tasks, isolated in transactions.

It is established that clients start transactions and clients end them. Beyond configuring a transaction to wrap a task for a specific set of access, isolation and lock resolution conditions, the client has other opportunities to interact with the transaction during the course of its activity. Figure 27.1 illustrates these points of interaction and the remaining topics in this chapter expand on the techniques and implications surrounding them.

Figure 27.1 Interaction of application and transaction



Nested Transactions

In Firebird, transactions are always started and finished by a client. Some other DBM systems can start and commit transactions from within stored procedures because two-phase locks and transaction logging are used to control transactions.

All versions of Firebird provide mechanisms that can operate on the flow of work within a transaction without breaking atomicity. Two of these mechanisms—exception handler blocks and executable strings⁴—are restricted for use within PSQL modules and are discussed in Part Seven. The other, not available in PSQL, is *user savepoints*.⁵

- Executable strings, via the EXECUTE STATEMENT command, have been supported in Firebird PSQL since v.1.5. The ability to perform autonomous transactions from within an EXECUTE STATEMENT construct was introduced in v.2.5.
- Executable strings and user savepoints are not available in v.1.0.x.

User savepoints

User savepoint statements—also known as nested transactions—enable you to “batch” groups of operations inside a transaction and signpost them when they post successfully. Should an exception occur later in the task sequence, the transaction can be rolled back to the last savepoint. Operations that were posted between the savepoint and the exception are rolled back and the application can correct, branch, commit, perform a full rollback, resume, or whatever is required.

Setting a savepoint

User savepoints are a client-side operation, available only as DSQL statements.

The statement for setting a savepoint is:

```
SAVEPOINT <identifier>;
```

The identifier can be any valid Firebird SQL identifier (31 characters maximum, US ASCII alphanumeric characters, and distinct within the transaction). You can re-use the same identifier in the same transaction and it will overwrite any existing savepoint associated with the name.

Rolling back to a savepoint

A rollback to a savepoint begins by rolling back all the work posted after the savepoint was set. The named saved point and any that preceded it are retained. Any savepoints that were set after the named savepoint are lost.

Any locks—explicit or implicit—that were acquired since the named savepoint was set are released, although transactions that are waiting for access to the released rows still do not get access to them until the transaction completes. This ensures that currently waiting contenders will not interfere with the established workflow if it resumes. It does not block a transaction that was not waiting at the time of the nested rollback.

Syntax `ROLLBACK [WORK] TO [SAVEPOINT] <identifier>;`

If the transaction is allowed to resume after rolling back to a save point, the workflow can roll back to the same savepoint again, as many times as necessary. Record versions that get rolled back will not be eligible for garbage-collection because the transaction is still active.

Releasing savepoints

On the server, the savepoint mechanism—a memory-based log—can have a significant impact on resources, especially if the same rows receive updates many times during the course of retracking through the task. The resources from unwanted savepoints can be released using a `RELEASE SAVEPOINT` statement:

```
RELEASE SAVEPOINT <identifier> [ONLY];
```

Without the keyword `ONLY`, the named savepoint and all savepoints that were set after it will be released and lost. Use `ONLY` to release just the named savepoint.

The following example illustrates how savepoints work:

```
CREATE TABLE SAVEPOINT_TEST (ID INTEGER);
COMMIT;
INSERT INTO SAVEPOINT_TEST
VALUES(99);
COMMIT;
INSERT INTO SAVEPOINT_TEST
```

```
VALUES(100);
--
SAVEPOINT SP1;
--
DELETE FROM SAVEPOINT_TEST;
SELECT * FROM SAVEPOINT_TEST; /* returns nothing */
--
ROLLBACK TO SP1;
--
SELECT * FROM SAVEPOINT_TEST; /* returns 2 rows */
ROLLBACK;
--
SELECT * FROM SAVEPOINT_TEST; /* returns the one committed row */
```

Internal savepoints

When the server engine executes rollbacks, its default point of reference for backing out a transaction is to internal savepoints stored in the memory-based auto-undo log. At the end of the rollback, it commits the transaction. The purpose of this strategy is to reduce the amount of garbage generated by rollbacks.

When the volume of changes performed under a transaction-level savepoint is getting large—into the range of 10,000 to a million rows affected—the engine abandons the auto-undo log and takes its references directly from the global TSB (the transaction state bitmap). If you have a transaction that you expect to command a large number of changes, disabling auto-undo logging will forestall the wastage of resource consumption that would occur if the server decided to abandon the logging. For more information, see the note about NO AUTO UNDO in the topic [Other Optional Parameters](#) in the previous chapter.

Exception handling extensions in PSQL

The equivalent of savepoints in PSQL modules is exception handling. Each PSQL exception handling block is bounded by automatic system savepoints. The PSQL extensions provide language wrappers implementing the same kind of nesting that user savepoints provide in DSQL. For more information, see [Trapping and Handling Exceptions](#) in Chapter 32, *Error Handling and Events*.

The Logical Context

The simple way to look at a transaction between SET (START) TRANSACTION and COMMIT or ROLLBACK is to view it as a series of client operations and client/server interactions that map exactly to a task. It is a very useful model for understanding how a transaction wraps a unit of work.

The tight boundaries of a transaction do not not necessarily reflect the reality of how users perform the actual task.

From the user's point of view, the “task” is not bounded by starting a transaction and committing it. Her task has a beginning, a middle and an end which may involve multiple transactions to reach completion. For example, a failure to post or to commit a statement will entail a rollback to end the physical transaction. Some form of intervention follows

which may cause the logical task to end or, in the normal case, will require another physical transaction in order to complete it.

A single physical transaction may involve a number of discrete user tasks forming a logical “whole” that requires the atomicity of a single physical transaction. Another variant—the typical batch data entry task—encompasses many repetitions of a similar task stacked up inside a single physical transaction to reduce keystrokes and comply with the user's workflow requirements.

In summary, the logical task—the one we as developers must address and design for—almost always extends beyond the boundaries of the start of the physical transaction and the COMMIT or ROLLBACK request that ends it. The physical transaction is but one facet of the logical context in which it is conducted.

Two key factors of concern in the logical context of transaction workflow are:

- how to retain the physical context of the initial transaction after ROLLBACK so that users' work does not disappear when the server refuses to accept it
- what to do if the flow is interrupted by an exception—how to diagnose the exception and how to correct it

To address these problems, we look at the COMMIT and ROLLBACK operations and the pros and cons of options available to preserve logical context between transactions. Following on from there, we examine the issue of diagnosing the exceptions that give rise to the need to “re-run” transactions.

Ending transactions

A transaction ends when the client application commits it or rolls it back. Unless the COMMIT statement, or the call to the equivalent API function `isc_commit_transaction`, succeeds, the transaction is not committed. If a transaction that can not be committed is not rolled back by an explicit client call to ROLLBACK (or the API function `isc_rollback_transaction`) the transaction is not rolled back. These statements are not the syllogism that they appear to be. Failure to terminate transactions after exceptions occur is a common feature of problem scenarios in the support lists!

The COMMIT statement

The syntax of the COMMIT statement is simple:

```
COMMIT [WORK] [RETAIN [SNAPSHOT]];
```

A simple COMMIT—sometimes referred to as a “hard commit”, for reasons that will become evident—makes changes posted to the database permanent and frees up all of the physical resources associated with the transaction. If COMMIT fails for some reason and returns an exception to the client, the transaction remains in an uncommitted state. The client application must handle the failure to commit by explicitly rolling back the transaction or, where possible, by fixing the problems and resubmitting the statement

COMMIT with the RETAIN option

The optional RETAIN [SNAPSHOT] extension to the COMMIT statement causes the server to retain a “snapshot” of the physical transaction's context at the time the statement is executed and start a new transaction as a clone of the committed one. It is referred to as a *soft commit*. If the soft commit is used on a SNAPSHOT or SNAPSHOT TABLE STABILITY transaction, the cloned transaction preserves the same snapshot of the data as the original transaction had when it started.

Although it does commit the work permanently and thus change the state of the database, COMMIT RETAIN (CommitRetaining) does not release resources. In the lifespan of a logical task that comprises many repetitions of a similar operation, cloning the context reduces some of the overhead that would be incurred by clearing resources each time with COMMIT, only to allocate the identical resources all over again when a new transaction is started. In particular, it preserves the cursors on sets selected and currently “open”.

In the TSD, the same TID remains active and never appears there as “committed”. That is why it is referred to as “soft” commit, in contrast with the “hard” commit performed by an unmodified COMMIT statement.

Each soft commit is like a savepoint with no return: any subsequent ROLLBACK reverses only the changes that have been posted since the last soft commit.

The benefit of the soft commit is that it makes life easy for programmers, especially those using components that implement “scrolling dataset” behavior. It was introduced to support the data grid user interface favored by many users of the Delphi® development environment. By retaining the transaction context, the application can display a seamless before-to-after transition that reduces the effort the programmer would otherwise need to invest in starting new transactions, opening new cursors and resynchronizing them with row sets buffered on the client.

Data access implementations frequently combine posting a single update, insert or delete statement with an immediate COMMIT RETAIN, in a mechanism that is dubbed “Autocommit”. It is common for interface layers that implement Autocommit capability to “dumb out” explicit control of transactions by starting one invisibly in situations where the programmer-written code attempts to pass a statement without first starting a transaction itself.

Explicit transaction control is worth the extra effort, especially if you are using a connectivity product that exploits the flexible options provided by Firebird. In a busy environment, the COMMIT RETAIN option can save time and resources but it has some serious disadvantages:

- A snapshot transaction continues to hold the original snapshot in its view, meaning the user does not see the effects of committed changes from other transactions that were pending at the start of the transaction
- As long as the same transaction continues being committed with RETAIN, resource “housekeeping” on the server is inhibited, resulting in excessive growth of memory resources consumed by the TSB. This growth progressively slows down performance, eventually freezing the server and, under adverse operating system conditions, even causing it to crash.
- No old record versions made obsolete by operations committed by a COMMIT RETAIN transaction can be garbage-collected as long as the original transaction is never hard-committed.

The ROLLBACK statement

Like COMMIT, ROLLBACK frees resources on the server and ends the physical context of the transaction. However, the application’s view of database state reverts to the way it would be if the transaction had never started. Unlike COMMIT, it never fails.

The syntax of the ROLLBACK statement:

```
ROLLBACK [WORK] [RETAIN [SNAPSHOT]];
```

In the logical context of “the task” on the client, after rollback your application must provide the user with the means to resolve the problem that caused the exception and to try again in a new transaction.

ROLLBACK with the RETAIN option

Firebird implements a RETAIN syntax for ROLLBACK from the “2” series onward. For all versions, a similar cloning mechanism is implemented at the API with the function *isc_rollback_retaining()*. It restores the application’s database view to the state it was in when the transaction handle was acquired or, in a ReadCommitted transaction, to the state it was in the last time **ROLLBACK RETAIN** was called. System resources allocated to the transaction are not released and cursors are retained. ROLLBACK RETAIN has the same traps as COMMIT RETAIN—and one more. Since any rollback call is done in response to an exception of some sort, retaining the context of the transaction will also retain the cause of the exception. ROLLBACK RETAIN should not be used if there is any chance that your subsequent exception-handling code will not find and fix the inherent exception. Any failure subsequent to ROLLBACK RETAIN should be handled with a full rollback, to release the resources and clear the problem.

Diagnosing exceptions

Common causes for failure to post or commit work include

- lock conflicts
- bad expressions that slipped through the user interface—resulting in arithmetic overflows, expressions that divide by zero, nulls in non-nullable fields, mismatched character sets and so on
- data that fails a CHECK constraint or other validation
- primary and foreign key violations
- and others!

Firebird recognizes a comprehensive (nay, mind-boggling!) range of exceptions and provides error codes to identify them at two levels.

- at the higher-level is the SQLCODE, defined (more or less, with emphasis on “less”) by the SQL standards or (from v.2.5) the SQLSTATE code, which is standards-compliant
- at the more detailed level is the GDSCODE, a larger and more precisely targeted, vendor-specific range of exception types that are grouped beneath the SQLCODE types (but not, unfortunately, beneath the SQLSTATE types)

SQLCODE

Table 27.2 shows the values that the SQL-89 and SQL-92 standards define for SQLCODE.

Table 27.2 SQLCODE values and definitions

SQLCODE	MESSAGE	Interpretation
0	SUCCESS	The operation completed successfully
1 – 99	SQLWARNING	Warning or information message from the server

SQLCODE	MESSAGE	Interpretation
100	NOT FOUND	An operation was requested for which no rows were found. This is not an error—it often just means that an end-of-file or end-of-cursor condition was detected or simply that no matching value was found
< 0	SQLERROR	Indicates the SQL statement did not complete. The numbers fall into broken ranges between –1 and –999

The mapping of negative SQLCODEs to actual errors is not defined by the standards. Firebird's negative SQLCODEs are quite generalized and provide groupings that are largely coincidental and sometimes bemusing. For example, while it is possible to guess that an SQLCODE of –204 means “something unknown”, the code on its own tells nothing about what is unknown.

SQLSTATE completion code

The SQL-2003 standard 5-alphanumeric SQLSTATE completion code was introduced in v.2.5 and will eventually replace the SQLCODE. The codes with their descriptions and, where known, their approximate SQLCODE equivalents, are listed in Appendix VIII. An API function, *fb_sqlstate()*, converts these codes to the corresponding GDSCODE when the error array is returned to the client.



The SQLCODE was signalled as “deprecated” with the v.2.5 release. It still works, at least for the 2.x releases. GDSCODE (also sometimes delivered as “ISC CODE”) will remain.

GDSCODE

The second-level GDSCODEs provide much better targeting of the actual exception. Each GDSCODE is a signed integer that is mapped to a constant in *iberror.h* (in your */include* subdirectory) and also to a message text string in *firebird.msg*⁶. As a rule, the GDSCODE message is quite precise about what occurred. It provides a highly useful diagnostic mechanism for applications and you can map the associated symbolic constants to custom messages in your own host-code module, for use by exception handlers.

The GDSCODE codes, symbols and English descriptions, in their SQLCODE groupings, are listed in Appendix VII.

Receiving exceptions

In the following example, an application makes an attempt to insert a row into a table that doesn't exist:

```
INSERT INTO NON_EXISTENT (TEST)
VALUES ('ABCDEF');
```

The following set of error information is returned to the application (for this example, the *IB_SQL* admin utility):

```
ISC ERROR CODE:335544569 <- GDSCODE
Dynamic SQL Error <- corresponding text from firebird.msg
SQL error code = -204 <- SQLCODE
Table unknown <- corresponding text from firebird.msg
```

6. *ibase.msg* if you are using v.1.0.x

NON_EXISTENT

Where does the application get the error codes and messages from? The answer is found in the error status vector, an array that is passed as a parameter in most of the API functions. These functions return status and error codes to the client, along with the corresponding strings from the firebird message file. The API also provides client applications with utility functions to interpret the contents of the error status vectors and pass them into local buffers. Exception handlers can then parse the contents of these buffers and use the information to decide what to do about the exception and to deliver a friendly message to the user.

iberror.h

The header file `iberror.h`, in your `/include` directory, contains the declarations that associate each error code with a constant. For example, here are the constant declarations for the two error codes from the preceding example:

```
...
#define isc_dsql_error          335544569L
...
#define isc_dsql_relation_err  335544580L <- an SQLCODE -204 error
```

Most of the established high-level language and scripting host interfaces already have translations of the constant declarations, although some pre-date Firebird and may not be open source. If you need a translation, it is recommended that you inquire in the support lists. The use of these error codes and language extensions for exploiting them in PSQL is the subject of Chapter 32, *Error Handling and Events*.

Multi-database Transactions

Firebird supports operations across multiple databases under the control of a single transaction. It implements two-phase commit (“2PC”) automatically, to ensure that the transaction will not commit the work in one database unless it is possible to commit it in the others. Data are never left partly updated.

In the first phase of a two-phase commit or rollback, Firebird prepares to commit (or roll back) the work for each database by splitting the transaction into sub-transactions, one for each database, and posting the changes to each. At this point, the sub-transactions are all in a transient state. If the first phase completes, then the second phase flags each sub-transaction for committing or rolling back, as the case may be, in the order in which the parts were prepared.

- If it is a commit and any sub-transaction can not be committed, an exception occurs. Any sub-transactions so far flagged for commit revert to 'transient' and database state is unchanged in all cases.
- If the commit succeeds throughout, then all sub-transactions go into 'committed' state and changes become permanent.
- If it is a rollback, the sub-transactions go into rolled back state

Limbo transactions

If network interruption or a disk crash makes one or more databases unavailable, causing the two-phase commit to fail during the second phase, sub-transactions left behind will be in a transient state, flagged as neither committed nor rolled back. Within each individual database, these sub-transactions that never completed the second phase (became committed or rolled back) are recognized as being in a “limbo” state.

Because rows in a database sometimes become inaccessible because of their association with a transaction that is “in limbo”, it is important to resolve these transactions.

Resolving limbo transactions

Until a limbo transaction is finished (by being committed or rolled back) it remains “interesting” to Firebird, which keeps statistics on unfinished transactions. Resolving a limbo transaction involves committing it or rolling it back. The *gfx* tool can recover limbo transactions and let you deal with them interactively. For more information, refer to [Transaction recovery](#) in Chapter 35, [Configuring and Managing Databases](#).



Applications that don't do multi-database operations never exhibit limbo transactions.

Restricting databases

Cross-database transactions can use a lot of server resources. DSQL does not provide SQL language support for restricting the databases that can be accessed by the transaction. DSQL interfaces can use special API structures in the transaction parameter block (TPB) for limiting multi-database transactions in various ways. Some data access component classes provide access to these options through properties or members.

Pessimistic Locking

In a pessimistic locking DBMS, rows that have been requested by one user or transaction for an operation that could potentially change data state become immediately inaccessible for reading or writing by other users or transactions. In some systems, the entire table becomes unavailable. Many developers moving databases and applications to Firebird from such systems are disconcerted by optimistic locking and search desperately for ways to mimic the old.

In Firebird, all updates are at row level—there is no mechanism for application code to lock an individual column. At almost all levels of transaction isolation, the engine employs an optimistic locking principle: all transactions not constrained by some form of pessimistic locking begin with a view of the currently committed state of all rows in all tables—subject to privileges, of course. When a transaction submits a request to update a row, the old version of that row remains visible to all transactions. Writers do not usually block readers.

Internally, upon a successful update request, the engine creates a new version of the row that implicitly locks the original version. Depending on the settings of that user's transaction and others, some level of lock conflict will occur if another transaction

7. In embedded SQL (ESQL) Firebird provides language support in the form of the USING clause, for optionally limiting the databases a transaction is permitted to access.

attempts to update or delete the locked row. For more information about Firebird's locking conditions, refer to the previous chapter.

Firebird is designed for interactive use by many concurrent users and there are seldom any genuine reasons to use a pessimistic lock. It should never be regarded as a magic bullet to be used to emulate the locking behavior of a desktop DBMS. Pessimistic locks by one transaction will cause conflicts for other transactions. There is no escape from the responsibility to work in sympathy with the multi-user transaction model and write handlers to anticipate locking conflicts.

Table-level locking

The transaction isolation level `TABLE STABILITY`, a.k.a. “consistency isolation”, provides full, table write-locking that flows on to dependent tables. It is too aggressive for interactive applications.

It is preferable to use a `RESERVING` clause with `READ COMMITTED` or `SNAPSHOT` isolation because it offers more flexibility and control for targeting tables that you want to lock for the duration of a transaction. It has parameters that determine how much protection is requested for each table:

```
RESERVING <reserving_clause>;
<reserving_clause> = table [, table ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

The `<reserving-clause>` can comprise multiple sets of reservation specifications, enabling different tables or groups of tables to be reserved with different rights. A specific table should appear only once in the entire reserving clause. Refer to the previous chapter for information about table reservation.

Statement-level locking

Pessimistic locking at statement level—affecting individual rows or sets—is not directly an issue of transaction configuration. However, the effects of these lock types are directly governed by the transaction settings of both the transaction in which the locking is defined and other transactions that try to access the locked row or set. It is essential to consider these techniques in terms of transaction settings.

Pessimistic locking of a row or a set is antithetical to the way Firebird was designed to work. In short, if you have lived with and depended on pessimistic locking until you discovered Firebird, then the time has come for you to get into the MGA groove and enjoy the benefits of optimistic row locking.

However, it can not be denied that design requirements do occasionally call for a pessimistic lock, though the need in Firebird is much rarer than your experiences with other data storage systems may have taught you. The likely scenario involves an absolute requirement for exclusive access to rows that, notwithstanding, does not justify a table-level lock. Typically, a row, once read, must be secured by that reader exclusively for an update or deletion. This requirement is sometimes referred to as *strict task serialization*.

For conditions where a row-level pessimistic lock is necessary, a pessimistic locking mechanism and supporting SQL syntax does exist⁸. But first, we look at the standard “hack” that client applications do—when language support for it is absent—to achieve a pessimistic lock. Next we examine the syntax and conditions for the explicit

8. ...although not in Firebird 1.0.x.

SELECT...WITH LOCK, that brought a degree of pessimistic locking support to Firebird SQL from v.1.5.

The “dummy update” hack

“Editing” data is not a server-side activity: hence, when the user clicks the Edit button—or whatever device your program uses to give her an editing interface—it changes nothing on the server. As far as the server is concerned, the transaction is merely reading. No new record version is created. There is no lock. Nothing changes until the user is finished editing the row and the application actually posts the user's work.

Developers who regard this behavior as a problem get around it by having their applications post a “dummy update” to a row as soon as the user asks to edit it. The hack is to post an update statement that sets a stable column to its current value. Usually, the primary key column is used. For example:

```
UPDATE ATABLE
  SET PKEY = PKEY
WHERE PKEY = PKEY;
```

Thus, the server creates a new record version, in which nothing is actually different to the latest committed version, and thenceforth blocks any other transaction from having write access to that row—effectively, it's a lock. Once the user signals that she has finished editing the row, the application posts the real update. For example:

```
UPDATE ATABLE
  SET COLUMN2 = 'Some new value',
    COLUMN3 = 99,
    ...
WHERE PKEY = <the value of the primary key>;
```

On the server, a second new record version is posted, overwriting the dummy one.

If you really need a pessimistic lock and there is no combination of transaction attributes that suits your special need, this technique is effective. It works on a single row only and lasts until the transaction is committed, even if the user decides not to perform any actual change to the row.



If you use this technique, make certain that you condition any BEFORE UPDATE and BEFORE DELETE triggers on the tables you subject to the dummy update, to preclude any possibility that they will fire.



It may be necessary to create a special, hidden FLAG column in your table for use specifically to flag dummy updates. For example, a hidden integer column could be incremented by your dummy update statement. Triggers would then be set up to perform “real” updates only IF (NEW.FLAG <> OLD.FLAG).

About “double-dipping” updates

It's not recommended to update the same row more than once in a single transaction, because it will interfere with the integrity of both automatic (referential integrity) and custom triggers. Where savepoints are involved, it causes record versions to proliferate.

This “double-dipping” generally reflects careless application design logic. However, a double-dipped update is exactly the intent of the “dummy update” technique. As long as the use of savepoints is avoided in these transactions and triggers are carefully conditioned to test for actual changes, it can be rendered harmless.

Explicit locking

Syntax The syntax pattern for optionally utilising the row-level explicit (pessimistic) locking is:

```
SELECT output-specification FROM table-name
[WHERE search-condition]
[FOR UPDATE [OF col1 [, col2 [,...]]]] WITH LOCK;
```

How it works

As the engine considers, in turn, each record falling under an explicit lock statement, it returns either the record version that is the most currently committed, regardless of database state when the statement was submitted, or an exception.

WAIT behaviour and conflict reporting depend on the transaction parameters specified in the TPB block.

The engine guarantees that all records returned by an explicit lock statement are actually locked and meet the search conditions specified in WHERE clause, as long no search condition depends on any other tables, for example through a join or subquery. It also guarantees to lock only rows that meet the search conditions.

However, the potential set is subject to the settings of its transaction— isolation level, lock resolution and record version.

A row does not get its lock until it is actually output to the server's row buffer.

There is no guarantee that the potential set will or will not be affected by parallel transactions that commit during the course of the locking statement's execution, allowing further rows to be eligible for selection by your transaction.

Refer to Table 273 (below) for a summary of interactions between transaction settings and explicit locks. ThisTransaction = the transaction that has, or is trying to get an explicit lock on a row or set.

Table 27.3 Interaction of transaction settings and explicit locks

Isolation	Resolution Policy	Behaviour
isc_tpb_consistency (SNAPSHOT TABLE STABILITY)	Not considered	Ignored. Table-level locks override explicit locks.
isc_tpb_concurrency (SNAPSHOT)	isc_tpb_nowait (NO WAIT)	If a row gets modified by any transaction that was committed since ThisTransaction started, or an already active transaction modified the row before it was output to the row cache, an update conflict exception is raised immediately

Isolation	Resolution Policy	Behaviour
isc_tpb_concurrency (SNAPSHOT)	isc_tpb_wait (WAIT)	If a row gets modified by any transaction that was committed since ThisTransaction started, or an already active transaction modified the row before it was output to the row cache, an update conflict exception is raised immediately. If an active transaction is holding the row with an explicit lock or a normal write lock, ThisTransaction waits for the outcome of the blocking transaction. If the blocking transaction then commits a modified version of this record, an update conflict exception will be raised.
isc_tpb_read_committed (READ COMMITTED)	isc_tpb_nowait (NO WAIT)	If an active transaction is holding the row with an explicit lock or a normal write lock, ThisTransaction gets an update conflict exception immediately.
ditto	isc_tpb_wait (WAIT)	If an active transaction is holding the row with an explicit lock or a normal write lock, ThisTransaction waits for the outcome of blocking transaction. When the blocking transaction completes, ThisTransaction attempts to get the lock on the row again. Update conflict exceptions can never be raised by an explicit lock statement with this configuration.

If a `SELECT...WITH LOCK` clause succeeds and the optional `FOR UPDATE` sub-clause is omitted, all of the rows in the set are pre-emptively locked, whether you actually update them or not. With careful transaction configuration and client-side control of buffering, the lock will prevent any other transaction from obtaining write access to any of those rows, or their dependants, until your transaction ends.

Pre-emptively locking a multi-row set will cause deadlocks to escalate and your application code must be ready to manage them.

The number of rows in the specified output set has important consequences if you use an access method that requests “datasets” or “recordsets” in packets of a few hundred rows at a time (“buffered fetches”) and buffers them in the client, typically to implement a scrolling interface. If the lock fails upon fetching a certain row and causes an exception, none of the rows currently in waiting in the server’s buffer will be sent and those already passed to the client buffer become invalid. Your application will have to roll back the transaction.

With this style of access, it is essential to provide your applications with a way to handle exceptions as they occur. Use a very strict `WHERE` clause to limit the range of the lock to one or a very small set of rows and avoid having partly-fetched sets made invalid.

Example The request

```
SELECT * FROM DOCUMENT
WHERE ID = ? WITH LOCK /* ID is the primary key */
```

ensures that the result will be exactly one row, either locked or refused.

If the set you want cannot be guaranteed to be a single row and your data access interface supports it, set the data access component’s fetch buffer to one row.

Not all data access layer provide the means to control the contents of the fetch buffer, however. The optional FOR UPDATE clause provides a way to define a multi-row set and fetch and process rows one at a time.

The FOR UPDATE clause

If the FOR UPDATE clause is included, buffered fetching will be disabled and the lock will be applied to each row, one by one, as it is fetched into the server-side row cache. In a stored procedure or executable block, where you have a named cursor controlling the position of the update, the clause can take an optional OF <column-list> to target updates at specific cursor columns.

Because the normal isolation rules apply to the transaction, it is possible for a lock that was available at the start of the request to fail subsequently. Unfetched rows remain “clear” and available for other transactions to obtain for update, leaving a “moving window” in which any unfetched row may become locked by another transaction, even if the lock appeared to succeed when the set was requested.

Example This statement defines an unrestricted multi-row output set and causes each row to be fetched into the server-side buffer individually. It will not fetch the next row until the client signals that it is ready. WITH LOCK causes the pessimistic lock to be attempted upon each row request. It will return either the next row in the set or an exception.

```
SELECT * FROM DOCUMENT
WHERE PARENT_ID=? FOR UPDATE WITH LOCK
```

Restrictions on explicit locking

The SELECT...WITH LOCK construct is available in DSQL and PSQL. It can succeed only in a top-level, single-table SELECT statement.

- It is not available in a subquery or a joined set.
- It cannot be specified with set quantifier (the DISTINCT operator, FIRST, SKIP or ROWS), a GROUP BY clause, or any other aggregating operation.
- It cannot be used in or with a view, a derived table, a CTE, an external table or the output set of a selectable stored procedure.

Stored Procedures, Triggers and Transactions

In this topic, we touch on aspects of transaction behaviour that affect procedural (PSQL) code.

For information about writing and using stored procedures, executable blocks and triggers, read on to Part Six, *Programming on the Server*.

Stored Procedures

Stored procedures execute in the context of the transaction that calls them. Work done, including any tasks fulfilled in embedded and recursive calls, will only take effect if everything completes without error, with exceptions handled, and the work committed. If the resolution to an exception condition in a single operation is to roll back the transaction, then all work in the transaction is rolled back.

Triggers

Triggers fire inside the context of the DML statement that prompts them. Work done, including any tasks fulfilled in embedded procedure calls, updates, inserts or deletes to other tables or through internal referential integrity or other triggers belonging to any of the tables will be committed in entirety or returned to the client in an unresolved state.

Unhandled exceptions in one operation will abort the operation where the error occurred and leave the transaction in a state where the application can decide to roll back or to attempt to fix the error and resubmit the request. Rollback will undo all of the operations executed in the transaction up to the point where the exception occurred.

“Savepoints” in PSQL

User savepoints allow the scope of rollbacks to be controlled from the application. PSQL has always had this capability on its side of the connection, by conditioning the workflow through use of *exceptions*. For details, refer to Chapter 32, **Error Handling and Events**.

Autonomous Transactions

In Firebird versions from 2.5 onward, PSQL code can launch a transaction to do a self-contained unit of work. It can be handy for a situation where you need to raise an exception but do not want the database changes to be rolled back.

The new transaction is initiated with the same isolation level as the one from which the client launched the request that prompted the PSQL to execute. The autonomous transaction itself cannot be controlled by the client at all. The engine will commit the changes made by the autonomous transaction if the block of code it controls succeeds in executing to its end. If any exception is raised within the block, the engine rolls back the changes it made up to the point of the exception.

Tips for Optimizing Transaction Behaviour

Some things that you might choose to do—or not to do—with respect to management of transactions in application or PSQL, “just because you can”, are recognised as sources of misbehaviour that can get in the way of efficiency and good housekeeping. It is as well to be aware of the traps and keep note of any tactics you employ that you later discard because they broke a flow or slowed down performance.

Following is a short list of tips and pieces of wisdom you are welcome to use to start your notebook.

Choose an appropriate transaction model

The single-transaction model tempts the inexperienced developer to ignore multi-user issues in favor of easier programming. The result is application architectures that perform poorly at all levels: slow queries and refresh responses, glutted networks, user-unfriendly work-flows and high levels of conflict.

Don't go “generic” unless you need to

Generic application-to-database interfaces, such as ODBC or the long-defunct BDE layers from old Borland products, merge a single database connection and a single transaction.

Because their purpose is to hide the differences between low-end, file-based data repositories and sophisticated transaction-capable database management systems, they do not support the capability to have multiple transactions concurrently active in a database session, or to have a transaction that spans connections to multiple databases.

At best, these generic interfaces provide an easy route for scaling up low-end databases or flattening out the differences between different vendors' DBMS implementations. If you don't need the lowest common denominator, don't choose it.

Exploit multiple transaction capabilities

A Firebird client can run multiple concurrent transactions. A user working on multiple tasks in a single application can perform a variety of activities over the same (or overlapping) sets of target data. Firebird's transaction model provides great benefits for designs that need to cater for a modular, multiple-tasking environment in a highly responsive manner. It is an important responsibility for the software engineer to develop techniques to keep commits flowing and user views synchronized with database state.

Keep the OAT moving!

A slow-moving OAT almost always indicates long-running transactions. Avoiding them is one of the best skills you can acquire when learning to write client applications for Firebird.

It is easy to blame long transactions on user behavior. You can help them to help themselves by teaching them to complete tasks within a reasonable time: not to go for coffee breaks leaving tasks unfinished, not to submit "wild queries" at peak times, and so on. However, good client application design avoids depending on users to behave well.

- Mechanisms that "time out" neglected transactions are effective.
- As a general rule, avoid browsing interfaces and favor the drill-down kind.
- If a browsing interface is unavoidable, isolate the statements that select the browsed data in READ-ONLY READ COMMITTED transactions
- Ensure that READ-WRITE transactions are committed regularly—even if users are only using them to view data.
- Restrict applications that permit random querying through vast sets, by enforcing the inclusion of WHERE clauses and timeout limits
- Make certain that your applications provide the means to perform periodic "hard commits" on any transactions that employ COMMIT RETAIN.
- Make it a rule to use RollbackRetaining not more than once in an exception handler. Don't put RollbackRetaining inside a loop!
- Understand what's going on inside transactions! Use *gstat -b* or equivalent tools to monitor the OIT and the OAT and pay attention to the "gaps".
- Don't allow housecleaning to be neglected. Sweeps must be allowed to happen at timely intervals and regular backups, even without restoring, will help to keep the transaction inventory in good shape.

SECOND EDITION

PART VI



Programming on the Server

CHAPTER 28

PROCEDURAL SQL—PSQL

One of the great benefits of a full-blooded SQL relational database implementation is its capability to compile and execute internal code modules—stored procedures and triggers—supplied as source code by the developer. The language that provides this capability on a Firebird server is PSQL—a simple but powerful set of SQL extensions that combines with regular data manipulation language (DML) statements to become compilable source modules.

Overview of Server Code Modules

The high-level language for Firebird server-side programming is SQL. Source code is presented to the engine in the form of SQL programming language extensions—PSQL statements and constructs—and DML statements. These statements are, themselves, wrapped inside single DDL statements of the form

```
{CREATE | RECREATE | ALTER} {TRIGGER | PROCEDURE} <name> ...
...
AS
...
BEGIN
<one or more blocks of statements>
END
```

The syntax for writing PSQL modules is explored in depth in the next chapters.

The objective of each of these “DDL super-statements” is to create and store one executable code module (stored procedure or trigger) or to redefine (RECREATE or ALTER) an existing object. A DDL statement is also used to destroy (DROP) executable code objects.

EXECUTE BLOCK syntax, supported from the “2” series onward, is a dynamic SQL construct that is similar in form to the DDL-wrapped stored procedure definition. It

provides a way to means to present a block of executable PSQl code directly from an application, without storing it as a persistent procedure.

PSQL supports the three data manipulation statements: INSERT, UPDATE, DELETE and the ability to SELECT single-row or multiple-row sets of data items into local variables. The PSQL extensions provide the following language and logic support:

- local variables and assignment statements
- conditional control-flow statements
- special context variables (for triggers only) for accessing the old and new values of each column of all DML input sets
- posting of user-defined database events to a listening client
- exceptions, including user-defined ones declared as database objects, and (in v.1.5) context-specific exception messages, and supporting structure and syntax for error handling
- input and output arguments (for stored procedures only)
- encapsulation of cursor behavior in FOR SELECT...INTO...DO looping syntax
- the SUSPEND statement (for stored procedures only), providing the capability to write stored procedures that output a virtual table in direct response to a SELECT statement—selectable stored procedures
- embedding of stored procedure calls in both stored procedures and triggers
- execution of external dynamic statements composed from within a PSQL module and, from v.2.5 onward, to connect to another database and execute the statement there
- ability to run a self-contained transaction (“autonomous transaction”) from within a PSQL module, also first supported in v.2.5
- ability to define multiple triggers for the BEFORE and AFTER phases of triggers for each DML event and to position them in a predefined execution order. V.1.5 introduced the ability to write conditional BEFORE and AFTER triggers encompassing any or all possible DML events.
- ability to define triggers that fire once per session or once per transaction, commonly (but wrongly) referred to as “database triggers”

Except for some specific elements, the entire PSQL language set is available to both stored procedures and triggers.

About Stored Procedures

Stored procedures can be used in applications in a variety of ways.

- Selectable procedures are used in place of a table or view in a SELECT statement.
- Executable procedures are used with an EXECUTE PROCEDURE statement to perform a single operation or start a set of operations on the server side.
- A stored procedure can be invoked by another stored procedure or by a trigger. It can call itself recursively.

All stored procedures are defined with the complex DDL statement CREATE PROCEDURE. Executable and selectable stored procedure declarations follow the same syntax rules: optional language elements distinguish a selectable one from an executable

one. One procedure can be nested within another, each performing a part of an atomic sequence of work that will be committed by the client application as a whole or rolled back as a whole.

Benefits of using stored procedures

Modular design—All applications that access the same database share stored procedures, thus centralizing business rules, re-using code and reducing the size of the applications

Streamlined maintenance—When a procedure is modified, changes propagate automatically to all applications without the need for further recompiling on the application side, unless changes affect input or output argument sets.

Improved performance—Execution of complex processing is delegated to the server, reducing network traffic and the overhead of operating on external sets.

Architectural economy—Client applications can focus on capturing user input and managing interactive tasks while delegating complex data refinement and dependency management to a dedicated data management engine.

Extra functionality—Tricky accessing and massaging of data, that can not be achieved with regular SQL, usually can be managed with one or a suite of stored procedures.

About Relation Triggers

A trigger is a self-contained routine associated with a relation (table or view) that automatically performs an action when a row in the table or view is inserted, updated, or deleted.

A trigger is never called directly. Instead, when an application or user attempts to INSERT, UPDATE, or DELETE a row in a table, any triggers associated with that table and operation are automatically executed, or fired. Triggers can make use of exceptions and can trigger events. They can also call stored procedures.

Triggers are a powerful feature with a variety of uses. Among the ways that triggers can be used are:

- To make correlated updates when DML is performed on the table. For example, a trigger could insert records to an internal or external change log; an AFTER DELETE trigger could insert a row to a history table.
- To validate input data.
- To transform data, e.g. to automatically convert text input to upper case or to fetch an auto-incrementing key value from a generator.
- To notify applications of changes in the database using event alerters.
- To perform custom cascading referential integrity updates.
- To make a read-only view updatable. For details, see *[Making read-only views updatable](#)* in Chapter 23.

Triggers are stored as objects in a database, like stored procedures and exceptions. Once defined to be ACTIVE, they remain active until deactivated with ALTER TRIGGER or removed from the database with DROP TRIGGER.

Benefits of using triggers

- Automatic enforcement of data restrictions, to make sure users enter only valid values into columns.
- Reduced application maintenance, since changes to a trigger are automatically reflected in all applications that use the associated table without the need to recompile and relink.
- Automatic logging of changes to tables. An application can keep a running log of changes with a trigger that fires whenever a table is modified.
- Automatic notification of changes to the database with event alerters in triggers.

Triggers as an auto-increment mechanism

Triggers can be used in combination with generators to implement an auto-incrementing key. Examples appear in several places in this book and the technique is explained in detail in *Implementing Auto-Incrementing Keys* in Chapter 30.

Triggers and transactions

Triggers always operate within the context of a specific DML operation, as part of that operation and inside the transaction that makes the DML statement request. They are in no sense separated from the transaction, or from the operation that causes them to fire. If the transaction is rolled back, then any actions performed by triggers are also rolled back.

About Database Triggers

From Firebird 2.1 forward, triggers can be defined to fire events at higher levels than the DML actions on relation objects. Known somewhat misleadingly as database triggers, they can be defined to fire on the higher-level events ON CONNECT, ON DISCONNECT, ON TRANSACTION START, ON TRANSACTION COMMIT and ON TRANSACTION ROLLBACK.

About Executable Blocks

The SQL language extension EXECUTE BLOCK makes “dynamic PSQL” available to applications for execution at run-time. It has the effect of allowing a self-contained block of PSQL code to be executed in dynamic SQL as if it were a stored procedure. Input and output parameters can be declared, if required. Output parameters can be returned to SELECT expressions.

PSQL Language Extensions

The PSQL language extensions include the following language elements:

- BEGIN and END statements for defining code blocks, which can be nested
- DECLARE VARIABLE statements for declaring local variables
- FOR SELECT <select-specification> INTO <variable-list> DO encapsulating an SQL cursor, for looping through sets—loops can be nested
- WHILE loops

SUSPEND statement for sending a row of output to the row cache

IF...THEN and ELSE for branching logic

The EXCEPTION <declared exception name> statement for raising custom exceptions

Optional WHEN <exception condition> DO blocks for catching and handling exceptions

POST_EVENT <string> to pass notifications to the client

The IN AUTONOMOUS TRANSACTION construct wrapping a block of code for execution in a separate transaction that is committed regardless of the outcome of the surrounding transaction (v.2.5 and later)

EXECUTE STATEMENT statement for executing ad hoc DML and DDL statements from within the module (v.1.5 and later). From v.2.5 onward, an EXECUTE STATEMENT statement can be embedded in a self-contained transaction to execute code in another database.

Boolean context variables UPDATING, INSERTING and DELETING

Context variable ROW_COUNT to read the number of rows affected by a completed DML statement within the same block

Optional syntaxes for EXCEPTION: with no argument for re-raising exceptions and with an additional text argument for passing run-time information back to the client

Restrictions on PSQL

- Metadata object identifiers—such as the names of tables, columns, views or stored procedures—can not be passed to or returned from stored procedures in arguments.
- Trigger procedures can not accept or return arguments

Statement types not supported in PSQL

The following statement types are not supported in triggers or stored procedures:

- Data definition language statements, i.e. any beginning with the any of the keywords CREATE, RECREATE, ALTER or DROP; SET GENERATOR, DECLARE EXTERNAL FUNCTION, and DECLARE FILTER
 - It is possible to use EXECUTE STATEMENT in PSQL to process DDL externally. However, it is strongly not recommended.
 - To process batches of DDL statements in Firebird, write scripts and submit them to the *isql* utility, either interactively or by way of a shell command, *run* script, batch file or similar.
- Transaction control statements: SET TRANSACTION, COMMIT, COMMIT RETAIN, ROLLBACK, SAVEPOINT, RELEASE SAVEPOINT, ROLLBACK TO SAVEPOINT. Stored procedures and triggers always execute within an existing client transaction context.

Prior to v.2.5, Firebird does not support any form of embedded transaction. From v.2.5 on, it is possible to embed an operation inside an autonomous transaction that is committed immediately, before returning control to the main flow. The committed work will survive even if the surrounding transaction is rolled back.
- Some other statement types that are reserved for use in different environments, e.g. isql, scripts or embedded SQL—such as the ESQL statements PREPARE, DESCRIBE and

EXECUTE et al., and *isql* commands starting with keywords SET or SHOW. All dynamic DML statements are allowed.

- CONNECT/DISCONNECT, and sending SQL statements to another database
- GRANT/REVOKE

Exceptions

An unhandled exception causes execution to stop. Any work done thus far is undone and an error message is returned to the application. If the code module is a trigger, the DML operation in which the error occurred will be undone also.

You can also have your code raise a custom exception itself and stop processing. You can handle the error in your code—or stop the processing and return a custom message to the client application. You can create as many custom exceptions as you need in a database. In any version except v.1.0.x, you can use run-time data and construct extensions to your exception messages “on-the-fly”.

The code module can handle an error itself with an optional piece of code—known as an exception block—which is a sequence of statements bounded by BEGIN and END, preceded by a directive beginning with the keyword WHEN. Exception handlers can be written to intercept an error and “swallow” it by dealing with it in some way. For example, an input row in an iterative routing that causes an exception does not need to cause the entire process to stop. The exception handling inside the trigger or procedure can allow the problem input to be skipped—perhaps logging the error in a text file or error table—and let further processing continue.

Exceptions and error handling are discussed in detail in Chapter 32.

Events

Firebird events are optional “signals” that PSQL modules can accumulate during execution, to be passed to client applications once the work has been committed. The signal is marked by the PSQL statement POST_EVENT. Client applications anywhere on the network can optionally listen—by way of “event alerters”—for specific events that they are interested in, without needing to poll for changes specifically.

Programming for events and setting up applications to listen for them are covered in the second part of Chapter 32.

Security

Procedures and triggers can be granted privileges for specific actions (SELECT, INSERT, DELETE and so on) on tables, just as users or roles can be granted privileges. There is no special syntax: an ordinary GRANT statement is used, but the recipient named in the TO clause is tagged as a trigger or procedure, instead of a user or a role. Similarly, privileges can be revoked from procedures and triggers.

It is not always necessary to grant privileges to trigger and procedure modules. It is enough for either the user or the module to have the privileges for the actions the module has to perform.

For example, if a user performs an UPDATE of table A, which fires a trigger, and the trigger performs an INSERT on table B, the action is allowed if the user has INSERT privileges on the table or the trigger has INSERT privileges on the table.

If there are not sufficient privileges for a trigger or procedure to perform its actions, Firebird fires an SQL error and sets the appropriate error code number. You can intercept this error code with an exception handler, just as with other exceptions.

For more information about GRANT and REVOKE, see Chapter 37, *Database-level Security*.

Elements of procedures and triggers

PSQL module definitions are really a single SQL statement that begins with a CREATE clause and ends with a terminator. Within the module definition are a number of elements—clauses, keywords, blocks of multiple statements, branches, loops and others. Some elements are mandatory; others are optional.

Although the complex definition of a PSQL module is a DDL statement, the SQL extensions within it are elements of a structured, high-level language that has certain distinctive rules. An important one to know about before you begin, especially if you are using *isql* to process the definitions, is the *statement terminator*.

Statement terminator

Each statement inside a stored procedure or trigger body—other than BEGIN and END—must be terminated by a semicolon. No other symbol is valid for terminating statements in PSQL. In DSQL, for both DML and DDL, the semi-colon also happens to be the default terminator for Firebird statements in many environments, including *isql*. It is also the SQL standard for terminating statements.

This situation is going to present a logical problem for the parser that compiles our PSQL modules—which semicolons terminate statements inside the module and which one terminates the CREATE definition?

To get around this problem, *isql* has a switching SET TERM syntax that allows you to set a different external terminator, to be in effect for external statements whilst PSQL definitions are being parsed. Many third-party SQL editors written for use with Firebird also implement this convention.

In scripts, experienced developers often use a single SET TERM statement at the beginning of all scripts, to have their favourite alternative terminator in effect at all times during scripting. Some database admin tools support configuring their editors and metadata extraction programs permanently with an alternative terminator so that there is no need to care about SET TERM at all.

isql pre-parses every statement and sends any terminated statement directly to the server as a single command. SET TERM being one of its own, non-SQL commands, it responds not by sending a request to the server but by preparing its parser to interpret terminators differently.

The DSQL layer does not recognize terminators for statements at all. Most of the other utilities that process scripts actually dispatch the DDL statements off to the server one-by-one without terminators. They provide parsing of their own to recognize the beginning and end of CREATE PROCEDURE statements and pass the internal semicolon

terminators simply as regular symbols within the compound statement syntax. Such utilities may throw exceptions at SET TERM, since it is not recognised as belonging to the SQL lexicon. However, in scripts, most utilities usually parse for and expect a SET TERM statement and use the alternative terminator internally, in a manner equivalent to the way *isql* handles it.

So—use SET TERM in *isql* if you are using that tool to process your CREATE PROCEDURE statements interactively; and use it in scripts.

The alternative terminator can be any string symbol you like, except a semicolon, a space character or a single-quote. If you use an alphabetic character, it will be case-sensitive. It can be multiple characters if you prefer, including embedded spaces, and it must not be a reserved keyword. Both of the following statements are valid:

```
SET TERM ^;
SET TERM boing! ;
```



It's wise not to get too creative with your terminator strings or you'll end up with a lot of awkward extra typing!

In PSQL definitions, use semicolons for all internal statements except BEGIN and END, and use the alternative terminator for the final END statement:

```
...
END ^
```

To return to “normal” statement termination, issue a second SET TERM statement that is the reverse of the first:

```
...
END ^
COMMIT ^
SET TERM ;^
```

The CREATE statement

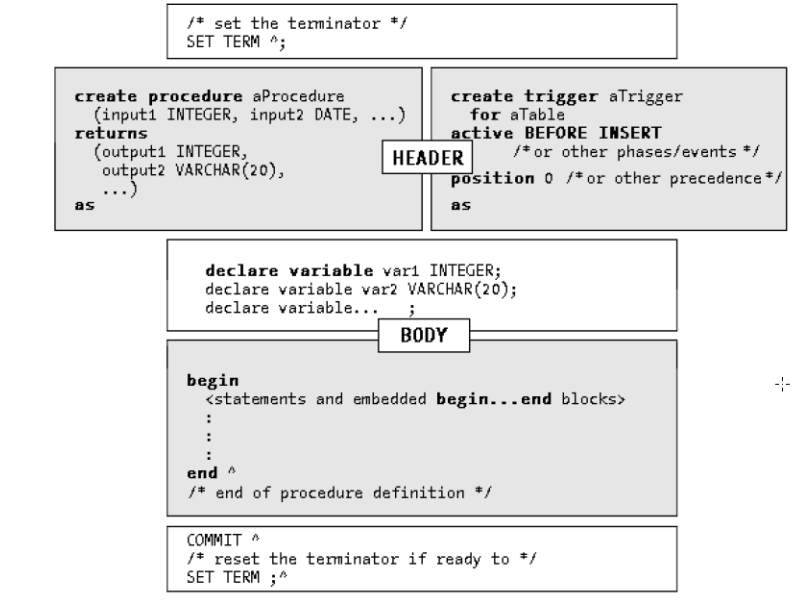
Source code for procedures and triggers is constructed inside a “super-statement” which begins with the keywords CREATE PROCEDURE or CREATE TRIGGER and ends with a terminator symbol following the final END statement. Each of these complex statements is composed of a header and a body.

For example,

```
CREATE PROCEDURE Name...
...
AS
...
BEGIN
...
END ^
```

With both stored procedures and triggers, all statements following the keyword AS comprise the local variables (if any) and the logic of the program module. The main difference between triggers and stored procedures is in the header portion of the CREATE statement—see Figure 28.1, below.

The main elements of a PSQL module definition are split to illustrate the required elements of the module's header and body sections. The mandatory parts are shaded.

Figure 28.1 Required elements of a PSQL module definition

Header elements

The **name of the procedure or trigger** must be unique in the database.

For a trigger:

- the keyword **FOR** and a table name, identifying the table or view that the trigger “belongs to”, i.e., the one that causes the trigger to fire when a data change occurs
- a **mode** (**ACTIVE** or **INACTIVE**)
- a **phase** parameter (**BEFORE** or **AFTER**) that determines when the trigger fires.
- an **event** parameter (**INSERT**, **UPDATE**, **DELETE**)
- optionally, the keyword **POSITION** followed by an integer, indicating **firing sequence**

For a stored procedure:

- An optional list of **input parameters** and their data types
- If the procedure returns values to the calling program, a list of **output parameters** and their data types

Body elements

For stored procedures and triggers:

- The module body can begin with a list of one or more **local variable declarations** (name and SQL data type. If you are using the “2” series or higher you can optionally use a domain in lieu of a data type)

- A **block of statements** in Firebird procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there may be many levels of nesting.
- Some embedded blocks can be **handlers for exceptions** occurring in preceding blocks. Such blocks are conditioned by a preceding WHEN predicate. Module-global exception handlers should follow all other embedded blocks.

Language elements

The following table shows the PSQL language elements available in Firebird:

Table 28.1 PSQL extensions for stored procedures and triggers

Statement	Description	Comment
BEGIN... END	Defines a block of statements that executes as one; the BEGIN reserved word starts the block, the END reserved word terminates it. Neither should be followed by a semicolon. In v.1.0.x it is not valid to submit a CREATE PROCEDURE statement without at least one statement between BEGIN and END.	“Empty” definitions are fine in v.1.5 and later.
variable = expression	Assigns the value of expression to variable, a local variable, input parameter, or output parameter	
/* comment-text */	Programmer’s comment, where comment-text can be any number of lines of text between a “/* */” pair.	Can also be used for in-line comments
-- comment text	One-line programmer’s comment, operational from the -- to end of line.	Not available in v.1.0.x.
EXCEPTION exception-name	Raises the named exception for optional handling by a WHEN block. Exception is pre-defined by the DBA using CREATE EXCEPTION	
EXCEPTION	Inside a handler block, re-raises the exception.	No arguments. Not available in v.1.0.x.
EXCEPTION exception-name run-time-message	Raises the named exception and attaches run-time-message, a varchar variable created and assigned a run-time message during the course of execution.	Not available in v.1.0.x. For details about defining and using exceptions, refer to Chapter 32, Error Handling and Events.
EXECUTE PROCEDURE proc-name [var [, var ...]] [RETURNING_VALUES var [, var ...]]	Executes stored procedure, proc-name. Input arguments follow the procedure name, return values are listed following the keyword RETURNING_VALUES. Enables procedure nesting and recursion. Input and output parameters must be variables defined within the procedure.	
EXECUTE STATEMENT <string>	Executes a dynamic SQL statement contained by <string>.	
EXIT	Jumps to the final END statement in the procedure.	Optional

Statement	Description	Comment
FOR..SELECT..INTO..DO..	Compound looping block syntax for processing an implicit cursor and optionally generating a virtual table for direct output to a SELECT request from a client. For more details, see FOR SELECT blocks (below).	
IF..THEN..[ELSE]..	Compound branching syntax. For more details, see Conditional blocks (below).	
LEAVE	Statement taking no parameters, for breaking out of loops. Causes execution to jump to the first statement following the end of the block that encloses the loop in which the LEAVE statement is executed.	Not available in v.1.0.x.
IN AUTONOMOUS TRANSACTION DO..	Wrapping for an auto-committing transaction that executes a block of code independently of the context of the transaction in which the surrounding module was invoked. The parameters of the embedded transaction are cloned from that transaction context. If the block executes without exception, that work is automatically committed. If an exception occurs within the embedded transaction, the embedded transaction is rolled back.	Not available prior to v.2.5.
NEW.column-name	Context variables available to INSERT and UPDATE triggers ¹ . There is one NEW variable for each column of the owning table, indicating the new value submitted by the client request.	Triggers only. Also available in some CHECK constraints.
OLD.column-name	Context variables available to UPDATE and DELETE triggers ² . There is one OLD variable for each column of the owning table, indicating the value as it was before the client request was submitted.	Triggers only. Also available in some CHECK constraints.
POST_EVENT event_name	Causes the event event_name to be “posted” to a stack. Event_name can be any string up to 78 characters and is not pre-defined on the server. Stacked events will be broadcast to clients listening via event alerter host routines.	For details, refer to Chapter 32, Error Handling and Events.
SELECT..INTO..	Passes output from an ordinary singleton SELECT specification into a list of pre-declared variables.	Throws an exception if the statement returns multiple rows.
SUSPEND	Statement used in procedures designed to output multiple-row sets as virtual tables—“selectable” stored procedures. It suspends execution of the procedure until the row is fetched from the row cache by the client application.	Not valid in triggers! It does not have this effect in an “executable” stored procedure, where it is equivalent to EXIT.

Statement	Description	Comment
WHILE <condition> DO	Conditional looping syntax that causes the succeeding block to be executed repeatedly until <condition> is false.	See Conditional blocks (below).
WHEN {error [, error ...] ANY}	Provides syntax for handling exceptions. Arguments can be one or more user-defined EXCEPTIONs or internally defined GDSCODE, SQLCODE or (v.2.5.1+) SQLSTATE exceptions.	

1. In multi-action triggers, reference to OLD.variables is valid, even though the trigger includes inserting action. It returns NULL if used in an inserting context.
2. In multi-action triggers, reference to NEW.variables is valid, even though the trigger includes deleting action. It returns NULL if used in a deleting context.

Programming constructs

The following topics examine the general programming constructs that are recognised in PSQL.

BEGIN...END blocks

PSQL is a structured language. Once variables are declared, the procedural statements are bounded by the keywords BEGIN and END. In the course of developing the logic of the procedure, other blocks can be embedded; and any block can embed another block bounded by a BEGIN...END pair.

No terminator symbol is used for the BEGIN or END keywords, except for the final END keyword that closes the procedure block and terminates the CREATE PROCEDURE or CREATE TRIGGER statement. This final END keyword takes the special terminator that was defined with the SET TERM statement before the definition began.

Conditional blocks

PSQL recognizes two types of conditional structures:

- branching, controlled by IF...THEN and, optionally, ELSE blocks
- looping through and executing a block repeatedly until a WHILE condition becomes false

The Boolean INSERTING, UPDATING and DELETING (for triggers only) and integer ROW_COUNT context variables are available for predicates within a block that performs a data state-changing operation. Refer to Chapter 30, *Triggers*, for details of using the Boolean context variables in multi-event triggers.

The IF...THEN...ELSE construct

The IF...THEN...ELSE construct branches to alternative courses of action by testing a specified condition. The syntax is:

```
IF (<condition>)
  THEN <compound_statement>;
```



```
[ELSE <compound_statement>;]
<compound_statement> = {<block>|<statement>;}
```

The condition clause is a predicate that must evaluate to true in order to execute the statement or block following THEN. The optional ELSE clause specifies an alternative statement or block to be executed if condition is false. Condition can be any valid predicate.



The predicate being tested by IF must be enclosed in brackets.

When you code conditional branching with SQL, use of the ELSE clause is sometimes a necessary “fall-back” for situations where the predicate tested by IF might not evaluate to either true or false. This could occur where run-time data conditions caused the IF predicate to compare two nulls and a logical result of true or false was impossible. An ELSE branch is the means for your code to guarantee that the block produces an outcome whenever the IF clause fails to do so.

The following code snippet illustrates the use of IF...THEN, assuming FIRST_NAME, LAST_NAME and LINE2, have been previously declared as variables or arguments:

```
...
IF (FIRST_NAME IS NOT NULL) THEN
    LINE2 = FIRST_NAME || ' ' || LAST_NAME;
ELSE
BEGIN
    IF (LAST_NAME IS NOT NULL) THEN
        LINE2 = LASTNAME;
    ELSE
        LINE2 = 'NO NAME SUPPLIED';
END
...
```



Pascal programmers—observe that the IF...THEN branch is terminated!

The WHILE...DO construct

WHILE...DO is a looping construct that repeats a statement or block of statements as long as the predicating condition is true. The condition is tested at the start of each loop. WHILE...DO uses the following syntax:

```
...
WHILE (<condition>) DO
BEGIN
    <execute one or more statements> ;
    <change the value of an operand of the predicating condition> ;
END
/* Continue execution here */
...
```



The condition being tested by WHILE must be enclosed in brackets.

In the following simple procedure, WHILE tests the value of a variable, *i*, that is initialized as an input argument. The looping block decrements *i* on each iteration and, as long as *i*

remains greater than zero, the value of the output argument, *r*, is raised by 1. When the procedure exits, the value of *r* is returned.

```
SET TERM ^;
CREATE PROCEDURE MORPH_ME (i INTEGER) RETURNS (r INTEGER)
AS
BEGIN
    r = 0;
    WHILE (i > 0) DO
    BEGIN
        r = r + i;
        i = i - 1;
    END
END^
```

Calling the procedure from isql:

```
SQL> EXECUTE PROCEDURE MORPH_ME(16);
```

we get

```
      R
=====
    136
```

The IN AUTONOMOUS TRANSACTION DO.. construct

For certain tasks, it is useful to be able to perform and commit an operation that does not depend on the successful completion of other tasks in a PSQL module. The introduction of the *autonomous transaction* construct in v.2.5 enables that. If the work in an autonomous transaction succeeds, it is committed automatically before control of execution returns to the surrounding module. Even if the surrounding module terminates subsequently with an exception and is rolled back, the work committed in the autonomous transaction survives.

Not all operations are good candidates for executing in autonomous transactions: for example, an attempt to operate on the same tables that are being accessed by the module or, indeed, by other statements executing in the same transaction, is almost certain to present a conflict exception to the autonomous transaction. If an autonomous transaction encounters an exception, it is rolled back with no other recourse to exception handling.

A handy use for an autonomous transaction is logging to a table of event history stored in the database. Prior to v.2.5, it is necessary to use an external table to write such messages, since any condition that results in rollback of the transaction would undo log records written to an internal table.

Transaction parameters

Although independent from the surrounding transaction, an autonomous transaction inherits its parameters, such as isolation level, timeout and so on.

Logging example

The following example, prepared by Paul Vinkenoog for the v.2.5 **Language Reference Update** document, illustrates how autonomous transactions could be helpful for the logging task mentioned above. It is used in an ON CONNECT trigger to keep a log of users' connection attempts:

```
CREATE TRIGGER tr_connect ON CONNECT
AS
```

```

BEGIN
-- ensure log message is preserved:
IN AUTONOMOUS TRANSACTION DO
    INSERT INTO log (msg) VALUES ('User ' || CURRENT_USER || ' connecting.');
```

IF (CURRENT_USER IN (SELECT username from blocked_users)) THEN

```

BEGIN
    -- preserve log message and post event to client:
    IN AUTONOMOUS TRANSACTION DO
    BEGIN
        INSERT INTO log (msg) values ('User ' || CURRENT_USER || ' refused.');
```

POST_EVENT 'Connection attempt by blocked user.';

```

    END
    -- return exception in surrounding transaction:
    EXCEPTION ex_baduser;
END
END
```

About CASE

PSQL does not yet support CASE logic as a programming construct. The DML CASE expression logic is, of course, available to PSQL. For more information, refer to [*Expressions Using CASE\(\) and Friends*](#) in Chapter 20, ***Expressions and Predicates***.

Variables and Parameters

Five types of variables can be used in the body of a code module, with some restrictions according to whether the module is a stored procedure or a trigger:

- local variables, used to hold values used only within the trigger or procedure.
- the NEW.ColumnName and OLD.ColumnName context variables, restricted to use in triggers, which store the new and old values of each column of the table when a DML operation is pending.
- other context variables—those specific to PSQL as well as those available to *isql* and DSQL
- input arguments, used to pass constant values to a stored procedure from a client application, another stored procedure or a trigger. Not available for triggers.
- output arguments, used to return values from a stored procedure back to the requester. Not available for triggers.

Any of these types of variables can be used in the body of a stored procedure where an expression can appear. They can be assigned a literal value, or assigned a value derived from queries or expression evaluations.

Local variables

Local variables are declared, one line per variable, preceding the first BEGIN statement. They have no effect outside the procedure or trigger and their scope does not extend to any procedures they call.

Variables must be declared before they can be used.

You should always ensure that your variables are initialized as early as possible in your procedure. To make it easier not to forget this important step, you can declare and initialize in a single statement (although not in v.1.0.x). For example, each of the following statements is valid to declare a counter variable and initialize it to 0:

```
...
DECLARE VARIABLE COUNTER1 INTEGER DEFAULT 0;
DECLARE VARIABLE COUNTER2 INTEGER = 0;
```

Example For an example using local variables, the following procedure is a piece of superstitious fun: assuming there is truth in the proverb “Never eat pork unless there is an ‘R’ in the month”, it returns an opinion about a given date. For the sake of illustration, it declares one local variable to get a starting condition for a WHILE loop and another to control the number of times the loop will execute:

```
CREATE PROCEDURE IS_PORK_SAFE(CHECK_MONTH DATE)
RETURNS (RESPONSE CHAR(3))
AS
DECLARE VARIABLE SMONTH VARCHAR(9);
DECLARE VARIABLE SI SMALLINT = 0; /* initialising this variable */
BEGIN
RESPONSE = 'NO ';
SELECT CASE (EXTRACT (MONTH FROM :CHECK_MONTH))
WHEN 1 THEN 'JANUARY'
WHEN 2 THEN 'FEBRUARY'
WHEN 3 THEN 'MARCH'
WHEN 4 THEN 'APRIL'
WHEN 5 THEN 'MAY'
WHEN 6 THEN 'JUNE'
WHEN 7 THEN 'JULY'
WHEN 8 THEN 'AUGUST'
WHEN 9 THEN 'SEPTEMBER'
WHEN 10 THEN 'OCTOBER'
WHEN 11 THEN 'NOVEMBER'
WHEN 12 THEN 'DECEMBER' END
FROM RDB$DATABASE
INTO :SMONTH;
WHILE (SI < 10) DO
BEGIN
SI = SI + 1;
IF (SUBSTRING(SMONTH FROM 1 FOR 1) = 'R') THEN
BEGIN
RESPONSE = 'YES';
LEAVE;
END
SMONTH = SUBSTRING(SMONTH FROM 2);
END
END ^
COMMIT ^
SET TERM ;^
```

Is it safe for the author to eat pork on her birthday?

```
EXECUTE PROCEDURE IS_PORK_SAFE ('2014-05-16');
```

```
RESPONSE
```

```
=====
```

```
NO
```



If this were a serious procedure, SQL has faster ways to get this result. For example, instead of the WHILE loop, you could simply test the variable SMONTH: IF (SMONTH CONTAINING 'R') THEN RESPONSE = 'YES' ELSE RESPONSE = 'NO '; You could probably find an external function to get the name of the month, too.

Input arguments

Input arguments (also known as *parameters*) are used to pass values from an application to a procedure or from one PSQL module to another. They are declared in a comma-separated list in parentheses following the procedure name. Once declared, they can be used in the procedure body anywhere an expression can appear.

For example, the following procedure snippet specifies one input argument, to tell the procedure which country the caller wants to be used in a search.

```
CREATE PROCEDURE SHOW_JOBS_FOR_COUNTRY (  
    COUNTRY VARCHAR(15))
```

```
...
```

Input parameters are passed by value from the calling program to a stored procedure. This means that if the procedure changes the value of an input parameter, the change has effect only within the procedure. When control returns to the calling program, the input parameter still has its original value.

Input arguments are not valid in triggers.

Output arguments

An output argument (parameter) is used to specify a value that is to be returned from a procedure to the calling application or PSQL module. Declare the arguments in parentheses and, if there are to be multiple return values, place the declarations in a comma-separated list. The output declarations follow the keyword RETURNS in the procedure header. Once declared, they can be used in the procedure body anywhere an expression can appear.

The following code completes the procedure definition shown in the previous snippet. It defines three items of data to be returned to the caller as a virtual table:

```
CREATE PROCEDURE SHOW_JOBS_FOR_COUNTRY (  
    COUNTRY VARCHAR(15))  
RETURNS (  
    CODE VARCHAR(11),  
    TITLE VARCHAR(25),  
    GRADE SMALLINT)  
AS  
BEGIN  
    FOR SELECT JOB_CODE, JOB_TITLE, JOB_GRADE FROM job  
    WHERE JOB_COUNTRY = :COUNTRY  
    INTO :CODE, :TITLE, :GRADE  
    DO  
    BEGIN /* begin the loop */
```

```

CODE = 'CODE: ' || CODE; /* mess about with the value a little */
SUSPEND; /* this outputs one row per loop */
END
END ^

```

If you declare output parameters in the procedure header, the procedure must assign values to them to return to the calling application. Values can be derived from any valid expression in the procedure.



Always initialize output parameters before beginning to process the data which will be sent to them.

Use of domains and existing column definitions

If you are using v.2.1 or higher, domains can be substituted for native SQL data types for declaring local variables and stored procedure input and output arguments. From v.2.5 onward, there is also the option to use a reference to a database column's type definition.

Using a domain

You have two choices as to how to use the domain definition:

- the domain identifier alone, to have the variable inherit all of the attributes (such as CHECK constraints and the DEFAULT value) of the domain
- the phrase TYPE OF <domain-name> to use the data type of the domain, without inheriting the attributes, other than the character set and collation if the domain is a character type.

Examples

```

CREATE DOMAIN DOM AS INTEGER;
CREATE PROCEDURE SP (
    I1 TYPE OF DOM,
    I2 DOM)
RETURNS (
    O1 TYPE OF DOM,
    O2 DOM)
AS
    DECLARE VARIABLE V1 TYPE OF DOM;
    DECLARE VARIABLE V2 DOM;
BEGIN
    ...
END

```



It would be wise to keep documentation on hand of any PSQL modules that use this feature. An ALTER DOMAIN operation has the potential to invalidate the precompiled code (BLR). The invalidation will be marked in the system tables but it will not trigger any kind of automatic recompilation of the PSQL. It will be up to the DBA to recompile and test any modules affected.

Using a reference to a column definition

From v.2.5 onward, the native SQL data type can be substituted by a fully qualified column name from a table or view somewhere in the same database. It requires the TYPE OF directive, analogous with its usage for using domains, viz.,

```
DECLARE VARIABLE MyVar1 TYPE OF aTable.aColumn;
```

The definition of the resulting variables in either usage will be the same as that of the domain or column referred to—except that it will not inherit any CHECK constraint, default value or NOT NULL constraint that was defined for the domain or column. However, a text type variable *does* inherit the character set and collation of the domain or column it refers to.

In versions prior to v.2.1, variables and arguments must be declared with native data types.

Extra attributes for variables

From the “2” series onward, options start to appear that enable you to include a variety of extra attributes in declarations of inputs, outputs and local variables.

Default values for inputs

From the “2” series onward, you can declare default values for stored procedure arguments. The syntax is similar to that for defining a default value for a column or domain, except that the PSQL assignment symbol (=) is used instead of the keyword DEFAULT.

Example

```
CREATE PROCEDURE MyProc (
    invar1 VARCHAR(20) COLLATE FR_CA,
    invar1 INTEGER = 123,
    invar2 DATE = CURRENT_DATE NOT NULL)
--
```

Rules for default inputs

Arguments with default values are strictly positional and must occur last in the argument list, i.e., after any arguments that are declared with no default. The caller is still required to supply values for all of the arguments that have no declared defaults and may omit assigning values to parameters where either

- all of the defaulted inputs are to use their defaults; or
- only the last defaulted input is to use its default

Substitution of default values occurs at run-time. Suppose you define a procedure P2 that has some default arguments and call it from another procedure P1, omitting some final, defaulted arguments. The default values for P2 will be substituted by the engine at the time P2 starts executing. The effect is that, if you change the default values for P2, it is not necessary to recompile P1.



The source and BLR for the argument defaults are stored in the system table RDB\$FIELDS.

COLLATE clause for text types

From v.2.1 you can include a COLLATE clause with any parameter or variable declaration that is a CHAR, VARCHAR or text BLOB type. Of course, it must name a collation that is available to the database for the character set of the data.

Example

```
CREATE PROCEDURE MyProc (
    invar1 VARCHAR(20) /* CHARACTER SET ISO8859_1 */ COLLATE FR_CA,
    invar1 INTEGER = 123,
    invar2 DATE = CURRENT_DATE)
--
```



It is possible to declare a character set for a variable in any version of Firebird. When including a COLLATE clause, it is not necessary to include the character set in the declaration unless it is different from the default character set for the database.

Non-nullable variables

NOT NULL can be included for inputs, outputs and local variables from v.2.1 onward. Use it with care—don't overlook the need to include adequate exception handling for blocks containing operations that have the potential to return NULL to non-nullable variables.

Example

```
CREATE PROCEDURE MyProc (
    invar1 VARCHAR(20) /* CHARACTER SET ISO8859_1 */ COLLATE FR_CA,
    invar1 INTEGER = 123,
    invar2 DATE = CURRENT_DATE NOT NULL)
--
```

Context Variables

Here we just look at the context variables that are particular to PSQL. All of the context variables described elsewhere—CURRENT_USER, CURRENT_TRANSACTION and so forth—are available in PSQL.

OLD.thing and NEW.thing

Triggers can use two complete sets of context variables representing the “old” and “new” values of each column in the owning table. **OLD.<column-name>** refers to the current or previous values in a row being updated or deleted. It has no meaning for inserts. **NEW.<column-name>** refers to the values submitted by the request to update or insert. It has no meaning for deletes. If an update does not change some columns, the NEW variable will have the same value as the OLD. Context variables are often used to compare the values of a column before and after it is modified.

Context variables can be used anywhere a regular variable can be used. NEW values for a row can only be altered by before actions. OLD values are read-only. For details and examples of usage, refer to Chapter 30, *Triggers*.



Since Firebird creates triggers to implement CHECK constraints, the OLD and NEW context variables can be used directly in a CHECK constraint, for example:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT EMPLOYEE_SALARY_RAISE_CK
CHECK ((OLD.SALARY IS NULL) OR (NEW.SALARY > OLD.SALARY));
```

INSERTING, UPDATING and DELETING

These three context variables are Booleans, available in triggers for use in multi-action triggers as condition testers. For details of usage, refer to *Special PSQL for Table-level Triggers* in Chapter 30.

ROW_COUNT

Returns an integer that is the number of rows affected by the most recent DML statement (INSERT, UPDATE, DELETE, SELECT or FETCH) that has executed inside the current module. It cannot be used to determine the number of rows touched by operations in calls to other procedures, although its value could be passed to a RETURNING_VALUES in the lower-level procedure.

It is available in triggers, stored procedures and executable blocks. Counts of rows affected by EXECUTE STATEMENT calls are not available.

For SELECT operations

- a FOR...SELECT loop returns the count of iterations
- a SELECT...INTO returns 1 if a row is returned, 0 otherwise. For a FETCH, the behaviour is similar—successive FETCH calls do not accrue to a total.



In v.1.5, a SELECT of any kind always returns 0 to ROW_COUNT.

Example

```
UPDATE BIKES
SET NUMBER_OF_WHEELS = 4
WHERE MODEL_TYPE = 'QUAD';
IF (ROWCOUNT > 0) THEN
  ANSWER = || ROW_COUNT || ' QUAD BIKES';
ELSE
  ANSWER = 'NO QUAD BIKES';
```

The colon (:) marker for variables

In SQL statements, prefix the variable's name with a colon character (:) whenever

- the variable is used in an SQL statement
- the variable is receiving a value from a [FOR] SELECT ... INTO construct

Omit the colon in all other situations.



Never prefix context variables with a colon.

Assignment statement

A procedure assigns values to variables with the syntax:

```
variable = expression;
```

The expression can be any valid combination of variables, operators, and expressions, and can include calls to external functions (UDFs) and SQL functions, including the GEN_ID() function or the NEXT .. FOR operator for stepping and/or returning a generator value.



NEXT .. FOR syntax does not support stepping.

Example

The following code snippet performs several assignments:

```
...
WHILE (SI < 9) DO
BEGIN
  SI = SI + 1; /* arithmetic expression */
  IF (SUBSTRING(SMONTH FROM 1 FOR 1) = 'R') THEN
  BEGIN
    RESPONSE = 'YES'; /* simple constant */
  LEAVE;
END
```

```

    SMONTH = SUBSTRING(SMONTH FROM 2); /* function expression */
END
...

```

Variables and arguments should be assigned values that do not conflict with the data type that they are declared to be. Numeric variables should be assigned numeric values, and character variables assigned character values. Although Firebird provides automatic type conversion in some cases, it is advisable to use explicit casting to avoid unanticipated type mismatches.

SELECT...INTO statements

Use a SELECT statement with an INTO clause to retrieve column values from tables store them into local variables or output arguments.

Singleton SELECTs

An ordinary SELECT statement in PSQL must return at most one row from the database—a standard *singleton* SELECT. An exception is thrown if the statement returns more than one row. An ORDER BY clause is not valid for a singleton select unless the SELECT statement is quantified by FIRST 1 or a ROWS 1 clause. (For information about the set quantifiers FIRST and ROWS, refer to [Optional set quantifiers](#) in Chapter 19, **DML Queries**.)

Normal rules apply to the input list and the WHERE clause and to the GROUP BY clause, if used. The INTO clause is required and must be the *last clause in the statement*.

For example, the following is a singleton SELECT statement in a parameterized DSQL query in an application:

```

SELECT SUM(BUDGET), AVG(BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept;

```

To use this statement in PSQL module, declare local variables or output arguments and add the INTO clause at the end as follows:

```

...
DECLARE VARIABLE TOT_BUDGET NUMERIC(18,2);
DECLARE VARIABLE AVG_BUDGET NUMERIC(18,2);
...
SELECT
    DEPARTMENT,
    SUM(BUDGET),
    AVG(BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
GROUP BY 1
INTO :department, :tot_budget, :avg_budget;

```

Multiple-row SELECTs

Any PSQL module can operate on multiple rows of input, acquired through a SELECT statement, as long as it provides a looping structure that can “move” through the set and

perform identical processing on each row. In the absence of looping logic context, a multi-row select will cause an exception (“Multiple rows in singleton select”).

FOR SELECT... Loops

The main method for implementing a looping structure for handling multi-row input sets is the FOR SELECT...INTO...DO structure. The abridged syntax is as follows:

```
FOR SELECT <set-specification-list>
FROM table-name
[JOIN..]
[WHERE..]
[GROUP BY..]
[ORDER BY..]
INTO <list-of-variables> DO
BEGIN
    <process-block>
    ...
    [SUSPEND];
END
...
```

As an example, the following procedure defines and acquires a set using a SELECT statement that brings the rows, one at a time, in the procedure's cursor buffer. It courses through the set, massaging each set of variables to fit a table specification. At the end of the loop, it inserts a record into the external table:

```
CREATE PROCEDURE PROJECT_MEMBERS
AS
    DECLARE VARIABLE PROJ_NAME CHAR(23);
    DECLARE VARIABLE EMP_NO CHAR(6);
    DECLARE VARIABLE LAST_NAME CHAR(23);
    DECLARE VARIABLE FIRST_NAME CHAR(18);
    DECLARE VARIABLE HIRE_DATE CHAR(12);
    DECLARE VARIABLE JOB_TITLE CHAR(27);
    DECLARE VARIABLE CRLF CHAR(2);
BEGIN
    CRLF = ASCII_CHAR(13)||ASCII_CHAR(10); /* Windows EOL */
    FOR SELECT DISTINCT
        P.PROJ_NAME,
        E.EMP_NO,
        E.LAST_NAME,
        E.FIRST_NAME,
        E.HIRE_DATE,
        J.JOB_TITLE
    FROM EMPLOYEE E
        JOIN JOB J ON E.JOB_CODE = J.JOB_CODE
        JOIN EMPLOYEE_PROJECT EP ON E.EMP_NO = EP.EMP_NO
        JOIN PROJECT P ON P.PROJ_ID = EP.PROJ_ID
    ORDER BY P.PROJ_NAME, E.LAST_NAME, E.FIRST_NAME
    INTO /* column variables */
```

```

:PROJ_NAME, :EMP_NO, :LAST_NAME, :FIRST_NAME,
:HIRE_DATE, :JOB_TITLE
DO
BEGIN /* starts the loop that massages the variable */
  PROJ_NAME = ''||CAST(PROJ_NAME AS CHAR(20))||''||',';
  EMP_NO = CAST(EMP_NO AS CHAR(5))||',';
  LAST_NAME = ''||CAST(LAST_NAME AS CHAR(20))||''||',';
  FIRST_NAME = ''||CAST(FIRST_NAME AS CHAR(15))||''||',';
  HIRE_DATE = CAST(HIRE_DATE AS CHAR(11))||',';
  JOB_TITLE = ''||CAST(JOB_TITLE AS CHAR(25))||'';
  INSERT INTO EXT_FILE
  VALUES (
    :PROJ_NAME, :EMP_NO, :LAST_NAME,
    :FIRST_NAME, :HIRE_DATE, :JOB_TITLE,
    :CRLF);
  END /* completes the DO-loop */
END ^

```



If an output parameter has not been assigned a value, its value is unpredictable, which can lead to errors. A procedure should ensure that all output parameters are initialized in advance of the processing that will assign values, to ensure that a SUSPEND will pass valid output.

SUSPEND

The SUSPEND statement has a specific use with the FOR..SELECT..INTO..DO construct just described. If SUSPEND is included in the DO loop, after the SELECT row has been read into the row variables, it causes the loop to wait until that row has been output to the server's row cache before fetching the next row from the SELECT cursor. This is the operation that enables Firebird's *selectable* stored procedures feature.

In the next chapter, we take a closer look at using SELECT statements that return multiple rows to a stored procedure, particularly the technique for writing selectable stored procedures.

The SUSPEND statement is not valid in triggers. In executable stored procedures it has the same effect as a EXIT statement, i.e. it terminates the procedure immediately and any statements following it will never execute.

By contrast, if a selectable procedure has executable statements following the last SUSPEND statement in the procedure, all of those statements are executed, even though no more rows are returned to the calling program. This style of procedure terminates with the final END statement.

Flow of control statements

PSQL provides a number of statements that effect the flow of control through code modules. The SUSPEND statement, just discussed, effectively passes control back to the calling procedure or client program, waiting for a just-processed row to be fetched from the server's row cache.

EXIT

In both select and executable procedures, EXIT causes program control to jump to the final END statement in the procedure. It is not meaningful in triggers.

The behaviours of SUSPEND, EXIT and END statements are summarized in Table 28.2 below

Table 28.2 SUSPEND, EXIT and END

Module type	SUSPEND	EXIT	END (Final)
Selectable procedure	Execution is suspended until the calling application or procedure fetches the preceding set of output variables.	Returns values (if any) and jumps to final END	Returns control to application and sets SQLCODE to 100
Executable procedure	Jumps to final END—not recommended	Jumps to final END	Returns values, returns control to application
Triggers	Never used	Jumps to final END	Causes execution of the next trigger in the same phase (BEFORE or AFTER) as the current one, if any; otherwise terminates trigger processing for the phase

LEAVE

The LEAVE statement is for breaking out of code blocks. It is SQL standard-compliant and deprecates the BREAK statement that is partly implemented in v.1.0.x. An example of its use is in the WHILE loop of our IS_PORK_SAFE procedure:

```
...
WHILE (SI < 10) DO
BEGIN
  SI = SI + 1;
  IF (SUBSTRING(SMONTH FROM 1 FOR 1) = 'R') THEN
  BEGIN
    RESPONSE = 'YES'; /* simple constant */
    LEAVE;
  END
  SMONTH = SUBSTRING(SMONTH FROM 2); /* function expression */
END
...
```

LEAVE causes execution to leave the loop—in this case, to stop testing the letters of a word for the character “R” and move on to the next statement after the END of the loop. If the branching containing the LEAVE statement is not executed, execution continues to the end of the loop.

Labelled loops

From the “2” series onward, you can label your loops and direct LEAVE to break its way through enclosing loops as well. Execution then continues from the statement following the END of the loop targeted by the LEAVE directive.

The syntax pattern is:

```
[label:]
{FOR | WHILE} ... DO
...
(possibly nested loops, with or without labels)
...
LEAVE [label];
```

Example The next example uses labels LOOP01 and LOOP02 to label an outer loop and a nested one, respectively.

LEAVE LOOP01 breaks execution out of the outer loop, LEAVE LOOP02 out of the inner loop. Using the label for breaking out of LOOP02 is optional, since LEAVE without a label breaks out of the current loop, anyway.

```
STATEMENT1 = 'SELECT MNGR_NO FROM DEPARTMENT';
LOOP01:
FOR EXECUTE STATEMENT :STATEMENT1 INTO :MANAGER
DO BEGIN
    STATEMENT2 = 'SELECT FULL_NAME FROM EMPLOYEE WHERE EMP_NO = ''';
    LOOP02:
    FOR EXECUTE STATEMENT :STATEMENT2 || :MANAGER || '' INTO :EMP
    DO BEGIN
        IF (EMP = 'Weston, K.J.') THEN LEAVE LOOP02
        ELSE IF (MANAGER = EMP) THEN LEAVE /* LOOP01 */;
        ELSE SUSPEND;
    END
END
```

v.1.0.x BREAK

BREAK immediately breaks execution out of a WHILE or FOR loop. Execution then continues from the first statement after the loop END. Its usage is parallel in form to the LEAVE statement. Although it is deprecated in all versions of Firebird after v.1.0.x, it will still work fine in later versions (up to v.2.5.x, at time of writing). It is not recommended to use it in new modules—future-proofing!

Example

```
...
WHILE (SI < 10) DO
BEGIN
    SI = SI + 1;
    IF (SUBSTRING(SMONTH FROM 1 FOR 1) = 'R') THEN
    BEGIN
        RESPONSE = 'YES'; /* simple constant */
        BREAK;
    END
    SMONTH = SUBSTRING(SMONTH FROM 2); /* function expression */
END
```

...

EXCEPTION

The **EXCEPTION** statement stops execution and passes control to the first exception handler block—a block starting with the keyword **WHEN**—that can handle the exception. If no handler is found for the exception, control passes to the final **END** statement and the procedure aborts. When this happens, one or more exception codes are passed back to the client via the error status vector (array).

The **EXCEPTION** statement is used in an **IF..THEN..ELSE** block to invoke custom exceptions that have been previously declared as database objects. The Firebird engine throws its own exceptions for SQL and context errors. Flow of control in these cases is the same as when a custom exception is called.

The syntax and techniques for calling and handling exceptions are addressed in Chapter 32, **Error Handling and Events**.

POST_EVENT

Firebird events provide a signaling mechanism which enables applications to listen for database changes made by concurrently running applications without the need applications to incur CPU cost or consume bandwidth to poll one another directly.

The statement syntax is

```
POST_EVENT event_name;
```

It causes the event **event_name** to be “posted” to a stack. **Event_name** can be any string up to 78 characters and is not pre-defined on the server. Stacked events will be broadcast to clients “listening” via event alterer routines in applications.

When the transaction commits, any events that occurred during triggers or stored procedures can be delivered to listening client applications. An application can then respond to the event by, for example, refreshing its open datasets upon being alerted to changes.



Because an event is not a metadata object, the list of possible events cannot be discovered by querying the system tables. It is entirely the responsibility of the application developer to know the texts of events of interest that are coded in modules.

For details about initiating and using events, refer to the second part of Chapter 32, **Error Handling and Events**.

Execute Statement

The “basics”

The PSQL extension **EXECUTE STATEMENT** executes a string containing a valid DSQL statement. An application or procedure can pass a DSQL statement (DML or DDL) as a string input argument to the procedure, or the procedure definition can construct one for it as a local variable within its own boundaries.

If the executing string is a **SELECT** statement, it can return values to variables, using the same **[FOR] SELECT... INTO <variables> [DO]** syntax that is used for a regular PSQL **SELECT** call.

EXECUTE STATEMENT adds a degree of extra flexibility to stored procedures and triggers but carries a high risk of errors that the DDL parser cannot detect. Although return values are strictly checked for data type in order to avoid unpredictable type-casting exceptions, the string itself cannot be parsed or validated at compile time.

In this chapter we look at the “basics” for processing strings with EXECUTE STATEMENT. Chapter 31, **Accessing Other Databases**, takes a detailed look at capabilities added in v.2.5, whereby an enhanced EXECUTE STATEMENT syntax can be used to execute DSQL statements in a different database inside a self-contained (“autonomous”) transaction.

Syntax The simplified syntax pattern is:

```
[FOR] EXECUTE STATEMENT <string>
[INTO :var1 [, :var2 [, :varN]]] DO
<compound-statement>;
```

The **<compound-statement>** represents what happens with any values returned by the executed string. It can be anything from a single statement to a block of statements embedding multiple statements and blocks.

The construction of the expression or string variable that is to form the DSQL statement string argument must be complete at the time the EXECUTE STATEMENT is executed.

In versions prior to v.2.5, the DSQL statement string argument to be executed can not contain any replaceable parameters. From v.2.5 forward, either named or positioned parameters can be provided in the definition of the executable string. Values are assigned immediately before an execution of the request encoded in the string.

In its simplest form EXECUTE STATEMENT executes an SQL statement requesting an operation that does not return any data rows, viz.

- INSERT, UPDATE, DELETE
- EXECUTE PROCEDURE
- any DDL statement except CREATE/DROP DATABASE.

Example

```
CREATE PROCEDURE EXEC_PROC
  (PROC_NAME VARCHAR(31))
AS
  DECLARE VARIABLE SQL VARCHAR(1024);
  DECLARE VARIABLE ...;
BEGIN
  ...
  SQL = 'EXECUTE PROCEDURE ' || PROC_NAME;
  EXECUTE STATEMENT SQL;
END ^
```

Calling it:

```
EXECUTE PROCEDURE EXEC_PROC('PROJECT_MEMBERS');
```

Variable values from a singleton SELECT

The next snippet shows how to execute a SELECT statement string that returns a single row into a set of variables. As with any SELECT statement in a PSQL module, it will throw an exception if the statement returns multiple rows.



Here, we are also able to do something that is not possible in regular PSQL: to perform an operation involving a table or column whose name is not known at compile-time:

```
CREATE PROCEDURE SOME_PROC (
  TABLE_NAME VARCHAR(31),
  COL_NAME VARCHAR(31))
AS
```



```

DECLARE VARIABLE PARAM DATE;
BEGIN
EXECUTE STATEMENT 'SELECT MAX(' || COL_NAME || ') FROM ' || TABLE_NAME
INTO :PARAM;
...
END ^

```

Variable values from a multi-row SELECT

The EXECUTE STATEMENT syntax also supports executing a SELECT statement string inside a FOR loop, to return output a row at a time into a list of variables. There are no restrictions on the SELECT statement that is used—but remember that the compile-time parser can not validate the contents of the string.

```

CREATE PROCEDURE DYNAMIC_SAMPLE (
TEXT_COL VARCHAR(31),
TABLE_NAME VARCHAR(31))
RETURNS (
LINE VARCHAR(32000))
AS
DECLARE VARIABLE ONE_LINE VARCHAR(100);
DECLARE VARIABLE STOP_ME SMALLINT;
BEGIN
LINE = '';
STOP_ME = 1;
FOR EXECUTE STATEMENT
'SELECT ' || TEXTCOL || ' FROM ' || TABLE_NAME
INTO :ONE_LINE
DO
BEGIN
IF (STOP_ME > 320) THEN
EXIT;
IF (ONE_LINE IS NOT NULL) THEN
LINE = LINE || ONE_LINE || ' ';
STOP_ME = STOP_ME + 1;
END
SUSPEND;
END
END ^

```

Using parameters with EXECUTE STATEMENT

From v.2.5 onward, use of replaceable parameters within the executable string is supported.

Syntax The expanded syntax pattern is:

```

[FOR] EXECUTE STATEMENT
(<parameterized-statement>) (<param-assignments>)
[<options>]
[INTO :var1 [, :var2 [, :varN]]] DO
<compound-statement>;

```

The simple **<string>** argument is augmented into **<parameterized-statement>**, a string comprising the SQL statement with a list of either

- named parameters, using the convention **:parameter-name**
- unnamed parameters, represented by the **?** character

If **<parameterized-statement>** is supplied directly to the procedure as a literal string—for example, as an input to the procedure or as a declared value for a local variable—it must be enclosed in single quotes.

The **<param-assignments>** argument is a comma-separated list of values for assignment to the parameters in the **<parameterized-statement>** string. The content of the items in the list depends on how the parameters have been presented in the **<parameterized-statement>**:

- named parameters, which can be in any order, have the form
(paramX := <value1> paramY := <value2>, ...)

Notice the use of the Pascal assignment operator.

- unnamed (positional) parameters must be in strictly the same order as presented in the statement and take the form
(<value1>, <value2>, ...)



*If the values assigned are variables then the use of a colon prefix is not valid in this context. The **<parameterized-statement>** string expression and the **<param-assignments>** list must both be parenthesised.*

[<options>] refers to the advanced syntax, also available from v.2.5 onward, for using EXECUTE STATEMENT to perform DSQL operations across database boundaries in an autonomous transaction. For details, refer to Chapter 31, **Accessing Other Databases**.

Examples¹ This snippet uses named parameters:

```
...
declare license_num varchar(15);
declare stmt varchar(100) =
    'select license from cars where driver = :driver and location = :loc';
begin
...
for select id from drivers into current_driver do
begin
for select location from driver_locations
where driver_id = :current_driver
into :current_location do
begin
...
execute statement
    (stmt)
    (driver := current_driver, loc := current_location)
into license_num;
...

```

The same snippet, using unnamed (positional) parameters:

1. Examples adapted from the description of EXECUTE STATEMENT in *Firebird 2.5 Language Update*, courtesy of Paul Vinkenoog and other Firebird Project authors.

```

...
declare license_num varchar(15);
declare stmt varchar(100) =
'select license from cars where driver = ? and location = ?';
begin
...
for select id from drivers into current_driver do
begin
  for select location from driver_locations
  where driver_id = :current_driver
  into current_location do
  begin
    ...
    execute statement (stmt) (current_driver, current_location)
    into license_num;
    ...
  end
end
end

```

Caveats

EXECUTE STATEMENT needs very careful use. Operations using it are risky and may be slower than doing the same task more directly. Make a rule to use it only when it is impossible to achieve the objective by other means or in the unlikely event that it actually improves the performance of the statement. Be aware of the risks:

- There is no way to validate the syntax of the statement string argument—so test it to death before you deploy.
- No dependency checks or protections exist to prevent tables or columns from being dropped or altered.
- If the stored procedure has special privileges on some objects, the dynamic statement submitted in the EXECUTE STATEMENT string does not inherit them. Privileges are restricted to those granted to the CURRENT_USER (the user who is executing the procedure).



*From v.2.5 onward, it is possible to run EXECUTE STATEMENT in an autonomous transaction, connecting to an external database with the option to do so using credentials different to the CURRENT_USER. For details, refer to Chapter 31, **Accessing Other Databases**.*

Using Cursors

Cursors consist of three main elements:

- a set of rows defined by a SELECT expression
- a pointer that passes through the set from the first row to the last, isolating that row for some kind of activity
- a set of variables—defined as local variables, output arguments, or both—to receive the columns returned in each row as it is touched by the pointer

The passage of the pointer through the set can be considered a “looping” operation. The operations that occur during the loop when a row has the pointer can be simple or complex.

PSQL has three cursor forms, one well-known, the other less used in older versions because it was not well documented and the third an enhanced version of the second:

- The best-known is surfaced in the FOR..SELECT construct, described in detail in the next chapter, which fully implements looping syntax and is widely used for procedures that are designed to return multi-row data sets to the caller (known as *selectable procedures*). It is, however, a perfectly valid and common usage to run FOR..SELECT cursor loops inside an *executable procedure*, to perform row-by-row DML operations (INSERT, UPDATE, DELETE). This form is discussed in detail in the next chapter.
- Lesser-known is an earlier implementation, an *undeclared named cursor*, that was inherited from ESQL and, for a long time, was considered to have been deprecated in favour of the FOR SELECT.. style. It surfaces a cursor object and allows positioned updates and deletes.
- From the “2” series onward, the named cursor evolves into a fully-fledged run-time object whose characteristics and usage will be more familiar to those who have used cursors in the procedure language of another database management system. It is declared in the header of the procedure, block or trigger, along with other variables. We refer to it as the *explicit named cursor*.

The named cursor provides a very fast way to do bulk operations in a stored procedure, since it uses the RDB\$DB_KEY to locate its targets. RDB\$DB_KEY—or just “the DBKEY”—is an internal feature that can be used with care in a number of situations. There’s more information about the RDB\$KEY in the topic [*Using the Internal RDB\\$DB_KEY*](#), in the next chapter.

The Undeclared Named Cursor

The original undeclared named cursor is available in versions 1.0.x and 1.5.x. Its syntax pattern looks like this:

```
...
FOR SELECT <column-list>
  FROM <named-table>
  FOR UPDATE
  INTO <variables>
  AS CURSOR <cursor-name>
DO
/* either UPDATE ... */
BEGIN
  UPDATE <named-table>
  SET ... WHERE CURRENT OF <cursor-name>;
  ...
END
/* or DELETE */
BEGIN
  DELETE FROM <named-table>
  WHERE CURRENT OF <cursor-name>
END
...
```

In this form, the cursor is not pre-declared but is assigned its name using the AS keyword. The **<named-table>** must be a table, not a view or any other kind of virtual object.

The WHERE CURRENT OF <cursor-name> phrase refers to the currently fetched row. A positioned update or delete can be performed on the underlying table or view row at this point.

The Explicit Named Cursor

v.2.0 onwards

Explicit named cursors are supported from the “2” series onwards, both in PSQL modules and in DSQL EXECUTE BLOCK statements.

Multiple named cursors can be in action in the same module and can co-exist in the same module with undeclared cursors.



Do not try to intermix the two styles, attempting for example to perform an OPEN, CLOSE or FETCH on a cursor of the undeclared style.

The enhanced implementation requires the cursors to be declared in the header section, as variables of other types are. The cursor can later be opened, used to walk the result set from top to bottom and then, for tidiness, closed.

Positioned updates and deletes use the WHERE CURRENT OF <cursor-name>) phrase, as with the older, undeclared form.

Syntax patterns for explicit cursors

Cursors can be declared in any order in the same block as the variable declarations:

```
DECLARE [VARIABLE] <cursor_name>
CURSOR FOR ( <select-statement> );
```

The **<cursor-name>** can be any valid Firebird identifier. Duplicating the name of a cursor—whether of the explicit or the undeclared form—is not allowed. However, the compiler will not object if a cursor has the same name as a local variable or parameter. However, for the sake of common sense and sound documentation, don’t do it!

The **<select-statement>** element must be a valid SELECT expression that retrieves columns from a single table or view, without joins or columns based on expressions. It must be enclosed in parentheses. If you want to be able to modify data via a cursor on a view, the view itself must be updatable.

You can include a FOR UPDATE phrase in the SELECT expression without causing an error, but it is redundant. Updates and deletes performed in the cursor loop will be executed regardless of its presence or absence.

Parameters

The **<select-statement>** can be parameterised, using the pre-declared local variables or input/output arguments, complete with the colon prefix, after the fashion of named parameters. They are not parameters in the manner of regular DML statements and the “?” placeholder symbol cannot be used.

The form is:

```
SELECT ... FROM ATABLE
WHERE ACOLUMN = :aVariable
```

When the cursor is opened, the “parameter” is assigned the current value of the variable.

Because these “parameters” are really variables, there’s a trap. If, during the execution of the loop, the value of the variable changes, the statement is likely to be re-evaluated before the next row is fetched. If that is the behaviour you intend, make certain to test rigorously to ensure you get the results you expect. If you need the whole set to remain stable throughout the loop, take pains to avoid using variables that the loop action could change.

Cursor manipulation statements

To open a previously declared cursor, execute its SELECT statement and prepare it to fetch a record from the result set:

```
OPEN <cursor_name>;
```



A call to open a cursor that is already open will cause a compile failure.

To move the cursor pointer forward one row and retrieve the current row of the result set:

```
FETCH <cursor_name> INTO <var-name> [, <var-name> ...];
```

<var-names> are declared variables, either local variables or output arguments.

The values of the retrieved column set are stored in the variables in the left-to-right order of the bound SELECT specification. The ROW_COUNT context variable after a FETCH will be either 1 (the cursor pointer retrieved data) or 0 (the pointer encountered end-of-file).

The WHERE CURRENT OF <cursor-name> phrase refers to the currently fetched row. A positioned update or delete can be performed on the underlying table or view row at this point.

Prior to the “2” series, the PSQL parser does not complain about a WHERE CURRENT OF <cursor-name> outside the cursor loop of the older-style undeclared cursor, it was possible referring to “current of <cursor>” outside the scope of the cursor loop. It was a situation that could give rise to run-time errors. From “2” series onward, the procedure will not compile if this logic error is present for either a declared or undeclared cursor.

To close an open cursor:

```
CLOSE <cursor_name>;
```

The cursor is closed and no further operation is possible unless it is reopened using another OPEN statement. If a cursor is reopened, the pointer will again be ready to receive the first row.



Closing a cursor is good practice. However, any open cursors will be closed automatically when execution exits the module.

A call to close a cursor that is not open will cause a compile failure.

Using explicit named cursors

The following are some simple usage scenarios for named cursors.

Example 1

Declare a cursor and use it to read out a list of names to be returned from a selectable stored procedure. Notice the use of the ROW_COUNT context variable after each FETCH statement to check whether any row was returned and break the loop if EOF was encountered:

```
...
AS
DECLARE RNAME CHAR(31);
DECLARE C CURSOR FOR
```

```

    ( SELECT RDB$RELATION_NAME FROM RDB$RELATIONS );
BEGIN
    OPEN C;
    WHILE (1 = 1) DO
    BEGIN
        FETCH C INTO :RNAME;
        IF (ROW_COUNT = 0) THEN
            LEAVE;
        SUSPEND;
    END
    CLOSE C;
END
...

```

Example 2 An explicit cursor is enclosed in a FOR...SELECT loop, each iteration supplying a value for a search parameter that is declared for the nested cursor:

```

...
AS
    DECLARE RNAME CHAR(31);
    DECLARE FNAME CHAR(31);
    DECLARE C CURSOR FOR
    ( SELECT RDB$FIELD_NAME FROM RDB$RELATION_FIELDS
      WHERE RDB$RELATION_NAME = :RNAME
      ORDER BY RDB$FIELD_POSITION );
BEGIN
FOR
    SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
    INTO :RNAME DO
    BEGIN
        OPEN C;
        FETCH C INTO :FNAME;
        CLOSE C;
        SUSPEND;
    END
END

```

Example 3 Positioned updates are executed for each row fetched by the cursor:

```

...
AS
    DECLARE BATCHID BIGINT;
    DECLARE MANUF_DATE TIMESTAMP;
    DECLARE QSTAMP BIGINT;
    DECLARE C CURSOR FOR
    ( SELECT BATCH_ID, MANUF_DATE FROM NEW_STOCK
      WHERE CAST(MANUF_DATE AS DATE) <= CURRENT_DATE
      AND SERIAL_NO IS NULL
      ORDER BY BATCH_ID, MANUF_DATE );
BEGIN
    OPEN C;

```

```

WHILE (1 = 1) DO
BEGIN
    FETCH C INTO :BATCHID, ;
    IF (ROW_COUNT = 0) THEN
        LEAVE;
    QSTAMP = NEXT VALUE FOR S_QSTAMP;
    UPDATE NEW_STOCK
        SET SERIAL_NO = '||:BATCHID ||':QSTAMP
        WHERE CURRENT OF C;
    END
    CLOSE C;
END

```

Example 4 The cursor operator `WHERE CURRENT OF` steps through a cursor set selected from an updatable view:

```

...
AS
DECLARE CURSOR VIEW_CURSOR FOR
(SELECT ... FROM MY_VIEW
INTO ... WHERE..);
BEGIN
...
DELETE FROM MY_VIEW
WHERE CURRENT OF VIEW_CURSOR;
...
END

```

Explicit vs undeclared cursors

The explicit cursor gives more control over the flow of execution and enables multiple cursors to be open at once. However, the undeclared cursor is simpler to use and less prone to coding errors, insofar as it takes care of its own opening, fetching and end-of-file actions.

Prior to the “2” series, of course, you are not faced with the choice. If you are using a server version that supports both styles, it is suggested that you experiment with both to determine which is better for the work you want a cursor to do.

Developing Modules

Developing the PSQL modules that give the “grunt” to the data management system is a vital and precious part of the roles of both developer and database administrator. Because Firebird's administrative demands are light, it is usually practicable for the roles to merge. During development, it is not unusual for many members of a software team to be developing, testing and modifying server-side program modules concurrently. Thus, the measures taken to protect and standardize application code are just as important for PSQL code.

Adding comments

Stored procedure code should be commented to aid debugging and application development. Comments are especially important in stored procedures since they are global to the database and can be used by many different application developers.

Both block (multi-line) and in-line comments can be included in the headers and bodies of PSQL modules.

Block comments

Block comments use the C convention:

```
/* This comment can span multiple
   lines in a script */
```

A block comment can be of any length, as long as it is preceded by `/*` and followed by `*/`.

In-line comments

The `/* */` style of comment can also be embedded inside a statement as an in-line comment, e.g.

```
/* The following statement retrieves the rows for
   the cursor */
FOR SELECT DEPT_NO, DEPARTMENT, BUDGET
FROM DEPARTMENT
   INTO :DNUM, :DDESC, :DBUDG
DO BEGIN
....
```

One-line comments

An alternative convention is available for commenting a single line: the double hyphen, e.g.,

```
-- don't change this!
```

The `--` commenting convention can be used anywhere on a line, to “comment out” everything from the marker to the end of the current line. For example:

```
FOR SELECT DEPT_NO, DEPARTMENT -- , BUDGET
FROM DEPARTMENT
   INTO :DNUM, :DDESC -- , :DBUDG
DO BEGIN
....
```

v.1.0.x This double-hyphen style of comment is not implemented in the old v.1.0 series.

Case-sensitivity, white space and size

If double-quote delimiters were used when your database objects were defined, then all case-sensitivity and quoting rules that apply to your data in dynamic SQL must also be applied when you refer to those objects in procedure statements.

All other code is case-insensitive. For example (assuming no delimited object identifiers are involved) these two statement fragments are equivalent:

```
CREATE PROCEDURE MYPROC...
create procedure myproc...
```

The compiler places no restrictions on white space or line feeds. It can be useful, for readability of your code, to adopt some form of standard convention in the layout of your procedure code. For example, you might write all keywords in upper case, indent code blocks, list variable declarations in separate lines, place comma separators at the beginning of lines, and so on.

However, be aware that the size of the BLR that is created when a stored procedure or trigger is compiled cannot exceed 64 Kb, including all spaces and counting all bytes for strings. If you are writing monolithic procedures, consider redesigning them into smaller units.

Managing your code

Considering that the high-level language for Firebird server-side programming is SQL and that source code is presented to the engine in the form of DDL “super-statements” for compilation into database objects, it is not surprising that all code maintenance is also performed using DDL statements. These statements are consistent with the conventions for maintenance of other objects in SQL databases:

- Redefinition of compiled objects—stored procedures and triggers—employs the `ALTER PROCEDURE | TRIGGER` syntax. For stored procedures, Firebird also provides `RECREATE PROCEDURE` and `CREATE OR REPLACE PROCEDURE` syntaxes.
- `DROP PROCEDURE | TRIGGER` statements are used to remove modules.

There are two ways to manage procedures and triggers: entering the statements interactively through *isql* or another tool which can submit direct DSQL; or with one or more input files containing data definition statements—commonly known as *scripts*.

The interactive interface seems a quick and easy way to do things—until the first time you need to change, check or recover something. Using scripts is the recommended way, because scripts not only provide retrievable documentation of the code you submitted, but can include comments and can be modified easily.

Editing tools

Any ASCII plain text editor can be used, as long as it does not save any non-printable characters other than carriage returns (ASCII 13), line feeds (ASCII 10) and tab characters (ASCII 9). Many editors are available which provide syntax highlighting for SQL, including Notepad Plus (for multiple OS platforms), Context (a free programmer’s editor for Windows), the IDE editors of Delphi™ and Lazarus and many others.



The `isql` command shell utility can be used as an editor, using the `EDIT` command. `EDIT` will use your choice of text editor if the appropriate environment variables are set up on your system. On POSIX, set the environment variable `VISUAL` or `EDITOR`. On Windows, set `EDITOR`.

It is a useful practice add the extension “.sql” to Firebird script files. Apart from its usefulness in identifying scripts in your filesystem, the “.sql” extension is recognised automatically as an SQL batch file by many editing tools that support SQL syntax highlighting.

Compiling stored procedures and triggers

To complete any script file that you are going to process through *isql*, you must include at least one “empty line” past the last statement or comment in the file. To do this, press the Return (Enter) key at least once in your text editor following the last line of code. When you have completed your procedure, save it to a file of any name you prefer.

To compile your stored procedure, simply run your script using the INPUT command in *isql* or through the scripting interface of your database management tool—while connected to the database, of course!

Script errors

Firebird generates errors during parsing if there is incorrect syntax in a CREATE PROCEDURE | TRIGGER statement. Error messages look similar to this:

```
Dynamic SQL Error
-SQL error code = -104
-Token unknown - line 4, char 9
-tmp
```

The line numbers are counted from the beginning of the CREATE statement, not from the beginning of the script file. Characters are counted from the left, including blank spaces, and the unknown token indicated is either the source of the error, or immediately to the right of the source of the error. When in doubt, examine the entire line to determine the source of the syntax error.

v.1.0.x *isql* in v.1.0.x is a lot less friendly than its antecedents about describing script errors.



Firebird does not provide a debugging feature for PSQL modules. Several third-party tools that are on the market do provide this capability.

Object dependencies

The Firebird engine is meticulous about maintaining information about the inter-dependencies of the database objects under its management. Special considerations are necessary when making changes to stored procedures that are currently in use by other requests. A procedure is “in use” when it is currently executing, or if it has been compiled internally to the metadata cache by a user request. Furthermore, the engine will defer or disallow compilation of ALTER or DROP statements if the existing compiled version is found in the metadata cache.

Changes to procedures are not visible to client applications until they disconnect from and reconnect to the database.



Triggers and procedures that invoke procedures that have been altered or recreated do not have access to the new version until the database is in a state in which all clients are disconnected.

Ideally, submit create and alter statements for PSQL modules when there are no client applications running.

Altering and dropping modules

When you alter a procedure or trigger, the new procedure definition replaces the old one. To alter a procedure or trigger definition, follow these steps:

- 1 Copy the original data definition file used to create the procedure. Alternatively, use `isql -extract` to extract the source of a procedure or trigger from the database to a file.
- 2 Edit the file, changing `CREATE` to `RECREATE` or `ALTER`, and modifying the definition as desired.
- 3 Run the modified script in "clean" conditions, as outlined above.

To drop a module:

```
DROP {PROCEDURE | TRIGGER} module-name;
```

Privileges

Only SYSDBA and the owner of a procedure or trigger can alter or drop it.

“Object is in use” error

This error tends to frustrate developers more than any other. You are logged in, typically, as SYSDBA or owner of the database. You have exclusive access—which is desirable when changing metadata—and yet, here is this phantom user, somewhere, using the object whose metadata you want to modify or delete.

The source of this mystery will be one or more of the following:

- In shutting down the database, preparing to get exclusive access, you (or another human) were already logged in as SYSDBA, owner or (on Linux/UNIX) a suitably privileged OS user. In testing the conditions for shutdown, Firebird ignores these users and any transactions they may have current, or begin after shutdown has begun. Any uncommitted transaction whatsoever—even a `SELECT`—that involves the object or any objects that depend on it, or upon which the object itself depends, will raise this error.
- An “interesting transaction” that remains in the database through an abnormal termination by some user at some point, that involves dependencies related to this object, will raise this error.
- You, or another suitably privileged user, previously attempted to redefine or delete this object, or another dependent object, and the operation was deferred because that object was in use.



This situation can provoke a chain of inconsistencies in your database. For example, if a `gbak` backup is performed whilst the database has objects in this state, the backup file may fail to restore.

Whenever you see this error and believe that you have eliminated all possible reasons for it, take it as a sign that your database needs a validation before you proceed with any further metadata changes—see *Analyzing and Repairing Logical Corruption* in Chapter 35, **Configuring and Managing Databases**.

To see a list of procedures or triggers and their dependencies in `isql`, use the `SHOW PROCEDURES` or `SHOW TRIGGERS` commands, respectively.

Deleting source from modules

Developers sometimes want to “hide” the source code of their PSQL modules when they deploy databases. You can delete the stored source code without affecting the capability of the module. Just be sure that you have up-to-date scripts before you do it!

The sources for all modules are stored in the system tables RDB\$PROCEDURES and RDB\$TRIGGERS.

To delete the source for a procedure:

```
UPDATE RDB$PROCEDURES
SET RDB$PROCEDURE_SOURCE = NULL
WHERE RDB$PROCEDURE_NAME = 'MYPROC';
```

To delete the source for a trigger:

```
UPDATE RDB$TRIGGERS
SET RDB$TRIGGER_SOURCE = NULL
WHERE RDB$TRIGGER_NAME = 'MYTRIGGER';
```



Be aware that deleting the source won't stop someone who is determined to steal your code. The executable code is stored in a language-like binary format that is very simple to reverse-engineer back into PSQL. So—consider whether the gain from obscuring the PSQL is worth the loss to you and others who maintain the system of the ability to review and extract the source.

Internals of the Technology

When any request invokes a stored procedure, the current definition for that stored procedure is copied at that moment to a metadata cache. On Classic, this copy persists for the lifetime of the user's connection. On Superserver, it stays “live” until the last connection is logged out.

A request is one of the following:

- A client application that executes the stored procedure directly
- A trigger that executes the stored procedure; this includes system triggers that are part of referential integrity or check constraints
- Another stored procedure that executes the stored procedure

Effects of changes to modules

Once invoked, a trigger or stored procedure request persists in the metadata cache whilst there are clients connected to the database, regardless of whether any connected client makes use of the trigger or stored procedure. There is no mechanism to force these outstanding requests to update their metadata cache. For this reason, changes to PSQL module definitions are deferred, to a greater or lesser extent, in most cases. The ability of clients to see changes is different for Superserver than for Classic.

Superserver

Because existing requests are emptied from the metadata cache only when the last client disconnects from the database, they simply may never update on a 24/7 system. The only way to guarantee that all copies of stored procedures and triggers are purged from the metadata cache is for all connections to the database to terminate. When users log in again, they will all see the newest version of the stored procedure.

Classic and Superclassic

Altering or dropping a stored procedure takes effect immediately, for new connections made after the change is committed. New connections that invoke the stored procedure will see the latest version. However, other connections continue to see the version of the stored procedure that they first saw. In practical terms, it makes sense to disconnect clients before you commit your changed module.

CHAPTER 29

STORED PROCEDURES AND EXECUTABLE BLOCKS

A stored procedure is a self-contained program written in Firebird PSQL, compiled by the Firebird's internal binary language interpreter and stored as executable code within the metadata of a database. Once compiled, the stored procedure can be invoked directly from an application or another PSQL module, using an `EXECUTE PROCEDURE` or `SELECT` statement, according to the style of procedure that has been specified.

Stored procedures can accept input parameters from client applications as arguments to the invoking query. They can return a set of values to applications as output parameters.

Firebird procedure and trigger language includes SQL data manipulation statements and some powerful extensions, including `IF ... THEN ... ELSE`, `WHILE ... DO`, `FOR SELECT ... DO`, system-defined exceptions and error handling and events.

Stored procedures can be called from applications using dynamic SQL statements. They can also be invoked interactively in *isql* and many of the database desktop tools recommended for use with Firebird. Executable modules—including nested procedures—can be used in scripts, with the limitation that all input parameters be constants and that there be no outputs.¹

Styles of Stored Procedure

In Firebird you can write two distinct styles of stored procedure:

Executable stored procedures, so called because they are invoked with an `EXECUTE PROCEDURE` statement, can optionally return a single row of one or more output values. They are often used to perform insert, update or delete operations or to launch a suite of operations such as a batch import of external data.

1. It is not possible to pass parameters around in scripts.

Selectable stored procedures are written with some special language extensions to produce a multiple-row output set that is returned to the caller using a `SELECT` query—a style of “virtual table”, in fact.

The engine does not differentiate between an executable and a selectable stored procedure. If requested, it will try to form a set from an executable procedure or to execute something in a selectable procedure—and throw exceptions when the logic of the request fails, of course!. It is up to you to make sure that your server code does what you intend it to do and that your client code submits the appropriate requests.

Creating a Stored Procedure

In your script or in *isql*, begin by setting the terminator symbol that will be used to mark the end of the `CREATE PROCEDURE` syntax. The following example will set the terminator symbol to '&':

Syntax

```
...
SET TERM &;

CREATE PROCEDURE procedure-name
[(input-argument data-type [, input-argument data-type [...]])]
[RETURNS (output-argument data-type [, output-argument data-type [...])]
AS
<procedure-body>
<procedure-body> =
[DECLARE [VARIABLE] var data-type;

[...]]]
BEGIN
    <compound-statement>;
END <terminator>
```

Header elements

In the header, declare

- the name of the procedure, which is required and must be unique in the database, e.g.
`CREATE PROCEDURE MyProc`
- any optional input arguments (parameters) required by the procedure, and their data types, as a comma-separated list enclosed by brackets, e.g.

```
CREATE PROCEDURE MyProc (
    invar3 VARCHAR(20),
    invar1 INTEGER,
    invar2 DATE)
...
```

The name of each argument must be unique within the procedure. The name of an input argument need not match the name of any parameter in the calling program. The data type can be

- any standard SQL datatype except `ARRAY`

- a domain name (v.2.1+)
- the same data type as a domain name, using TYPE OF <domain-name> (v.2.1+)
- the same data type as a column in a table or view, using TYPE OF <relation.column-name> (v.2.5+)
- any optional output arguments (parameters) required, and their data types, as a comma-separated list enclosed in brackets following the keyword RETURNS, e.g.,

```
CREATE PROCEDURE MyProc (
    invar3 VARCHAR (20) COLLATE FR_CA,
    invar1 INTEGER,
    invar2 DATE)
RETURNS (
    outvar1 INTEGER,
    outvar2 VARCHAR(20),
    outvar3 DOUBLE PRECISION)
...
```

The name of each argument must be unique within the procedure. The same rules apply to the data type in the declaration as are described above for the input arguments. For a more detailed description of the use of the non-native data types in the declarations, refer to the topic *Use of domains and existing column definitions* in the previous chapter.



From v.2.1 you can include a COLLATE clause with any parameter or variable declaration that is a CHAR, VARCHAR or text BLOB type. Of course, it must name a collation that is available to the database for the character set of the data. You can also declare a parameter or variable non-nullable (2.1+) and declare default values for inputs and outputs (2.0+).

- the keyword AS, which is required:

```
CREATE PROCEDURE MyProc (
    invar3 varchar(20) COLLATE FR_CA;
    invar1 INTEGER,
    invar2 DATE = CURRENT_DATE NOT NULL)
RETURNS (
    outvar1 INTEGER,
    outvar2 VARCHAR(20),
    outvar3 DOUBLE PRECISION)
AS
...
```

Body elements

Body Syntax

The syntax outline for the procedure body is:

```
<procedure_body>=[<variable-declaration-list>]
<compound-statement>
```

Local variables

If you have local variables to declare, their declarations come next. Each declaration is terminated with a semi-colon. Local variables can be optionally initialized in the declaration.²

The syntax outline for the `<variable_declaration_list>` is:

```
DECLARE [VARIABLE] var-name data-type [{ '=' | DEFAULT } value];
[DECLARE [VARIABLE] var-name data-type; ...]
```

For example,

```
CREATE PROCEDURE MyProc (
    invar1 INTEGER,
    invar2 DATE)
RETURNS (
    outvar1 INTEGER,
    outvar2 VARCHAR(20),
    outvar3 DOUBLE PRECISION)
AS
...
DECLARE VARIABLE localvar integer DEFAULT 0;
DECLARE VARIABLE anothervar DOUBLE PRECISION = 0.00;
```



The keyword `VARIABLE` is optional.³

Main code block

Next comes the main code block, designated in the higher-level syntax diagram as `<compound-statement>`. It starts with the keyword `BEGIN` and ends with the keyword `END`.

The syntax outline for the `<compound-statement>` is:

```
BEGIN
    <compound_-statement>
    [<compound-statement>...]
END <terminator>
```

All `<compound-statement>` structures consist of single statements and/or other `<compound-statement>` structures that can nest others. For example,

```
CREATE PROCEDURE MyProc (
    invar1 INTEGER,
    invar2 DATE)
RETURNS (
    outvar1 INTEGER,
    outvar2 VARCHAR(20),
    outvar3 DOUBLE PRECISION)
AS
...
DECLARE VARIABLE localvar integer DEFAULT 0;
DECLARE VARIABLE anothervar DOUBLE PRECISION = 0.00;
BEGIN
    <compound-statement>
END &
```

2. Not supported in v.1.0.x.
3. `VARIABLE` is not an optional keyword in v.1.0.x.

The `<compound-statement>` element can be any or all of a single statement, a block of statements and embedded blocks of statements bounded by `BEGIN...END` pairs. Blocks can include

- Assignment statements, to set values of local variables and input/output parameters.
- `SELECT` statements, to retrieve column values into variables. `SELECT` statements must have an `INTO` clause as the last clause; and corresponding local variable or output argument declarations for each column selected.
- Looping structures, such as `FOR SELECT ... DO` and `WHILE ... DO`, to perform conditional or looping tasks.
- Branching structures using `IF ... THEN ... [ELSE]`
- `EXECUTE PROCEDURE` statements, to invoke other procedures, with optional `RETURNING_VALUES` clauses to return values into variables. Recursion is allowed.
- `SUSPEND` and `EXIT` statements, that return control and, optionally, return values, to the calling application or PSQL module.
- Comments to annotate procedure code.
- `EXCEPTION` statements, to return custom error messages to applications or to signal conditions for exception handlers
- `WHEN` statements to handle specific or general error conditions.
- `POST_EVENT` statements to add an event notification to the stack

Example

```
...
BEGIN
  FOR SELECT COL1, COL2, COL3, COL4
  FROM TABLEA INTO :COL1, :COL2, :COL3 DO
  BEGIN
    <statements>
  END
  <statements>
END &
SET TERM ; &
COMMIT;
```

Notice the termination of the entire procedure declaration with the terminator character previously defined by the `SET TERM` statement. After the procedure body is complete, you can set the terminator character back to the default semicolon. It need not always be the case. In a DDL script, where you may be declaring several PSQL modules, you can keep the alternative terminator current. Some people make it a practice to use an alternative terminator throughout all of their scripts—thus reserving the semicolon only for PSQL statement terminators—as a matter of personal preference.

Executable Procedures

When you work with Firebird's stored procedure language and program module structures, it is necessary to make the distinction between procedures that are executed—with the aim of altering data—and those that are intended to return a virtual table to the caller by way of a `SELECT` statement. The first—the *executable*

procedure—is the form most familiar to those used to working with other database management systems: the executable procedure.

Complex processing

One of the more obvious and common uses of executable procedures is to perform complex calculations on input data and perform updates on one or many tables. Complex business rules and routines are centralized on the server. Any client application with the appropriate permissions can invoke the same routines and get the same results, regardless of the host language environment. Apart from the savings in programmer and testing hours, server-side execution eliminates the integrity risks inherent in repeating and maintaining the same complex operations in multiple client language environments.

Support for “live” client sets

Many client interface layers implement dataset or recordset classes that fetch output sets via SELECT statements. These client classes typically provide DML methods that target a single row from a buffer that manages output from a server-side cursor. The row is selected by the user and the class instance (object) uses the row’s unique key to simulate a positioned update or delete in the underlying database table. For inserts, the object “opens an empty row”, an input list of columns of the same types as those in the buffer, and accepts key and other values as input for the columns.

A single UPDATE, DELETE or INSERT statement in SQL can operate on only one table. When the dataset (recordset) is selected from a natural table and contains the table’s unique key, it can be considered “live”, since its methods can pass an ordinary UPDATE, DELETE or INSERT statement. The usual term for this type of set is *naturally updatable*. A set that joins multiple tables is not naturally updatable. Executable stored procedures can be designed with input arguments that accept keys and values for multiple tables and execute the required operations on each table. The technique allows the client applications to treat joined sets as though they were “live”.

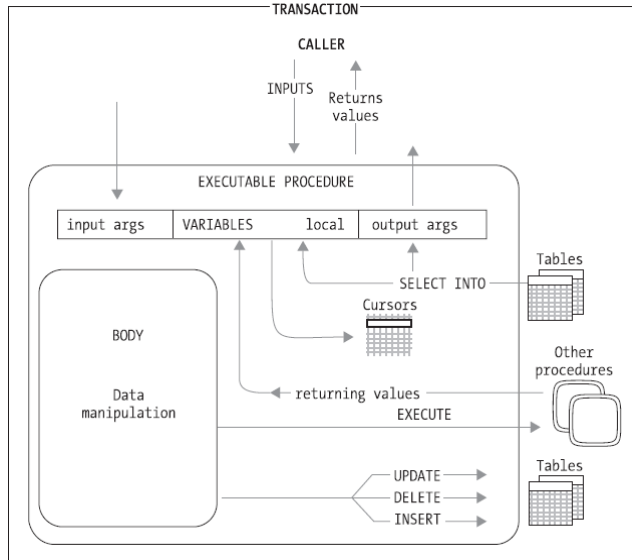
Operations in executable procedures

Virtually every data manipulation in SQL is available to the executable stored procedure. All activity is executed within the transaction context of the caller and is committed when the transaction commits. Rows that are changed by an operation in the procedure are versioned in exactly the same way as those posted by a DML request from the client.

Procedures can call other procedures, passing variables to input arguments and accepting return values into variables. They can insert one or many rows, change single rows, form cursors to target specific rows for positioned updates and deletes and they can perform searched updates and deletes.

Once the procedure begins executing, values brought in as input arguments behave like local variables. Output arguments are read-write variables that can change value (though not data type) many thousands of times during the course of execution.

Figure 29.1 illustrates the flow of operations when an executable procedure is called.

Figure 29.1 Operations in an executable procedure

A multi-table procedure

The executable procedure `DELETE_EMPLOYEE` is a version of one that you can find in the `EMPLOYEE` database in your Firebird /examples directory. It implements some business rules regarding employees who are leaving the company.

Declaring an exception

Because we are going to use an exception in this procedure, it needs to be created before the procedure:

```
CREATE EXCEPTION REASSIGN_SALES
    'Reassign the sales records before deleting this employee.' ^
COMMIT ^
```

The procedure

Now, the procedure itself. The input argument, `EMP_NUM`, corresponds to the primary key of the `EMPLOYEE` table, `EMP_NO`. It allows the procedure to select and operate on a single employee record and foreign keys in other tables that reference that record.

```
CREATE PROCEDURE DELETE_EMPLOYEE (
    EMP_NUM INTEGER )
AS
    DECLARE VARIABLE any_sales INTEGER DEFAULT 0;
BEGIN
    ...
```

We want to establish whether the employee has any sales orders pending. If so, we raise an exception. In Chapter 32, **Error Handling and Events**, we'll pick up the same procedure and implement some extra processing to trap the exception and handle the condition right

inside the procedure. For now, we let the procedure terminate and use the exception message to inform the caller of the situation.

The SELECT.. INTO construct

The SELECT..INTO construct is very common in PSQL. When values are queried from tables, the INTO clause enables them to be stored into variables—local variables or output arguments. In this procedure, there are no output arguments. We use the variable ANY_SALES that we declared and initialized at the head of the procedure body to store a count of some sales records. Notice the colon (:) prefix on the variable ANY_SALES. We'll look at that when the procedure is done.

```
...
SELECT count(po_number) FROM sales
  WHERE sales_rep = :emp_num
  INTO :any_sales;
IF (any_sales > 0) THEN
  EXCEPTION reassign_sales;
```

In the case that purchase order records are found, the procedure ends tidily with the EXCEPTION statement that, in the absence of an exception handler, takes execution directly to the last END statement in the whole procedure. Under these conditions, the procedure has completed and the exception message is transmitted back to the caller.

If there is no exception, execution continues. Next, the procedure has some little jobs to do, to update some positions as vacant (NULL) if they are currently held by our employee, remove the employee from projects and delete his/her salary history. Finally, the employee record itself is deleted.

```
...
UPDATE department
  SET mgr_no = NULL
  WHERE mgr_no = :emp_num;
UPDATE project
  SET team_leader = NULL
  WHERE team_leader = :emp_num;
DELETE FROM employee_project
  WHERE emp_no = :emp_num;
--
DELETE FROM salary_history
  WHERE emp_no = :emp_num;
DELETE FROM employee
  WHERE emp_no = :emp_num;
...
```

Job done, employee gone. An optional EXIT statement can be included for documentation purposes—it can help a lot when you are navigating scripts where there are many procedure definitions and those procedures have many nested BEGIN..END blocks:

```
EXIT;
END ^
COMMIT^
```

The colon (:) prefix for variables

In this procedure, we noted two different ways in which the colon prefix was used on variables:

- 1 earlier, it was applied to the local variable :ANY_SALES when it was being used in an INTO clause as the destination for a data item returned from a SELECT.
- 2 in the latter statements, it was used for a slightly different reason. PSQL syntax requires the colon prefix to be on any variable or argument when it is used in a DSQL statement.

These two usages of the colon prefix are consistent throughout PSQL. If you forget them when they should be used, or use them when PSQL does not require them, your procedures won't compile and the parser will throw an exception. Worse, if a variable of the same name as a column in the table is used in an SQL statement, without the colon, the engine thinks it is the column that is being referred to, processes it on that assumption and throws no exception. Needless to say, the result of the operation will be quite unpredictable.

Using (calling) executable procedures

An executable procedure is invoked with EXECUTE PROCEDURE. It can return, at most, one row of output. To execute a stored procedure in *isql*, use the following syntax pattern:

```
EXECUTE PROCEDURE name [( [argument [, argument ...]] )];
```

The procedure name must be specified.

Calling rules about input argument values

- Values must be supplied for all input arguments.
- If there are multiple input arguments, they are passed as a comma-separated list.
- Each argument is a constant, an expression that evaluates to a constant or a replaceable parameter.
- Variables can be passed as input arguments only inside a PSQL module.
- Replaceable parameters can be passed only by external DSQL statements
- Constants and expressions that resolve to constants are valid for any call.
- Expressions that operate on variables or replaceable parameters are not allowed
- Brackets enclosing the argument list are optional.

Because our procedure DELETE_EMPLOYEE has no return arguments, the syntax for calling it from a client application or from another procedure is the same:

```
EXECUTE PROCEDURE DELETE_EMPLOYEE (29);
```

However, when executing a procedure from within another procedure, input arguments can be (and usually are) furnished as variables. Because EXECUTE PROCEDURE is a DSQL statement, the PSQL syntax requires that the variable name be prefixed by a colon:

```
EXECUTE PROCEDURE DELETE_EMPLOYEE (:EMP_NUMBER);
```

Another procedure ADD_EMP_PROJ takes two input arguments, keys for an employee and a project, respectively. An instance might be called like this:

```
EXECUTE PROCEDURE ADD_EMP_PROJ (32, 'MKTPR');
```

Replaceable parameters are used for input arguments when calling this procedure from a client application:

```
EXECUTE PROCEDURE DELETE_EMPLOYEE (?);
```

Outputs and exits

If an output parameter has not been assigned a value, its value is unpredictable and this can lead to errors, sometimes subtle enough to compromise data integrity. A procedure should ensure that not just local variables but also all output parameters are initialized to a default final value, in advance of the processing that will assign values, to ensure that valid output is available when SUSPEND or EXIT is executed.

EXIT and SUSPEND

In both select and executable procedures, EXIT causes execution to jump immediately to the final END statement in the procedure, without executing anything.

What happens when a procedure reaches the final END statement depends on the type of procedure:

- In a SELECT procedure, the SQLCODE 100 is set, to indicate that there are no more rows to retrieve, and control returns to the caller.
- In an executable procedure, control returns to the caller, passing the final output values, if any. Calling triggers or procedures receive the output into variables, as RETURNING_VALUES. Applications receive them in a record structure.

In executable procedures, SUSPEND has exactly the same effect as EXIT, since the only output (if any) is a single row—so there’s nothing for the procedure to wait for.

Recursive Procedures

If a procedure calls itself, it is recursive. Recursive procedures are useful for tasks that involve repetitive steps.

Each invocation of a procedure is referred to as an *instance*, since each procedure call is a separate entity that performs as if called from an application, reserving memory and stack space as required to perform its tasks.

Stored procedures can be nested up to 1,000 levels deep. This limitation helps to prevent infinite loops that can occur when a recursive procedure provides no absolute terminating condition. However, memory and stack limitations of the server can restrict nesting to fewer than 1,000 levels.

The procedure DEPT_BUDGET, in the example employee database, illustrates how a recursive procedure might work. It accepts as input a code DNO, equivalent to DEPT_NO, the key of the DEPARTMENT table. DEPARTMENT has a multiple-level tree structure—each department that is not a head department has a foreign key HEAD_DEPT pointing to the DEPT_NO of its immediate "parent".

The procedure queries for the targeted DEPARTMENT by the input key. It saves the BUDGET value for this row into the output variable TOT. It also performs a count of the number of departments immediately below this one in the departmental structure. If there are no sub-departments, the EXIT statement causes execution to jump right through to the final END ^ statement. The current value of TOT is output and the procedure ends.

```
SET TERM ^ ;
```



```

CREATE PROCEDURE DEPT_BUDGET (
    DNO CHAR(3) )
RETURNS (
    TOT DECIMAL(12,2) )
AS
    DECLARE VARIABLE sumb DECIMAL(12, 2);
    DECLARE VARIABLE rdno CHAR(3);
    DECLARE VARIABLE cnt INTEGER;
BEGIN
    tot = 0;
    SELECT budget FROM department WHERE dept_no = :dno INTO :tot;
    SELECT count(budget) FROM department WHERE head_dept = :dno INTO :cnt;
    IF (cnt = 0) THEN
        EXIT;
    ...

```

If there are sub-departments, execution continues. The input code DNO is used in the WHERE clause of a FOR...SELECT cursor (see below) to target each DEPARTMENT row, in turn, that has this DNO as its HEAD_DEPT code and place its DEPT_NO into local variable, RDNO:

```

    FOR SELECT dept_no FROM department
    WHERE head_dept = :dno
    INTO :rdno DO
    BEGIN
    ...

```

This local variable now becomes the input code for a recursive procedure call. At each recursion, the output value TOT is incremented by the returned budget value, until all of the eligible rows have been processed:

```

        EXECUTE PROCEDURE dept_budget :rdno RETURNING_VALUES :sumb;
        tot = tot + sumb;
    END
    ...

```

Finally, the return value accumulated by the recursions is passed to the caller and the procedure is done:

```

    EXIT; /* the EXIT statement is optional */
END^
COMMIT^

```

Calling the procedure

This time, our procedure has input parameters. Our simple DSQL call might look like this:

```
EXECUTE PROCEDURE DEPT_BUDGET ('600');
```

Or, we can use a replaceable parameter:

```
EXECUTE PROCEDURE DEPT_BUDGET (?);
```

Selectable Stored Procedures

Selectable stored procedures are so-called because they are designed to be executed using a SELECT statement and return a set consisting of multiple rows.

To readers accustomed to the server programming techniques available to other DBM systems, the concept of a stored procedure that outputs rows directly to the calling application or procedure, without any form of intermediate “temporary” table, will be much less familiar than the executable procedure.

Uses for selectable procedures

Selectable procedures can be used to specify virtually any set but they are especially useful when you need a set that cannot be extracted, or is slow or difficult to extract, in a single DSQL statement.

Esoteric sets

The selectable stored procedure technique offers enormous flexibility for extracting sets that defeat the logic available through regular SELECT statement specifications. It literally makes it possible to form a reproducible set from any combination of data you store. You can use calculations and transformations across a number of columns or rows in the set to condition what appears in the output. For example, output sets with running totals are difficult or impossible to achieve from a dynamic query but can be generated and managed speedily and efficiently with a selectable stored procedure.

Selectable procedures can come in handy for generating sets of data that are not stored in the database at all. It's a technique for which we all find a use some time. In the following trivial example, a comma-separated list of strings, each of 20 or fewer characters, is fed in as input. The procedure returns each string to the application as a numbered row:

Example

```
CREATE PROCEDURE BREAKAPART(
    INPUTLIST VARCHAR(1024))
RETURNS (
    NUMERO SMALLINT,
    ITEM VARCHAR(20)
)
AS
    DECLARE CHARAC CHAR;
    DECLARE ISDONE SMALLINT = 0;
BEGIN
    NUMERO = 0;
    ITEM = '';
    WHILE (ISDONE = 0) DO
    BEGIN
        CHARAC = SUBSTRING(INPUTLIST FROM 1 FOR 1);
        IF (CHARAC = '') THEN
            ISDONE = 1;
        IF (CHARAC = ',' OR CHARAC = '') THEN
            BEGIN
```

```

NUMERO = NUMERO + 1;
SUSPEND; /* Sends a row to the row buffer */
ITEM = '';
END
ELSE
  ITEM = ITEM || CHARAC;
INPUTLIST = SUBSTRING(INPUTLIST FROM 2);
END
END ^
COMMIT;
/* */
SELECT * FROM BREAKAPART('ALPHA,BETA,GAMMA,DELTA');
NUMERO  ITEM
-----
1  ALPHA
2  BETA
3  GAMMA
4  DELTA

```

Performance gain for complex sets

Often, complex queries involving many joins or subqueries are too slow to be satisfactory for interactive applications. Some queries can be slow because of unpropitious indexes on foreign keys. Because of its ability to operate on sets in nested loops, a stored procedure is capable of producing the required sets much more quickly and, especially, to begin returning rows much sooner than the conventional SQL sequences permit.

The Technique

The technique for extracting and manipulating the data for the output set uses a cursor to read each row in turn from a select statement into a pre-declared set of variables. Often, it may be the output arguments that the column values are read into, but it can be local variables. Inside the loop, the variables are operated on in whatever manner is required: transformed by calculations, if need be, or used as search arguments for nested loops to pick up values from further queries. At the end of the loop, when all output arguments have their final values, a SUSPEND statement causes execution to pause while the set is passed to the row cache. Execution resumes when the next fetch is called.

As we saw in the BREAKAPART example above, the SUSPEND statement is the element that causes the procedure to deliver a row.

The FOR SELECT...DO construct

To retrieve multiple rows in a procedure, we use the FOR SELECT..DO construct. The syntax pattern is:

```

FOR
  <select-expression>
  INTO <:variable [, :variable [, ...]]
DO
  <compound_statement>;

```

FOR SELECT..DO is a loop construct that retrieves the row specified in the <select-expression> and executes the statement or block following DO for each row in turn, from top to bottom.

The <**select-expression**> can be any select query, using joins, unions, views, other select procedures, CTEs, derived tables, expressions, function calls and so on, in any valid combination. A FOR SELECT differs from a standard DML SELECT statement in that it requires variables into which to receive the columns and fields specified.



When you use a common table expression (CTE) in a FOR loop header, the syntax looks a little bizarre:

Example

```
FOR WITH MY_SET (  
  CHAPTER,  
  VERSE,  
  FINE_DETAIL)  
AS  
  (SELECT * FROM MY_TABLE  
   WHERE ORIGINATOR = CURRENT_USER)  
SELECT CHAPTER, VERSE, FINE_DETAIL FROM MY_SET  
INTO :VARCHAPTER, :VARVERSE, :VARFINE_DETAIL  
DO  
BEGIN  
  ...  
END
```

The <**compound-statement**> can be a single SUSPEND statement or a block of two or more statements. The <compound-statement> can nest other compound statements.

The **INTO <variables>** clause is required and must come last.

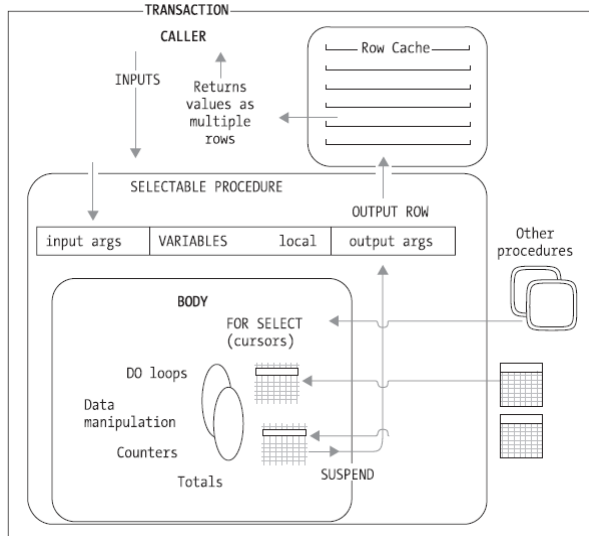
The variable (or variables) must be prefixed with a colon, because the statement is SQL. If there are more than one, they are comma-separated.



The standard DML SELECT syntax simply does not work in a PSQL module since there is no buffer into which to store a “dataset”.

Processing in the loop

Figure 29.2 illustrates the flow of operations when a selectable procedure is called.

Figure 29.2 Operations in a selectable procedure

In the following examples, we'll take a look at how combinations of the operations available in PSQL can meet your less ordinary SQL challenges.

A simple procedure with nested SELECTs

The selectable procedure `ORG_CHART`, which is in the example employee database, takes no input arguments. It uses a `FOR..SELECT` loop to line up a set from a self-referencing join on the `DEPARTMENT` table and pass column values, one row at a time, to a set of variables—some local, some declared as output arguments.

```
CREATE PROCEDURE ORG_CHART
RETURNS (
    HEAD_DEPT CHAR(25),
    DEPARTMENT CHAR(25),
    MNGR_NAME CHAR(20),
    TITLE CHAR(5),
    EMP_CNT INTEGER )
AS
    DECLARE VARIABLE mgr_no INTEGER;
    DECLARE VARIABLE dno CHAR(3);
BEGIN
FOR SELECT
    h.department,
    d.department,
    d.mngr_no,
    d.dept_no
FROM department d
LEFT OUTER JOIN department h
    ON d.head_dept = h.dept_no
```

```

ORDER BY d.dept_no
INTO :head_dept, :department, :mgr_no, :dno
DO
...

```

Each time the loop processes a row, it picks up a key value (MNGR_NO) into the local variable mgr_no. If the variable is null, the procedure fabricates values for the output arguments MNGR_NAME and TITLE. If the variable has a value, it is passed as a search argument to a nested query on the EMPLOYEE table, uniquely targeting a row and extracting the name and job code of a department manager. These values are passed to the remaining output arguments.

```

...
BEGIN
  IF (:mgr_no IS NULL) THEN
    BEGIN
      mgr_name = '--TBH--';
      title = '';
    END
  ELSE
    SELECT
      full_name,
      job_code
    FROM employee
    WHERE emp_no = :mgr_no
    INTO :mgr_name, :title;
    --
    SELECT COUNT(emp_no)
    FROM employee
    WHERE dept_no = :dno
    INTO :emp_cnt;
  ...

```

When all of the outputs for one row are assigned, a SUSPEND statement passes the row to the cache. Execution resumes back at the start of the loop when the next fetch request is made.

```

...
SUSPEND;
END
END^
COMMIT^

```

Calling a selectable procedure

The syntax pattern for calling a selectable procedure is very similar to that for a table or view. The one difference is that a procedure may have input arguments:

```

SELECT <column-list> from <procedure-name> ([argument [, argument ...]])
[WHERE <search-conditions>]
[ORDER BY <order-list>];

```

The procedure name must be specified

Input argument rules are identical to those for executable procedures—see the earlier topic, *Calling rules about input argument values*.

The column-list is a comma-separated list of one or more of the output parameters returned by the procedure, or * to select all columns.

The output set can be limited by a search-condition in a WHERE clause and sorted by an ORDER BY clause.

Calling the org_chart procedure

This procedure has no input parameters, so the SELECT call looks like a simple select on a table, viz.

```
SELECT * FROM ORG_CHART;
```

Selecting aggregates from procedures

In addition to selecting values from a procedure, you can use aggregate functions. For example, to use our procedure to display a count of the number of departments, use the following statement:

```
SELECT COUNT(DEPARTMENT) FROM ORG_CHART;
```

Similarly, to use ORG_CHART to display the maximum and average number of employees in each department, use the following statement:

```
SELECT
    MAX(EMP_CNT),
    AVG(EMP_CNT)
FROM ORG_CHART;
```



If a procedure encounters an error or exception, the aggregate functions do not return the correct values, since the procedure terminates before all rows are processed.

Nested procedures

A stored procedure can itself execute a stored procedure. Each time a stored procedure calls another procedure, the call is said to be nested because it occurs in the context of a previous and still active call to the first procedure. A stored procedure called by another stored procedure is sometimes known as a *nested* or *embedded* procedure.

A sample procedure with nesting

The following procedure returns a listing of users, roles and privileged objects in a database, with their SQL privileges. Inside the procedure, two calls to another procedure are nested. It is necessary to begin by defining and committing the nested procedure—otherwise, the outer procedure will fail to commit. You should always start at the bottom of the “chain” when creating procedures that nest other procedures.

As it happens, this nested procedure performs no SQL. It simply takes an obscure constant, from a set used internally by Firebird to represent object types, and returns a string that is more meaningful to humans:

```
SET TERM ^ ;
CREATE PROCEDURE SP_GET_TYPE (
    IN TYPE SMALLINT )
RETURNS (
    STRING VARCHAR(7) )
```

```

AS
BEGIN
    STRING = 'Unknown';
    IF (IN_TYPE = 0) THEN STRING = 'Table';
    IF (IN_TYPE = 1) THEN STRING = 'View';
    IF (IN_TYPE = 2) THEN STRING = 'Trigger';
    IF (IN_TYPE = 5) THEN STRING = 'Proc';
    IF (IN_TYPE = 8) THEN STRING = 'User';
    IF (IN_TYPE = 0) THEN STRING = 'Table';
    IF (IN_TYPE = 9) THEN STRING = 'Field';
    IF (IN_TYPE = 13) THEN STRING = 'Role';
END^
COMMIT^

```

Now, for the outer procedure. The table it queries for its data is the system table `RDB$USER_PRIVILEGES`. It uses a number of manipulation techniques, including calls to the internal SQL function `CAST()` and to the function `TRIM()`⁴, to massage `CHAR(31)` items into `VARCHAR(31)`. We do this because we want to concatenate some of these string and we don't want the trailing blanks.

```

SET TERM ^ ;
CREATE PROCEDURE SP_PRIVILEGES
RETURNS (
    Q_ROLE_NAME VARCHAR(31),
    ROLE_OWNER VARCHAR(31),
    USER_NAME VARCHAR(31),
    Q_USER_TYPE VARCHAR(7),
    W_GRANT_OPTION CHAR(1),
    PRIVILEGE CHAR(6),
    GRANTOR VARCHAR(31),
    QUALIFIED_OBJECT VARCHAR(63),
    Q_OBJECT_TYPE VARCHAR(7) )
AS
    DECLARE VARIABLE RELATION_NAME VARCHAR(31);
    DECLARE VARIABLE FIELD_NAME VARCHAR(31);
    DECLARE VARIABLE OWNER_NAME VARCHAR(31);
    DECLARE VARIABLE ROLE_NAME VARCHAR(31);
    DECLARE VARIABLE OBJECT_TYPE SMALLINT;
    DECLARE VARIABLE USER_TYPE SMALLINT;
    DECLARE VARIABLE GRANT_OPTION SMALLINT;
    DECLARE VARIABLE IS_ROLE SMALLINT;
    DECLARE VARIABLE IS_VIEW SMALLINT;
BEGIN
    ...

```

First we loop through the table `RDB$USER_PRIVILEGES`, extracting and massaging some values directly into output arguments and others into local variables:

```

    ...

```

4. If you are using a version of Firebird that does not support the `TRIM()` function, you can use `RTRIM()` from the standard external function library *ib_udf*.


```

FOR SELECT
  TRIM(CAST(RDB$USER AS VARCHAR(31))),
  RDB$USER_TYPE,
  TRIM(CAST(RDB$GRANTOR AS VARCHAR(31))),
  TRIM(CAST(RDB$RELATION_NAME AS VARCHAR(31))),
  TRIM(CAST(RDB$FIELD_NAME AS VARCHAR(31))),
  RDB$OBJECT_TYPE,
  TRIM(CAST(RDB$PRIVILEGE AS VARCHAR(31))),
  RDB$GRANT_OPTION
FROM RDB$USER_PRIVILEGES
  INTO :USER_NAME, :USER_TYPE, :GRANTOR, :RELATION_NAME,
  :FIELD_NAME, :OBJECT_TYPE, :PRIVILEGE, :GRANT_OPTION
...

```

Taking the current value of the output variable `USER_NAME`, we query `RDB$ROLES` to get the matching role owner and name, in case the "user" in the current row is actually a role. If it is not a role, these fields will be represented in the output by dashes:

```

...
DO BEGIN
  SELECT
    TRIM(CAST(RDB$OWNER_NAME AS VARCHAR(31))),
    TRIM(CAST(RDB$ROLE_NAME AS VARCHAR(31)))
  FROM RDB$ROLES
  WHERE RDB$ROLE_NAME = :USER_NAME
    INTO :ROLE_OWNER, :ROLE_NAME;
  IF (ROLE_NAME IS NOT NULL) THEN
    Q_ROLE_NAME = ROLE_NAME;
  ELSE
    BEGIN
      Q_ROLE_NAME = '-';
      ROLE_OWNER = '-';
    END
  END
...

```

`WITH GRANT OPTION` is a special privilege that we want to know about in our output so, next, we convert this attribute to 'Y' if it is positive (1) or a blank if it is not:

```

...
IF (GRANT_OPTION = 1) THEN
  W_GRANT_OPTION = 'Y';
ELSE
  W_GRANT_OPTION = '';
...

```

Now, another query into `RDB$ROLES`, this time to find out whether the object that the privilege applies to is a role. If it is, we add a helpful prefix to its object name. If it is not a role, we go on to test whether our object is a column in a table and give it a qualified name if it is.

```

...
IS_ROLE = NULL;
SELECT 1 FROM RDB$ROLES

```

```

        WHERE RDB$ROLE_NAME = :RELATION_NAME
    INTO :IS_ROLE;
    IF (IS_ROLE = 1) THEN
        QUALIFIED_OBJECT = '(Role) '||RELATION_NAME;
    ELSE
    BEGIN
        IF (
            (FIELD_NAME IS NULL)
            OR (RTRIM(FIELD_NAME) = '')) THEN
            FIELD_NAME = '';
        ELSE
            FIELD_NAME = '.'||FIELD_NAME;
        QUALIFIED_OBJECT = RELATION_NAME||FIELD_NAME;
    END
    ...

```

In RDB\$USER_PRIVILEGES, tables and views are both object type 0. That is not good enough for us, so the next query checks the table RDB\$RELATIONS to discover whether this particular object is a view:

```

    ...
    IF (OBJECT_TYPE = 0) THEN
    BEGIN
        IS_VIEW = 0;
        SELECT 1 FROM RDB$RELATIONS
            WHERE RDB$RELATION_NAME = :RELATION_NAME
            AND RDB$VIEW_SOURCE IS NOT NULL
        INTO :IS_VIEW;
        IF (IS_VIEW = 1) THEN
            OBJECT_TYPE = 1;
    END
    ...

```

At this point in our loop, we have almost everything we need. But our object still has its internal number and we still don't know what sort of “user” we have. Users can be other things besides people. Here's where we make the nested calls to get the internal numbers translated to meaningful strings. When that is done, the record is ready to output to the row cache and we call SUSPEND to complete the loop.

Return values

Nested procedure calls in triggers or stored procedures are almost identical to the calls we make from DSQL to execute them. Where the syntax differs is in the handling of the return values. In DSQL, the engine transports the return values back to the client in a record structure. In stored procedures, we use the PSQL keyword RETURNING_VALUES and provide variables to receive the values.

```

    ...
    EXECUTE PROCEDURE SP_GET_TYPE(:OBJECT_TYPE)
    RETURNING_VALUES :Q_OBJECT_TYPE;
    EXECUTE PROCEDURE SP_GET_TYPE (:USER_TYPE)
    RETURNING_VALUES :Q_USER_TYPE;
    SUSPEND;

```

```
END
END^
```

Calling the procedure

This is another simple call:

```
SELECT * FROM SP_PRIVILEGES;
```

If we don't want all of the columns, or we want them in a special order, we can specify the column list we want by specifying the names of the output parameters in the SELECT statement.

For example, suppose we just want to look at the privileges for all humanoid users other than SYSDBA:

```
SELECT
    USER_NAME,
    QUALIFIED_OBJECT,
    PRIVILEGE
FROM SP_PRIVILEGES
WHERE Q_USER_TYPE = 'User'
AND USER_NAME <> 'SYSDBA'
ORDER BY USER_NAME, QUALIFIED_OBJECT;
```

Replaceable search parameters can be used:

```
SELECT
    USER_NAME,
    QUALIFIED_OBJECT,
    PRIVILEGE
FROM SP_PRIVILEGES
WHERE Q_USER_TYPE = ?
ORDER BY USER_NAME, QUALIFIED_OBJECT;
```



*You might find this procedure useful for checking out the SQL privileges in your database. For information on setting up privileges, refer to Chapter 39, **Database-level Security**.*

A procedure with running totals

In this procedure, we process records from the SALES table in the EMPLOYEE database. We keep two running totals: one for each sales representative and one for overall sales. As inputs we have just a start and end date for the group of sales records we want.

```
SET TERM ^;
CREATE PROCEDURE LOG_SALES (
    START_DATE DATE,
    END_DATE DATE)
RETURNS (
    REP_NAME VARCHAR(37),
    CUST VARCHAR(25),
    ORDDATE TIMESTAMP,
    ITEMTYP VARCHAR(12),
    ORDTOTAL NUMERIC(9,2),
```

```

    REPTOTAL NUMERIC(9,2),
    RUNNINGTOTAL NUMERIC(9,2))
AS
    DECLARE VARIABLE CUSTNO INTEGER;
    DECLARE VARIABLE REP SMALLINT;
    DECLARE VARIABLE LASTREP SMALLINT DEFAULT -99;
    DECLARE VARIABLE LASTCUSTNO INTEGER DEFAULT -99;
BEGIN
    RUNNINGTOTAL = 0.00;
    FOR SELECT
        CUST_NO,
        SALES_REP,
        ORDER_DATE,
        TOTAL_VALUE,
        ITEM_TYPE
    FROM SALES
        WHERE ORDER_DATE BETWEEN :START_DATE AND :END_DATE + 1
        ORDER BY 2, 3
    INTO :CUSTNO, :REP, :ORDDATE, :ORDTOTAL, :ITEMTYP
    ...

```

Notice that we are using an ordered set. If you are making a virtual table from a selectable stored procedure and you want an ordered set, it can be valuable to do the ordering set inside the procedure code. The optimizer can help performance here if it has useful indexes to work with; whereas, ordering applied to the output set has, by the virtual nature of the data, no indexes to work with.

Once inside the loop, we begin massaging the data for our row and for the two running totals. We do a little magic to avoid repeating the rep's name—it looks neater on a read-only display—although you would not do this if your application needed to target a row at random and use that column as a search key! We control the customer name in a similar way, to avoid an unnecessary search when the same customer occurs in consecutive records.

```

    ...
DO
BEGIN
    IF(REP = LASTREP) THEN
    BEGIN
        REPTOTAL = REPTOTAL + ORDTOTAL;
        REP_NAME = '';
    END
    ELSE
    BEGIN
        REPTOTAL = ORDTOTAL;
        LASTREP = REP;
        SELECT FULL_NAME FROM EMPLOYEE
        WHERE EMP_NO = :REP
        INTO :REP_NAME;
    END
    IF (CUSTNO <> LASTCUSTNO) THEN

```

```

BEGIN
    SELECT CUSTOMER FROM CUSTOMER
    WHERE CUST_NO = :CUSTNO
    INTO :CUST;
    LASTCUSTNO = CUSTNO;
END
RUNNINGTOTAL = RUNNINGTOTAL + ORDTOTAL;
SUSPEND;
...

```

Our row is now complete and it goes to the row cache with the two running totals updated.

```

END
END^
SET TERM ;^

```

Calling the procedure

Our input arguments are of DATE type, a start date and an end date. The procedure is searching a TIMESTAMP to select the rows for the cursor. It adds a day to the input end date, to ensure that we get every record up to the end of that day. That keeps things simple: when we call the procedure we need only provide the first and last dates, without having to worry about any records with timestamps later than midnight of the final day.

This call is certain to return the whole table:

```
SELECT * FROM LOG_SALES ('16.05.1970', CURRENT_DATE);
```

We might want a procedure like this to be parameterized:

```
SELECT * FROM LOG_SALES (?, ?);
```

Viewing an array through a stored procedure

If a table contains columns defined as arrays, you cannot view the data in the column with a simple SELECT statement, since only the array ID is stored in the table. A stored procedure can be used to display array values, as long as the dimensions and data type of the array column are known in advance.

The JOB table in the sample database has a column named LANGUAGE_REQ containing the languages required for the position. The column is defined as an array of five VARCHAR(15) elements.

The following example uses a stored procedure to view the contents of the column. The procedure uses a FOR...SELECT loop to retrieve each row from JOB for which LANGUAGE_REQ is not null. Then a WHILE loop retrieves each element of the array and returns the value to the calling application.

```

SET TERM ^;
CREATE PROCEDURE VIEW_LANGS
RETURNS (
    code VARCHAR(5),
    grade SMALLINT,
    cty VARCHAR(15),
    lang VARCHAR(15))
AS
    DECLARE VARIABLE i INTEGER;

```

```
BEGIN
  FOR SELECT
    JOB_CODE,
    JOB_GRADE,
    JOB_COUNTRY
  FROM JOB
    WHERE LANGUAGE_REQ IS NOT NULL
  DO
    BEGIN
      i =1;
      WHILE (i <=5) DO
        BEGIN
          SELECT LANGUAGE_REQ[:i] FROM JOB
            WHERE ((JOB_CODE = :code)
              AND (JOB_GRADE = :grade)
              AND (JOB_COUNTRY = :cty))
          INTO :lang;
          i =i +1;
          SUSPEND;
        END
      END
    END ^
  SET TERM ; ^
```

Invoking it:

```
SELECT * FROM VIEW_LANGS;
```

CODE	GRADE	CTY	LANG
=====	=====	=====	=====
Eng	3	Japan	Japanese
Eng	3	Japan	Mandarin
Eng	3	Japan	English
Eng	3	Japan	
Eng	3	Japan	
Eng	4	England	English
Eng	4	England	German
Eng	4	England	French
...			

The procedure could be modified to take input arguments and return a different combination of data as output.

Testing procedures

It should be unnecessary to remind developers of the need to test PSQL modules with rigor and scepticism before launching them into production environments where, on a bad day, they could really do some harm. The parser will spank you

for PSQL coding errors but, as programs, your modules are at least as vulnerable to logic and run-time errors as any application code you write.

For example, our procedure LOG_SALES works fine as long as every sales record has a non-null SALES_REP value. However, this is a nullable column. The procedure happens to be one that generates a result set in which each output row depends on values in the preceding rows. If we don't address the potential effects of null occurring in that key, our procedure is vulnerable to inconsistent results. In the topic *Changing a Stored Procedure*, later in this chapter, we add a safeguard the logic to deal with this particular problem.

Procedures for combined use

BEWARE!

Although it is possible to write a selectable procedure that executes a data-changing operation in the course of constructing an output set, it is not recommended.

A selectable stored procedure is designed to output a set of data to the client, in the transaction context which invokes it. Until the client application has finished using that output set, the transaction remains uncommitted. If DML operations are included in the code that generates the output set, those DML requests remain uncommitted until the transaction is completed by the client.

In particular, data have the potential to be stored inconsistently if values from the stored procedure output are passed as parameters to operations in other transactions.

Using the Internal RDB\$DB_KEY

Firebird inherited a sparsely documented feature that can speed up query performance in some conditions. This is RDB\$DB_KEY—usually referred to simply as the *db_key*—a cardinality key maintained by the database engine for internal use in query optimization and record version management.

Inside the transaction context in which it is captured, it represents a row's position in the table.

About RDB\$DB_KEY

The first lesson to learn is that RDB\$DB_KEY is a raw position, related to the database itself and not to a physical address on disk. The second is that the numbers do not progress in a predictable sequence. Don't consider performing calculations involving their relative positions! The third lesson is that they are volatile—they change after a backup and subsequent restore and, sometimes, after the transaction is committed. It is essential to understand the transience of the *db_key* and to make no assumptions about its existence once an operation that refers to it is committed or rolled back.

Size of RDB\$DB_KEY

For tables, RDB\$DB_KEY uses 8 bytes. For a view, it uses as many multiples of 8 bytes as there are underlying tables. For example, if a view joins three tables, its RDB\$DB_KEY uses 24 bytes. This is important if you are working with stored procedures and want to

```
store RDB$DB_KEY in a variable. You must use a CHAR(n) data type of the correct
length.

By default, db_keys are returned as hex values— two hex digits represent each byte,
causing 16 hex digits to be returned for 8 bytes. Try it on one of your sample tables in isql:

SQL> SELECT RDB$DB_KEY FROM MYTABLE;
RDB$DB_KEY
=====
000000B600000002
000000B600000004
000000B600000006
000000B600000008
000000B60000000A
```

Benefits

Because a RDB\$DB_KEY marks the raw position of a row, it is faster for a search than even a primary key. If for some special reason a table has no primary key or active unique index, or is primed on a unique index that is allowed to contain nulls, it is possible for exact duplicate rows to exist. Under such conditions, RDB\$DB_KEY is the only way to identify each row unequivocally.

Several kinds of statements run faster when moved into a stored procedure using a RDB\$DB_KEY—typically, updates and deletions with complex conditions. For inserts—even huge batches—the RDB\$DB_KEY is of no avail, since there is no way to ascertain what the values will be.

However, if the database pages being searched for update or delete are already in main memory, the difference in access speed is likely to be negligible. The same is true if the searched set is quite small and all of the searched rows are close to one another.

Optimization of queries

Performance problems are likely if you try to run a DSQL update like the following example against a huge table:

```
UPDATE TABLEA A
  SET A.TOTAL = (
    SELECT SUM (B.VALUEFIELD)
    FROM TABLEB B
    WHERE B.FK= A.PK)
where <conditions..>
```

If you run the same operation often and it affects a lot of rows, it would be worth the effort to write a stored procedure that handles the correlated total on each row without needing to perform a subquery:

```
CREATE PROCEDURE ...
..
AS
BEGIN
  FOR SELECT B.FK, SUM(B.VALUEFIELD) FROM TABLEB B
  GROUP BY B.FK
  INTO:B_FK, :TOTAL DO
```



```

UPDATE TABLEA SET A.TOTAL = :TOTAL
WHERE A.PK= :B_FK
AND ...
END

```

Although speedier, it still has the problem that records in A have to be located by primary key each time a new pass of the FOR...DO loop happens.

Some people claim better results with this alien syntax:

```

...
DECLARE VARIABLE DBK CHAR(8); /* 8 CHARS FOR A TABLE'S DBKEY */
...
FOR SELECT
    B.FK,
    SUM(B.VALUEFIELD),
    A.RDB$DB_KEY
FROM TABLEB B
JOIN TABLEA A ON A.PK = B.FK
WHERE <conditions>
GROUP BY B.FK, A.RDB$DB_KEY
INTO :B_FK, :TOTAL, :DBK DO
    UPDATE TABLEA SET A.TOTAL = :TOTAL
    WHERE A.RDB$DB_KEY = :DBK;

```

NOTE Firebird doesn't need the key column to form the join but it does need to be present in the SELECT list in order for the GROUP BY clause to be valid.

The benefits of this approach are:

- 1 Filtering of the common records for A and B is efficient where the optimizer can make a good filter from the explicit join.
- 2 If the join can apply its own search clause, there is a gain in getting the extra filtering before the update checks its own condition.
- 3 Rows from the right-side table (A) are located by raw db_key values, extracted at the time of the join, making the search faster than looking through the primary key or its index.

Inserting

Since inserting does not involve a search, the simplest insert operations—for example, reading constant values from an import set in an external table—are not affected by the need to locate keys.

Not every insert statement's VALUES input set is obtained so simply, however. It can be a very complicated set of values derived from expressions, joins or aggregations. In a stored procedure, an INSERT operation may well be branched into the ELSE sub-clause of an IF (EXISTS(...)) predicate. For example,

```

...
IF EXISTS(SELECT...) THEN
    ...
ELSE
BEGIN

```

```

INSERT INTO TABLEA
SELECT
    C.PKEY,
    SUM(B.AVALUE),
    AVG(B.BVALUE),
    COUNT(DISTINCT C.XYZ)
FROM TABLEB B JOIN TABLEC C
    ON B.X = C.Y
WHERE C.Z = 'value'
AND C.PKEY NOT IN(SELECT PKEY FROM TABLEA)
GROUP BY C.PKEY;
END
...

```

Implementing this in a stored procedure:

```

...
FOR SELECT
    C.PKEY,
    SUM(B.AVALUE),
    AVG(B.BVALUE),
    COUNT(DISTINCT C.XYZ)
FROM TABLEB B JOIN TABLEC C
    ON B.X = C.Y
WHERE C.Z = 'value'
AND C.PKEY NOT IN (SELECT PKEY FROM TABLEA)
GROUP BY C.PKEY
    INTO :C_KEY, :TOTAL, :B_AVG, :C_COUNT DO
BEGIN
    SELECT A.RDB$DBKEY FROM TABLEA A
    WHERE A.PKEY = :C_KEY
    INTO :DBK;
    IF (DBK IS NULL) THEN /* the row doesn't exist */
        INSERT INTO TABLEA(PKEY, TOTAL, AVERAGE_B, COUNT_C)
        VALUES(:C_KEY, :TOTAL, :B_AVG, :C_COUNT);
    ELSE
        UPDATE TABLEA SET
            TOTAL = TOTAL + :TOTAL,
            AVERAGE_B = AVERAGE_B + :B_AVG,
            COUNT_C = COUNT_C + :C_COUNT
        WHERE A.RDB$DB_KEY = :DBK;
    END
...

```

Duration of validity

By default, the scope of a db_key is the current transaction. You can count on it to remain valid for the duration of the current transaction. A commit or rollback will cause the RDB\$DB_KEY values you had to become unpredictable. If you are using

CommitRetaining, the transaction context is retained, blocking garbage collection and thus, preventing the old db_key from being “recycled”. Under these conditions, the RDB\$DB_KEY values of any rows affected by your transaction remain valid until a “hard” commit or rollback occurs.

After the hard commit or rollback, another transaction might delete a row that was isolated inside your transaction’s context and was thus considered “existent” by your application. Any RDB\$DB_KEY value might now point to a non-existent row. If there is a long interval between the moment when your transaction began and when your work completes, you should check that the row has not been changed or locked by another transaction in the meantime.

Some application interfaces, for example, IB Objects, are super-smart about inserts and can prepare a “slot” for a newly-inserted row in the client buffers to short-circuit the refresh following the commit. Such features are important for performance across the network. However, “smarts” like this are based on exact, real keys. Since the db_key is merely a proxy key for a set that has been derived from previously committed data, it has no meaning for a new row—it is not available for spot updates of the client buffers.

Changing the scope of duration

The default duration of RDB\$DB_KEY values can be changed at connection time, by using the API parameter *isc_dpb_dbkey_scope*. Some development tools—for example, the IB Objects components in ObjectPascal environments—surface it in a connectivity class. However, it is not recommended to extend the scope of db_keys in a highly interactive environment, since it will disable garbage collection, with the unwanted side-effect of causing your database file to grow at an alarming rate and slowing down performance until the system hangs or crashes.

Do not pool connections having non-default db_key scope.

RDB\$DB_KEY with multi-table sets

All tables maintain their own distinct, 8-byte RDB\$DB_KEY columns. Views and joins generate run-time db_keys by concatenating those of the rows in the source tables. If you use RDB\$DB_KEY in multi-table sets, be very careful to qualify each one accurately.

RDB\$DB_KEY can not be used across table boundaries. There is no possibility of establishing a dependency relationship between the RDB\$DB_KEY of one table and another, except in re-entrant (self-referencing) joins.

Changing a Stored Procedure

There are three different DDL statements to change a stored procedure. Although they are similar in structure, they behave in different ways.

- ALTER PROCEDURE, which changes the definition of an existing stored procedure while preserving its dependencies on other objects.
- RECREATE PROCEDURE, which works even if the named procedure does not exist. If it does exist, the existing version is dropped and then recreated. Existing dependencies do not survive.

- `CREATE OR ALTER PROCEDURE`,⁵ which gives the best of both. If the procedure exists, `ALTER` rules apply and dependencies are preserved. If not, it will work exactly as `CREATE PROCEDURE` does.

Any of these operations will fail with an exception if any change is attempted that would break a dependency.

Effect on applications

Changes made internally to a procedure are transparent to all client applications that use the procedure: you do not have to rebuild the applications unless the changes affect the interface between the caller and the procedure—type, number or order of input and output arguments.

Syntax for changing procedures

Except for keyword you choose to effect a change to a stored procedure, the syntax of the statements for each is the same as for `CREATE PROCEDURE`. Just as with any compiled or interpreted module, there is no way to direct a change at a element without recreating the whole module. Every "alteration", regardless of the keyword you choose to specify the operation, is a matter of creating a new source version and a new binary-coded object.

The syntax pattern is:

```
{CREATE | ALTER | RECREATE | CREATE OR ALTER} PROCEDURE name
[(var datatype [, var datatype ...])]
[RETURNS (var datatype [, var datatype ...])]
AS
procedure_body;
```

ALTER PROCEDURE

For `ALTER PROCEDURE`, the procedure name must be the name of an existing procedure.

This is the low-impact way to change procedure code because, if it has dependencies that are not logically affected by the changes, there will be no structural side-effects.

In general, dependencies involving other objects that depend on the changed procedure are not affected, either⁶. However, if the changes to the stored procedure alter the definitions of its input or output arguments, it will be necessary to perform a `RECREATE PROCEDURE` on any other stored procedure that is called during the execution.

RECREATE PROCEDURE

`RECREATE PROCEDURE` is identical to `CREATE PROCEDURE`, except that it causes any existing procedure of the same name to be subjected internally to a full `DROP PROCEDURE` operation before the new binary object is created. The procedure name does not have to exist.

5. Not supported in v.1.0.x.

6. Some versions of Firebird 1.0.x exhibit a bug, whereby dependent objects will throw an exception when the depended-on object is recompiled, even if the interface between the objects is not affected by the change.

You can use it like ALTER PROCEDURE but it will not preserve existing dependencies. The operation will be blocked if there are dependent objects (for example, views or other procedures that refer to the procedure).

The procedure name need not exist, but take care with case-sensitivity of object names if quoted identifiers were used when creating the procedure. For example, suppose you created this procedure:

```
CREATE PROCEDURE "Try_Me"
  RETURNS (AWORD VARCHAR(10))
AS
BEGIN
  AWORD = "turtle";
END ^
```

Now, you decide to change it using RECREATE PROCEDURE:

```
RECREATE PROCEDURE Try_Me
  RETURNS (AWORD VARCHAR(10))
AS
BEGIN
  AWORD = "Venezuela";
END ^
```

The original procedure, with its case-sensitive name "Try_Me" remains, unchanged. The “recreated” procedure is a new and quite separate object with the case-insensitive name TRY_ME.

CREATE OR ALTER PROCEDURE

This tolerant syntax creates a new procedure if there is none of the supplied name, or alters an existing procedure of that name.

Fixing the LOG_SALES procedure

As an example, we are going to fix that procedure, LOG_SALES, that promises to bite us because we overlooked a nullable key. Here is the block that could cause the problems:

```
CREATE PROCEDURE LOG_SALES (...)
...
DO
BEGIN
  IF(REP = LASTREP) THEN /* will be false if both values are null */
  BEGIN
    REPTOTAL = REPTOTAL + ORDTOTAL;
    REP_NAME = '';
  END
  ELSE
  BEGIN
    REPTOTAL = ORDTOTAL;
    LASTREP = REP;
    SELECT FULL_NAME FROM EMPLOYEE
    WHERE EMP_NO = :REP
    INTO :REP_NAME; /* will return null if variable REP is null */
  END
```

```
...
END ^
```

We fix the logic to handle nulls (grouped together at the end of the cursor, because the set is ordered by this column) and use CREATE OR ALTER to update the code:

```
CREATE OR ALTER PROCEDURE LOG_SALES (...
...
DO
BEGIN
    /* ***** */
    IF((REP = LASTREP) OR (LASTREP IS NULL)) THEN
    /* ***** */
    BEGIN
        REPTOTAL = REPTOTAL + ORDTOTAL;
        REP_NAME = '';
    END
    ELSE
    BEGIN
        REPTOTAL = ORDTOTAL;
        LASTREP = REP;
    /* ***** */
    IF (REP IS NOT NULL) THEN
        SELECT FULL_NAME FROM EMPLOYEE
        WHERE EMP_NO = :REP
        INTO :REP_NAME;
    ELSE
        REP_NAME = 'Unassigned';
    /* ***** */
    END
    ...
END ^
COMMIT ^
```

“Object is in use” error

Committing the change will throw the notorious error (discussed in the previous chapter) if any user is currently using the procedure, or another object that depends on it. Even if we clear that hurdle, the new version of the procedure won’t be immediately available on Superserver if the old version is still in the cache. All users must log out and, when they log in again, they will see the new version.

On Classic and Superclassic, the new version will be available to the next client that logs in.

Dropping a Stored Procedure

The DROP PROCEDURE statement deletes an existing stored procedure from the database. You can use this statement anywhere it is possible to use DDL statements.



DDL statements can not be executed as PSQL statements, per se. However, a DDL statement can be passed via an EXECUTE STATEMENT construct. Is it necessary to caution the reader against using EXECUTE STATEMENT to have a procedure drop itself?

Syntax The syntax pattern is simple:

```
DROP PROCEDURE procedure-name;
```

The procedure name must be the name of an existing procedure. Take care with case-sensitivity of object names if quoted identifiers were used when creating the procedure.

The following statement deletes the LOG_SALES procedure:

```
DROP PROCEDURE LOG_SALES;
```

Restrictions

The following restrictions apply to dropping a procedure:

- Only the owner of the procedure or a user with SYSDBA privileges can drop it.
- A procedure that is in use by any transaction can not be dropped. This will be an especial issue in systems where procedures are being called from transactions that are committed using COMMIT WITH RETAIN
- If any other objects in the database refer to or call the procedure, it will be necessary to alter the dependent objects to remove the references and commit that work, before the procedure can be dropped.
- A recursive procedure can not be dropped without first removing the recursive calls and committing that change. Similar complications apply to a procedure that calls other procedures that, in turn, call the procedure you want to drop. All such dependencies must be removed and committed before the procedure is free to be dropped.

Run-time PSQL

The SQL language extension EXECUTE BLOCK makes a block of non-persistent PSQL available to be executed as a dynamic statement. It can be written to execute some DML on database objects as an executable procedure would do; or in a form similar to a selectable procedure, designed to return a set of virtual data to the caller.

EXECUTE BLOCK is a part of the dynamic DML lexicon—it cannot be used inside a compiled PSQL module. However, the language subset for composing the block of run-time code in your client application is PSQL.

Syntax The syntax model is very similar to that for stored procedures:

```
EXECUTE BLOCK [ (inparam <data type> = ?, inparam <data type> = ?, ...) ]
[ RETURNS (outparam <data type>, outparam <data type>, ...) ]
AS
    [DECLARE VARIABLE var <data type>;
    ...]
BEGIN
    ...
END
```

If there are input parameters, the EXECUTE BLOCK statement must be prepared. The *inparam <data type> = ?* phrase is the only valid way to specify an input parameter: you cannot supply a literal argument.

Coding EXECUTE BLOCK

We'll set out to write a block of fairly useless code, that might nevertheless serve as an example of potential uses of run-time blocks to create sets of virtual data.



Two SET TERM commands are included here because you'll need them if you run any EXECUTE BLOCK statements in isql. In fact, in isql, at the time of writing, you cannot run blocks with input parameters at all, because it has no way to assign values to parameters. Feature request, anyone?

Our block takes a year as input and returns the day of the week for the first day of each month of that year.

```
SET TERM ^;
EXECUTE BLOCK
    (qyear smallint = ?)
...
```

Notice the use of '?' as the placeholder for input parameters. A colon prefix is not valid because it conflicts with the use of the colon prefix in the body of the PSQL block, required whenever a variable is involved in an SQL statement.

From here on, coding is very much like coding a stored procedure. You declare output variables the same way as you do for a stored procedure. But getting the output to show up in the client is different, as we shall see.

```
...
returns (
    vdate date,
    vdow varchar (10))
as
    declare vday smallint;
    declare vmonth smallint = 1;
    declare nextdate varchar(10);
    declare exestump varchar (35) = 'SELECT EXTRACT (WEEKDAY FROM CAST('';
    declare exestring varchar(200);
    declare exestring1 varchar(200);
begin
    while (vmonth < 13) do
    begin
        nextdate = vmonth || '/01/' || qyear;
        exestring = exestump || nextdate || ''' AS DATE)) from RDB$DATABASE';
        exestring1 = 'SELECT CAST (''' || nextdate || ''' as date) from RDB$DATABASE';
        EXECUTE STATEMENT exestring into :vday;
        EXECUTE STATEMENT exestring1 into :vdate;
        vdow = case
            when (vday = 0) then 'Sunday'
            when (vday = 1) then 'Monday'
            when (vday = 2) then 'Tuesday'
```



```

when (vday = 3) then 'Wednesday'
when (vday = 4) then 'Thursday'
when (vday = 5) then 'Friday'
when (vday = 6) then 'Saturday' end;
...

```

In order to get any output from a block, you must use `SUSPEND`, even if the output is just a single row.

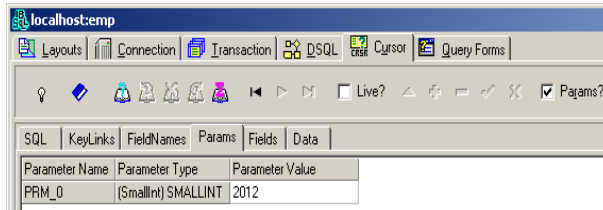
```

...
SUSPEND;
monthnumber = monthnumber + 1;
end
end^
SET TERM ;^

```

Testing the sample

We cannot test a block like this successfully in *isql* as it stands at v.2.5.1. Other tools and interfaces that have the ability to prepare statements and accept parameters have no problem. To test and illustrate this sample, the `EXECUTE BLOCK` code was pasted into the cursor interface in the free `IB_SQL` toolset for Windows, from www.ibobjects.com and prepared. The input parameter that was returned was not the name we gave it in the declaration (`QYEAR`) but one called `PRM_0` that was internally generated by the engine:



The following screenshot is the output:

Figure 29.3 Output from `EXECUTE BLOCK` sample

VDATE	VDW
1/01/2012	Sunday
1/02/2012	Wednesday
1/03/2012	Thursday
1/04/2012	Sunday
1/05/2012	Tuesday
1/06/2012	Friday
1/07/2012	Sunday
1/08/2012	Wednesday
1/09/2012	Saturday
1/10/2012	Monday
1/11/2012	Thursday
1/12/2012	Saturday

Of course, getting sets of “non-data” is not all you can do with EXECUTE BLOCK. You can compose a block to do all sorts of useful tasks in the database, with or without output. The **Firebird 2.5 Language Reference Update** provides a number of handy examples. You can read this volume online in the [documentation section](#) of the Firebird website or find it in the /doc/ sub-directory of your Firebird installation.

Using EXECUTE BLOCK in applications

Most existing interfaces provide the API calls necessary to run an EXECUTE BLOCK statement that simply executes some DML without taking parameters or returning output. When inputs and outputs are involved, an EXECUTE BLOCK statement must be prepared in order to set up the structures needed to pass parameters and receive output. As has already been pointed out, Firebird’s own *isql* tool is one application that does not have this capability.

Given that DSQL does not have a PREPARE statement in its lexicon, it may be necessary to update the drivers, components and tools that you use for interfacing with the Firebird API in your application development, to a version that supports the explicit preparing functions that are required.

.

C H A P T E R

30

TRIGGERS

Triggers are a key element among the capabilities provided by Firebird for implementing business rules centrally inside the database management system. A trigger is a self-contained module that executes automatically when a request is executed that will change the state of data in a table.

Triggers are stored procedures and the PSQL techniques for writing executing code for them are no different to those for callable procedures. However, triggers can not be invoked by applications or other procedures. Accordingly, they can not take inputs or pass outputs as other procedures do. In addition, PSQL includes certain contextual language extensions applicable only to trigger modules.

Classes of Trigger

All versions of Firebird support table-level triggers—PSQL blocks that can be written to execute automatically whenever the engine receives a request to modify, add or remove data. Versions 2.1+ also support higher-level triggers at both transaction and connection levels.



Triggers are never called by procedures, other triggers or applications. They do not support input or output arguments at all. Be aware, though, that triggers which perform DML on other tables (or inadvisably, on themselves!) will cause the triggers on those tables to fire. If you don't keep a good handle on what a trigger is going to do under all conditions, the results may not be pretty!

Table-level Triggers

Table-level triggers execute at row level, once each time the row image changes. Firebird supports a high degree of granularity for defining the timing, sequence and conditions under which a particular trigger module will fire. Multiple modules can be defined for each phase (before and after) and event (insert, update, delete or, conditionally, all three).

Triggers are part of the work of the transaction in which a DML event changes the state of a row. If the transaction commits successfully, all of the trigger actions will “take”. If the transaction is rolled back, the trigger actions are all undone.

Table-level triggers can be defined for a table and for any views it subtends. Naturally updatable views share the table’s triggers. View triggers are usually written to make read-only views updatable. If view triggers fire, table triggers do not fire; and vice versa.

“Database Triggers”

Triggers that fire in events beyond the boundaries of statement execution are referred to, somewhat erroneously, as *database triggers*. The term encompasses PSQL modules that you can define to be executed at events occurring at either transaction level or session level.

Firebird supports both transaction-level and session-level triggers from v.2.1 onward. For details, see *Higher-level Triggers*, later in this chapter.

DDL Triggers

DDL triggers fire when database objects are created, altered or dropped. A typical use is to block DDL activity by unauthorised users. Firebird versions current at the time of this writing do not support DDL triggers but they are under development for future versions.

About Table-level Triggers

A table-level trigger can execute in one of two phases relative to the execution of the requested action events (change of data state): **before** the write or **after** it. It can apply to one of three DML action events: inserting, updating or deleting.

The trigger actions for two or three DML events can be merged into one “before” or “after” trigger module, with processing conditioned on the Boolean context variables INSERTING, UPDATING and DELETING. They are known as *multi-action triggers*.

v.1.0.x Multi-action triggers are not supported in v.1.0.x.

Phase and event

Table 30-1 summarizes the eight kinds of table-level trigger module.

Table 30.1 Phase/event combinations for table-level trigger modules

Trigger kind	Description	Version
BEFORE INSERT	Fires before a new row is created. Allows input values to be changed.	All
AFTER INSERT	Fires after a new record version is created. Does not allow input values to be changed. Usually used to modify other tables.	All

Trigger kind	Description	Version
BEFORE UPDATE	Fires before a new record version is created. Allows input values to be changed.	All
AFTER UPDATE	Fires after a new record version is created. Does not allow input values to be changed. Usually used to modify other tables and/or post events.	All
BEFORE DELETE	Fires before a row is deleted. Does not accept changes to any columns in the row.	All
AFTER DELETE	Fires after the row has been deleted. Does not accept changes to any columns in the row. Usually used to modify other tables and/or post events.	All
BEFORE <event> OR <event> [OR <event>]	Fires before any requested data state change is executed. DML event actions must be coded conditionally. "Deleting" action can not change any columns in the row.	1.5+
AFTER <event> OR <event> [OR <event>]	Fires after any requested data state change is executed. DML event actions must be coded conditionally. Actions can not change any columns in the row. Usually used to modify other tables and/or post events.	1.5+

Sequence

Firebird allows multiple trigger modules for any phase/event combination. There is probably some practical limit, but it is safe to say you can create as many as you need, using whole numbers between 0 and 32,767. The default is sequence number ("POSITION") zero. It is good science to set an order of execution by numbering the triggers, but explicit sequencing is optional. If sequence numbers are present, the triggers will be executed in ascending order. Numbers don't have to be unique and the sequence can have gaps.

Suites of triggers for a phase/event with the default POSITION 0 will be executed in the *alphabetical order* of their names. The same can be expected if you have groupings of triggers sharing the same non-zero sequence number.

The following example demonstrates how four update triggers for the ACCOUNT table would be fired:

```
CREATE TRIGGER BU_ACCOUNT5 FOR ACCOUNT
  ACTIVE BEFORE UPDATE POSITION 5 AS ...
CREATE TRIGGER BU_ACCOUNT0 FOR ACCOUNT
  ACTIVE BEFORE UPDATE POSITION 0 AS ...
CREATE TRIGGER AU_ACCOUNT5 FOR ACCOUNT
  ACTIVE AFTER UPDATE POSITION 5 AS ...
CREATE TRIGGER AU_ACCOUNT3 FOR ACCOUNT
  ACTIVE AFTER UPDATE POSITION 3 AS ...
```

Someone updates some rows in ACCOUNTS:

```
UPDATE ACCOUNT
  SET C = 'CANCELED'
  WHERE C2 = 5;
```

This is the sequence of events on each affected row:

- Trigger BU_ACCOUNT0 fires

- Trigger BU_ACCOUNT5 fires
- The new record version is written to disk
- Trigger AU_ACCOUNT3 fires
- Trigger AU_ACCOUNT5 fires

Status Active/Inactive

A trigger can be active or inactive. Only active triggers fire. With care, a trigger can be deactivated temporarily to perform some sandboxed task, using an ALTER TRIGGER DDL request. Refer to the notes on ALTER TRIGGER for details about deactivating a trigger.

Creating Table-level Triggers

A trigger is defined with the CREATE TRIGGER statement, which is composed of a header and a body.

The table-level trigger header, which is quite different to a stored procedure header, contains:

A **trigger name**, unique within the database.

A **table name**, preceded by the keyword FOR, identifying the table with which to associate the trigger.

Attributes that determine state, phase, DML event and, optionally, sequence.

The trigger body, like a stored procedure body, contains:

An optional list of one or more **local variable declarations**, their data types and, if required, their default values.

A **block of statements** in Firebird procedure and trigger language, bracketed by BEGIN and END. These statements are performed when the trigger fires. A block can itself include other blocks, so that there may be many levels of nesting.

Syntax For all versions of Firebird, the CREATE TRIGGER syntax pattern for table-level triggers is:

```
CREATE TRIGGER trigger-name FOR {table | view}
[ACTIVE | INACTIVE]
{BEFORE | AFTER} {DELETE | INSERT | UPDATE |
{DELETE OR {[INSERT [OR UPDATE]]} | {INSERT OR [...] | {UPDATE OR [...]}}
[POSITION number]
AS <trigger_body> ^
    <trigger_body>=[<variable_declaration_list>] <block>
    <variable_declaration_list> =DECLARE VARIABLE variable data type | [TYPE OF] <domain>;
[DECLARE VARIABLE variable data type [TYPE OF] <domain>;...]
<block>=
BEGIN
<compound_statement>[<compound_statement>...]
END
```

```
<compound_statement>=<block>|statement;
```

An alternative syntax

v.2.1 +

From Firebird 2.1 onward, an alternative, SQL-2003-compliant syntax is available for trigger headers on tables and views. In this syntax you create a trigger ON a relation, instead of FOR it and the clause order is different:

```
CREATE TRIGGER trigger-name
[ACTIVE | INACTIVE]
{BEFORE | AFTER} {DELETE | INSERT | UPDATE |
{DELETE OR {[INSERT [OR UPDATE]]} | {INSERT OR [...] } | {UPDATE OR [...]}}
[POSITION number]
ON {table | view}
AS <trigger_body>
...
```

Everything else about the table-level trigger is the same in both forms.

Header elements

Everything preceding the AS clause forms the trigger header. The header must specify the unique name of the trigger and the name of an existing, committed table or view that it is to belong to.

Naming triggers

The syntax requires that the trigger name be unique among all trigger names in the database. It is good practice to adopt some convention for naming triggers that is meaningful to you and obvious to others who will work with your database. The author uses a “formula” identifying phase and event (BI | AI | BU | AU | BD | AD | BA | AA, the latter two representing “Before All” and “After All”), table-name and, if relevant, sequence number.

For example, a Before Insert trigger for the Customer table might be named 'BI_Customer1'.

```
CREATE TRIGGER BI_CUSTOMER1 FOR CUSTOMER...
```

Trigger attributes

The remaining attributes in the trigger header are:

The trigger **status**, ACTIVE or INACTIVE, determines whether the trigger will be “up-and-running” when it is created. ACTIVE is the default. Deactivating a trigger is useful during development and testing and for special jobs a DBA might do with exclusive access to the database.

The **phase** indicator, BEFORE or AFTER, determines the timing of the trigger relative to the write action being executed by the DML event.

The **DML event** indicator specifies the type of SQL operation that shall trigger execution of the module—INSERT, UPDATE, or DELETE or a multi-action directive such as INSERT OR UPDATE.

Multi-action triggers

The optional `<event> OR <event>..` extension, available from v.1.5 onwards, allows two or three events to be coded conditionally into a single module. For example, `..BEFORE INSERT OR UPDATE OR DELETE..` allows you to provide actions for all three events. The Boolean context variables `INSERTING`, `UPDATING` and `DELETING` support the branching logic that is necessary to make the trigger behave appropriately for each specified event.

The optional sequence indicator, ***POSITION <number>***, specifies when the trigger is to fire in relation to other trigger modules for the same phase and event.

The trigger body

In all Firebird code modules, the body consists of an optional list of local variable declarations followed by a block of statements. Refer to the topic *Variables and Parameters* in Chapter 28 for guidance as to declaring them.

Programming a trigger body is exactly the same as programming a stored procedure body—refer to the preceding chapter.

- Triggers can embed stored procedures. The calling rules are exactly the same for trigger modules as for stored procedures.
- Triggers can process cursors, perform operations on other tables and post events. They can throw and handle exceptions, including those raised from nested procedures. Exception-handling techniques are discussed in the next chapter.

Of interest to us in this chapter are some special extensions PSQL provides to support the trigger context and some special roles for triggers in implementing and enforcing business rules.

Special PSQL for Table-level Triggers

Two special PSQL elements are available for triggers: the `NEW` and `OLD` context variables and the Boolean event context variables `INSERTING`, `UPDATING` and `DELETING`.

Event variables

The Boolean context variables `INSERTING`, `UPDATING` and `DELETING` to support the conditional branching for the multi-event trigger feature. Possible syntax patterns would be:

```
IF ({INSERTING | UPDATING | DELETING}
OR {UPDATING | DELETING | INSERTING}
[OR {DELETING | INSERTING | UPDATING}]) THEN ...
```

Follow through the examples in the topic *Special PSQL for Table-level Triggers* to see these useful predicates at work.

NEW and OLD context arrays

The `NEW` and `OLD` context variables are a trigger-specific extension to PSQL that enable your code to refer to the existing (“old”) and requested (“new”) values of each column. The `NEW.*` variables have values in `INSERT` and `UPDATE` events; `OLD.*` in `UPDATE` and `DELETE` events. `NEW.*` in delete events and `OLD.*` in insert events are null. The

applicable old and new counterparts are available for every column in a table or view, even if the columns themselves are not referred to in the DML statement.

The OLD.* values (if available) can be used as variables within the trigger but they are read-only. The NEW.* values (if available) are read-write during the BEFORE phase and read-only during the AFTER phase. If you want to manipulate them as variable values in an “After” trigger, move the values into local variables and refer to those.

Uses for NEW and OLD

NEW and OLD variables are the essential tool for harnessing the power of Firebird triggers to develop databases that take care of data integrity independently of humans and external programming. They can be used to:

- provide valid default values under any conditions (as contrasted with the column or domain definition default, that works only on inserts and then only if the defaulted column is missing from the insert list)
- validate and, if required, transform user input
- supply keys and values for performing automatic updates to other tables
- implement auto-incrementing keys by means of generators (sequences). You can read about this technique in the topic *Implementing Auto-Incrementing Keys* later in this chapter.

New values for a row can only be altered by before actions. A trigger that fires in the AFTER phase and tries to assign a value to NEW.column will have no effect. In higher versions, it will actually cause an exception—so fix that legacy code!

NEW values are writeable all through the BEFORE phase and take up a reassignment of their value *inside the trigger* immediately.



The new record version will not receive any reassignments until all BEFORE triggers have completed. At that point, the NEW values become read-only. Hence, if you have multiple triggers adjusting the same NEW values, it is important to ensure that they all have different POSITION numbers, correctly ordered.

Table-level triggers at work

To demonstrate some of the ways triggers can support integrity and reduce the work required by applications, we take a look at a few typical tasks that can be done with triggers.

Transformations

A NEW variable can be used to transform a value into something else. A common trick is to use a trigger to maintain a “proxy” column for doing case-insensitive searches on another column which may be any mixture of case. The trigger reads the NEW value of the mixed-case column, converts it to upper-case and writes it to the NEW value of the proxy column. Ideally, the column being “proxied” should have a NOT NULL constraint to ensure that there will always be a value to find:

```
CREATE TABLE MEMBER (
  MEMBER_ID INTEGER NOT NULL PRIMARY KEY,
  LAST_NAME VARCHAR(40) NOT NULL,
  FIRST_NAME VARCHAR(35),
  PROXY_LAST_NAME VARCHAR(40),
  MEMBER_TYPE CHAR(3) NOT NULL,
```

```

        MEMBERSHIP_NUM VARCHAR(13),
        ....);
COMMIT;
--
SET TERM ^;
CREATE TRIGGER BA_MEMBER1 FOR MEMBER
    ACTIVE BEFORE INSERT OR UPDATE
    POSITION 0
AS
BEGIN
    ...
    NEW.PROXY_LAST_NAME = UPPER(NEW.LAST_NAME);
    ...
END ^

```

All kinds of transformations are possible. Suppose we want to issue membership numbers (MEMBERSHIP_NUM) made up of the MEMBER_TYPE followed by a string of 10 digits, left-padded with zeros, based on the generated primary key of the MEMBER table. We can have them automatically generated in a BEFORE INSERT trigger :

```

CREATE TRIGGER BI_MEMBER2 FOR MEMBER
    ACTIVE BEFORE INSERT
    POSITION 2
AS
    DECLARE VARIABLE ID_AS_STRING VARCHAR(10);
BEGIN
    ID_AS_STRING = CAST(NEW.ID AS VARCHAR(10));
    WHILE (NOT (ID_AS_STRING LIKE '_____%')) /* 10-character mask */
    DO
        ID_AS_STRING = '0' || ID_AS_STRING;
    NEW.MEMBERSHIP_NUM = NEW.MEMBER_TYPE || ID_AS_STRING;
END ^

```

Validation and defaults

Triggers can improve on standard SQL constraints when it comes to the issues validating input and applying default values.

Validation

SQL provides for CHECK constraints to ensure that only “good” data are stored. For example, columns created under this domain are restricted to upper case characters and digits:

```

CREATE DOMAIN TYPECODE CHAR(3)
CHECK (VALUE IS NULL OR VALUE = UPPER(VALUE));

```

This is fine: we want this rule to be enforced. On its own, the constraint will throw an exception if any client application tries to submit lower-case characters. With a trigger, we can avoid the exception altogether by fixing any attempted violations in situ:

```

CREATE TRIGGER BA_ATALE FOR ATABLE
    ACTIVE BEFORE INSERT OR UPDATE
AS

```

```
BEGIN
    NEW.ATYPECODE = UPPER(NEW.ATYPECODE);
END ^
```

Default values

In domains and column definitions, you can specify a DEFAULT value. While it seems a good idea to be able to default a non-nullable column to some zero-impact value, the SQL DEFAULT attribute is a toothless beast. It works if and only if two conditions are met:

- the operation is an INSERT
- the column has not been included in the statement's input list

Since many modern application interfaces automatically compose an insert statement using the output columns of a SELECT statement as the basis of the input list, it follows that “something” is always passed for each column. If the application itself does not apply a default, the usual behavior is for the application to pass NULL. When the server explicitly receives NULL for a defaulted column, it stores NULL. Other column constraints may kick in and cause an exception to be thrown—especially a NOT NULL constraint—but a column default never overrides or corrects any value passed from the client interface.

The second problem, of course, is that column defaults are never applied when the operation is an update.

In short, triggers do a far more effective job at managing defaults than do default column attributes. Take, for example, a column defined under this domain:

```
CREATE DOMAIN MONEY NUMERIC(18,0)
NOT NULL DEFAULT 0.00;
```

A Before Insert or Update trigger on any column using the MONEY domain will take care of the default, no matter what comes through:

```
CREATE TRIGGER BI_ACCOUNT FOR ACCOUNT
    ACTIVE BEFORE INSERT OR UPDATE
AS
BEGIN
    IF (NEW.BALANCE IS NULL) THEN
        NEW.BALANCE = 0.00;
    END ^
```



You can take care of all of the defaults for a table in a single trigger module in a single BEFORE UPDATE OR INSERT trigger.¹

“Auto-stamping”

Triggers are useful for “auto-stamping” contextual information into columns defined for the purpose. Firebird provides a number of context variables that you can use for this sort of operation; you can also provide “flags” of your own that you calculate or simply supply as static constants during the course of the trigger’s execution.

In this example, we use an “after” multi-event trigger to auto-stamp the user name, a timestamp and the transaction ID onto a log file, along with some information about the data event. Since process logging (if we do it) is likely to be the last thing we want to do in a DML event, the trigger has a high sequence number:

1. In v.1.0.x, you cannot use multi-action triggers and would need to write two parallel modules, one for Before Insert and one for Before Update.

```
CREATE TRIGGER AA_MEMBER FOR MEMBER
  ACTIVE AFTER INSERT OR UPDATE OR DELETE
  POSITION 99
AS
  DECLARE VARIABLE MEM_ID INTEGER;
  DECLARE VARIABLE DML_EVENT CHAR(4);
BEGIN
  IF (DELETING) THEN
  BEGIN
    MEM_ID = OLD.MEMBER_ID;
    DML_EVENT = 'DEL ';
  END
  ELSE
  BEGIN
    MEM_ID = NEW.MEMBER_ID;
    IF (UPDATING) THEN
      DML_EVENT = 'EDIT';
    ELSE
      DML_EVENT = 'NEW ';
    END
  END
  INSERT INTO PROCESS_LOG (
    TRANS_ID,
    USER_ID,
    MEMBER_ID,
    DML_EVENT,
    TIME_STAMP)
  VALUES (
    CURRENT_TRANSACTION,
    CURRENT_USER,
    :MEM_ID,
    :DML_EVENT,
    CURRENT_TIMESTAMP);
END ^
```

Of course, you can auto-stamp your new or edited rows directly as well, in the course of a “before” trigger.

Updating Other Tables

We just saw in the previous example how “after” triggers can perform updates on other tables to automate management tasks like logging. The capability to extend the reach of a DML event beyond the immediate context of the table and row that “own” the data has some important applications for managing difficult relationships.

Enforcing a mandatory relationship

A mandatory relationship exists when two tables are linked by a foreign key dependency and there must be at least one dependent row for each primary row. Since SQL does not

provide a “mandatoriness” constraint, trigger logic is needed to enforce the “minimum of one child” rule—not only at creation time but also when dependent rows are deleted.

Example The following example outlines one way to use triggers to enforce this mandatory “master-detail” relationship. It assumes that the primary key of the master table is known by the application before the new detail (child) row is posted.

First, we create the two tables:

```
CREATE TABLE MASTER (
    ID INTEGER NOT NULL PRIMARY KEY,
    DATA VARCHAR(10));
COMMIT;

--
CREATE TABLE DETAIL (
    ID INTEGER NOT NULL PRIMARY KEY,
    MASTER_ID INTEGER, /* The foreign key column is deliberately nullable */
    DATA VARCHAR(10),
    TEMP_FK INTEGER,
    CONSTRAINT FK_MASTER FOREIGN KEY(MASTER_ID)
    REFERENCES MASTER
    ON DELETE CASCADE);
COMMIT;
```

When the application posts the rows for the master and detail tables, it will pass the detail rows first, with NULL in the foreign key column and the primary key value of the master in the column TEMP_FK.

Next, we need an exception that can be raised if an attempt is made to violate the mandatory rule and delete the last detail row. We also create a generator for the detail row.

```
CREATE EXCEPTION CANNOT_DEL_DETAIL
    'This is the only detail record: it can not be deleted.';

--
CREATE GENERATOR GEN_DETAIL;
COMMIT;
```

This trigger tests the detail table after the new master record version has been written. It can “see” the details rows previously posted in the same transaction that have its primary key value (NEW.ID) in the TEMP_FK column. In the case of an update, rather than an insert, it can also identify any rows that it already owns.

- Any rows that meet the TEMP_FK condition get their foreign key filled and TEMP_FK is set null.
- If it finds no rows meeting these conditions, it inserts an “empty” detail row itself.

```
SET TERM ^;
CREATE TRIGGER AI_MASTER FOR MASTER
    ACTIVE AFTER INSERT OR UPDATE POSITION 1
AS
BEGIN
    IF (NOT (EXISTS (
        SELECT 1 FROM DETAIL WHERE MASTER_ID = NEW.ID
        OR TEMP_FK = NEW.ID))) THEN
        INSERT INTO DETAIL (MASTER_ID)
```

```

VALUES (NEW.ID);
ELSE
  IF (NOT (EXISTS (
    SELECT 1 FROM DETAIL WHERE MASTER_ID = NEW.ID))) THEN
    UPDATE DETAIL SET
      MASTER_ID = NEW.ID,
      TEMP_FK = NULL
    WHERE TEMP_FK = NEW.ID;
END ^

```

The detail table gets an automatically-generated key:

```

CREATE TRIGGER BI_DETAIL FOR DETAIL
  ACTIVE BEFORE INSERT AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_ID(GEN_DETAIL, 1);
  END ^

```

This Before Delete trigger for the detail table will ensure that the row can not be deleted if it is the only one.

```

CREATE TRIGGER BD_DETAIL FOR DETAIL
  ACTIVE BEFORE DELETE POSITION 0
AS
BEGIN
  IF (NOT (EXISTS (
    SELECT 1 FROM DETAIL
      WHERE MASTER_ID = OLD.MASTER_ID
      AND ID <> OLD.ID)))
  THEN
    EXCEPTION CANNOT_DEL_DETAIL;
  END ^

```

Currently, we have the situation where the mandatory relationship is protected so well that, if an attempt is made to delete the master row, this trigger will cause the cascading delete to fail. We need two more triggers for the master, extending the automatic triggers created by the system for the cascading delete. In the master's Before Delete, we “null out” the last detail row's foreign key and stamp the TEMP_FK column. After the master row's deletion has been written, we go back and delete the detail row.

```

CREATE TRIGGER BD_MASTER FOR MASTER
  ACTIVE BEFORE DELETE
AS
BEGIN
  UPDATE DETAIL SET
    MASTER_ID = NULL,
    TEMP_FK = OLD.ID
  WHERE MASTER_ID = OLD.ID;
END ^
/* */
CREATE TRIGGER AD_MASTER FOR MASTER
  ACTIVE AFTER DELETE AS

```

```
BEGIN
  DELETE FROM DETAIL
  WHERE TEMP_FK = OLD.ID;
END ^
COMMIT ^
SET TERM ;^
```

Let it be stressed that this example is unlikely to fit every requirement for mandatory relationships. There are usually several other factors to consider, in terms of both the business rules requirements and the programming interface. It is rare for trigger logic not to rise to the occasion.

Referential Integrity Support

Formal—or declarative—referential integrity constraints should be used wherever it is practicable to do so. The checking that occurs when formal referential integrity constraints are in force is all done with triggers, internally. If you want to extend the activity of the RI actions you have defined for a relationship, triggers are the way to do it.

The internal triggers in each phase fire after the custom ones you define yourself. Take care not to write triggers that conflict with what the declarative triggers have been assigned to do. If you find that your desired custom trigger action conflicts with the internal actions, a rethink will be needed of both your declarative RI definition and your custom trigger action.

Implementing RI without constraints

Some diehards who have been developing with Firebird and its InterBase® cousins for years eschew declarative RI with a passion and use triggers to “roll their own”. There is no technical reason, with any Firebird version, to avoid declarative RI if you need RI—it works very well and it doesn’t eat much.

However, declarative RI requires a foreign key which, in turn, requires a mandatory index. . Firebird does not yet² have a way to enforce foreign keys without the mandatory index. There is a situation, common enough to warrant special attention, where the index on a foreign key may be so bad for the performance of queries involving the tables concerned that a formal referential relationship must be avoided. The phenomenon occurs when the design incorporates tables of the sort known as the “lookup” or “system” or “control” table.

The lookup table

A lookup or control table is typically a static table with a small row count, which may be used in a similar way in several different contexts. It consists of a small primary key and a description field, and maybe a calculation factor or some rule that processes need to refer to. Examples are tax tables, account types, transaction types, reason codes, and so on. It has been distilled out of the normalization process as a system table that is linked to by other tables, often many different ones, by storing the lookup key in the user table. Because one row in a lookup table supplies information to many rows, a slavish adherence to relational analysis rules often results in foreign keys being bestowed on the lookup key columns of the user tables.

It is a perfectly valid and standard way to use relations—what would we do without it? However, it tends to distribute a small range of possible lookup key values across large,

2. ...as at v.2.5.1.

dynamic user tables. Such large tables often carry a number of these lookup keys as foreign keys and with them, a number of very unhealthy automatic indexes that can not be dropped.

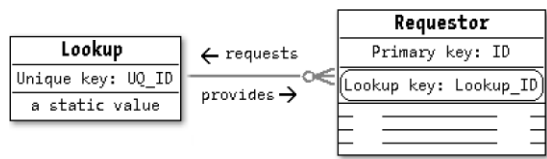
The phenomenon of few values in a large index can give rise to indexes that become less and less selective as the table grows. Because of the nature of indexing in Firebird, these lookup indexes can kill query performance over time.

The indexes that support foreign keys are mandatory and can be dropped only by dropping the constraint. Yet, by dropping the constraint, you lose the protection of the automatic referential integrity triggers. The way out of this dilemma is to write your own referential integrity triggers.

Special relationship—custom RI handling

This topic addresses a particular kind of relationship—system lookups—that is not usefully supported by declarative RI. The terms used here reflect the requirements of this case, since a fully-customized RI setup is pointless for regular master-detail relationships. Figure 30.1 illustrates the situation. A requestor—which can be any table—has a lookup key that points to a single, uniquely-keyed row in a lookup table. The value in the row is provided by the lookup table on request, often via a subquery.

Figure 30.1 Lookup-Requestor relationship



To preserve referential integrity, we want triggers that will provide an eclectic set of safeguards for the users of the lookup table—the requestors—just as declarative RI provides safeguards to protect master-detail dependencies:

- 1 The lookup row must not be deleted if a requestor is using it. For this, we need a Before Delete trigger on Lookup to check this and, if necessary, raise an exception and stop the action.
- 2 We should make and enforce a rule that requires Requestor's lookup key to be one that matches a key in Lookup. Our rule may or may not permit the lookup key to be null.
- 3 We may want to make a rule that the static value should never be changed. In a tax table, for example, the same (external) tax code may be associated with different rates and formulae from year to year. Perhaps the Chief Accountant may be permitted to change a lookup row.
- 4 A Before Update trigger on the Requestor would be required to handle a complex rule such as the one described in (3) to check dates and possibly other criteria, in order to enforce the rule and pick the correct key.

Implementing the custom RI

Suppose we have these two tables:

```
CREATE TABLE LOOKUP (  
  UQ_ID SMALLINT NOT NULL UNIQUE,  
  VALUE1 VARCHAR(30) NOT NULL,  
  VALUE2 CHAR(2) NOT NULL,
```



```

    START_DATE DATE,
    END_DATE DATE);
COMMIT;
--
CREATE TABLE REQUESTOR (
    ID INTEGER NOT NULL PRIMARY KEY,
    LOOKUP_ID SMALLINT,
    DATA VARCHAR(20)
    TRANSAC_DATE TIMESTAMP NOT NULL);
COMMIT;

```

We'll proceed to set up the existence rules for the two tables. We plan to use exceptions to stop DML events that would violate integrity, so we create them first:

```

CREATE EXCEPTION NO_DELETE
    'Can not delete row required by another table';
CREATE EXCEPTION NOT_VALID_LOOKUP
    'Not a valid lookup key';
CREATE EXCEPTION NO_AUTHORITY
    'You are not authorized to change this data';
COMMIT;

```

The first trigger does the existence check when an attempt is made to delete a lookup row:

```

SET TERM ^;
CREATE TRIGGER BD_LOOKUP FOR LOOKUP
    ACTIVE BEFORE DELETE
AS
BEGIN
    IF (EXISTS(
        SELECT LOOKUP_ID FROM REQUESTOR
        WHERE LOOKUP_ID = OLD.UQ_ID)) THEN
        EXCEPTION NO_DELETE;
    END ^

```

This one is the other side of the existence enforcement: a lookup key can not be assigned if it doesn't exist in the lookup table:

```

CREATE TRIGGER BA_REQUESTOR FOR REQUESTOR
    ACTIVE BEFORE INSERT OR UPDATE
AS
BEGIN
    IF (NEW.LOOKUP_ID IS NOT NULL
    AND NOT EXISTS (
        SELECT UQ_ID FROM LOOKUP
        WHERE UQ_ID = NEW.LOOKUP_ID)) THEN
        EXCEPTION NOT_VALID_LOOKUP;
    END ^

```

We might now add further triggers to enforce other integrity rules we need. For example, this trigger will restrict any update or delete of the Lookup table to a specific user:

```

CREATE TRIGGER BA_LOOKUP FOR LOOKUP
    ACTIVE BEFORE UPDATE OR DELETE

```

```

AS
BEGIN
    IF (CURRENT_USER <> 'CHIEFACCT') THEN
        EXCEPTION NO_AUTHORITY;
    END ^

```

This one will check the input lookup code to make sure that it is the right one for the period of the transaction and correct it if necessary:

```

CREATE TRIGGER BA_REQUESTOR1 FOR REQUESTOR
    ACTIVE BEFORE INSERT OR UPDATE POSITION 1
AS
    DECLARE VARIABLE LOOKUP_NUM SMALLINT;
    DECLARE VARIABLE NEED_CHECK SMALLINT = 0;
BEGIN
    IF (INSERTING AND NEW.LOOKUP_ID IS NOT NULL) THEN
        NEED_CHECK = 1;
    IF (UPDATING) THEN
        IF (
            (OLD.LOOKUP_ID IS NULL
             AND NEW.LOOKUP_ID IS NOT NULL)
            OR (OLD.LOOKUP_ID IS NOT NULL
                AND NEW.LOOKUP_ID <> OLD.LOOKUP_ID)) THEN
            NEED_CHECK = 1;
        --
    IF (NEED_CHECK = 1) THEN
        BEGIN
            SELECT L1.UQ_ID FROM LOOKUP L1
            WHERE L1.START_DATE <= CAST(NEW.TRANSAC_DATE AS DATE)
            AND L1.END_DATE >= CAST(NEW.TRANSAC_DATE AS DATE)
            AND L1.VALUE2 = (SELECT L2.VALUE2 FROM LOOKUP L2
                            WHERE L2.UQ_ID = NEW.LOOKUP_ID)
            INTO :LOOKUP_NUM;
            NEW.LOOKUP_ID = LOOKUP_NUM;
        END
    END ^
COMMIT ^
SET TERM ;^

```

Updating rows in the same table

Before considering using a trigger to update other rows in the same table, look carefully at the effect of triggering off a cycle of nested activity. If a trigger performs an action that causes it to fire again—or fires another trigger that performs an action that causes it to fire—an infinite loop results. For this reason, it is important to ensure that a trigger's actions never cause the trigger to fire, even indirectly.

If you arrive at a point in your database design at which it is necessary to write a trigger to implement a data dependency *between rows in the same table*, it is likely to be a sign of poor normalization unless the dependency pertains to a tree structure (see below). If a segment

of the row structure affects, or is affected by, a change of state in another row, that segment should be normalized out to a separate table, with foreign keys to enforce the dependency rule.

Self-referencing tables and trees

Self-referencing tables that implement tree structures are a special case. Each row in such a table is a node in a tree and inter-row dependencies are inherent. Any node potentially has two “lives”: one as a parent to nodes beneath it, the other as a child to a higher node.

Triggers are likely to be required for all DML events, both to modify the behavior of referential integrity constraints and to maintain the metatables (graphs) used by some tree algorithms to make the state of the tree’s geometry available to queries.



Triggers for trees should always be designed with conditions and branches that protect the structure from infinite loops.

Updating the same row

Never try to use an SQL statement to update or delete the same row that the trigger is operating on. The following, for example, is not advisable:

```
CREATE TRIGGER O_SO_SILLY FOR ATABLE
ACTIVE BEFORE UPDATE
AS
BEGIN
    UPDATE ATABLE SET ACOLUMN = NEW.ACOLUMN
    WHERE ID = NEW.ID;
END ^
```

Always use assignment to the NEW variables for same-row modifications and never NEVER resolve an exception by attempting to delete the row from within the trigger!

Implementing Auto-Incrementing Keys

A recommended usage of BEFORE INSERT triggers in Firebird is for implementing @IDENTITY-style auto-incrementing primary keys. The technique is simple and most Firebird developers learn to write these triggers in their sleep!

It involves two steps:

- 1 Create a generator (sequence) to be used to generate the unique numbers for the key
- 2 Write a BEFORE INSERT trigger for the table that pulls the next value off the named generator and applies it to the primary key or other unique column

To illustrate the technique, we’ll implement an auto-incrementing primary key for a table named CUSTOMER that has the primary key CUSTOMER_ID, a BIGINT type.



If the use case needs an INTEGER or a SMALLINT, the technique will work fine. Just be careful to get it right—otherwise, one day, the generator will start generating numbers that are too big for your column!

For a dialect 1 database, which does not support BIGINT, make sure CUSTOMER_ID is an integer.

First, create the generator:

```
CREATE GENERATOR GEN_PK_CUSTOMER;
```

or

```
CREATE SEQUENCE SEQ_PK_CUSTOMER;
```

Now, the trigger:

```
CREATE TRIGGER BI_CUSTOMER FOR CUSTOMER
  ACTIVE BEFORE INSERT
  POSITION 0
AS
BEGIN
-- *****
-- Alternative 1: using GENERATOR syntax
  IF (NEW.CUSTOMER_ID IS NULL) THEN
    NEW.CUSTOMER_ID = GEN_ID(GEN_PK_CUSTOMER, 1);
-- *****
-- Alternative 2: using SEQUENCE syntax (v.2.0+)
  IF (NEW.CUSTOMER_ID IS NULL) THEN
    NEW.CUSTOMER_ID = NEXT VALUE FOR SEQ_PK_CUSTOMER;
-- *****
END ^
COMMIT ^
```



Don't use both of the suggested blocks. The "2" series introduced the standards-compliant SEQUENCE syntax for compatibility with databases being migrated from another data management system. Unlike with many situations where alternative syntaxes co-exist, you can mix and match, treating a generator as a sequence and vice versa. It makes it useful for use cases where you want the sequence to "step" by more than one number. NEXT VALUE FOR does not support that; GEN_ID() does.

When an insert is performed, CUSTOMER_ID is deliberately omitted from the input list of the INSERT statement:

```
INSERT INTO CUSTOMER (
  LAST_NAME,
  FIRST_NAME,
  ...)
VALUES (?, ?, ...);
```

Without the trigger, this statement would cause an exception because primary keys are not nullable. However, the BEFORE INSERT trigger executes before the constraint validation, detects that CUSTOMER_ID is null and does its stuff.

Why NEW.value is tested for null

If the trigger can do that for me, you might ask, why should it need to test for null?

It can be of benefit for an application to know what the primary key of a new row will be without having to wait until its transaction commits. For example, it is a very common requirement to create a "master" record and link new "detail" records to it, usually by way of a foreign key, in a single transaction. It is clumsy—even risky sometimes—to break the atomicity of the master-detail creation task by committing the master in order to acquire the foreign key value for the detail records, as one needs to do when relying solely on the trigger.

Applications written for Firebird take advantage of one peculiar characteristic of generators: they are not subject to any user transaction. Once generated, a value can not be taken by another transaction and it can not be rolled back.³

A quick query from the application returns the value:

```
SELECT GEN_ID(GEN_PK_CUSTOMER, 1) AS RESULT FROM RDB$DATABASE;
```

If our trigger omitted the null test and just did this:

Wrong code:

```
...
AS
    NEW.CUSTOMER_ID = GEN_ID (GEN_PK_CUSTOMER, 1);
END ^
```

then the value posted by the application would get overwritten by the second “pull” off the generator and break the link to the detail records.

This situation is not an argument for dispensing with the triggered key. On the contrary, the trigger with the null test ensures that the business rule will be enforced under any conditions.



In an environment where there is not good integration of the application work of different developers, or where users are allowed free access to the database using query tools, it will be necessary for your triggers to include more protection for the integrity of keys, such as range-checking or another appropriate form of validation.

Changing Triggers

Three forms are available for modifying triggers:

- **ALTER TRIGGER**, which changes the definition of an existing trigger module while preserving its dependencies on other objects. It can be used with minimal disturbance to deactivate a trigger.
- **CREATE OR ALTER TRIGGER**⁴ creates the trigger module if it doesn’t exist and works exactly as **CREATE TRIGGER** does; otherwise **ALTER** rules apply and dependencies are preserved.
- **RECREATE TRIGGER**⁵ creates the trigger module if it doesn’t exist and works exactly as **CREATE TRIGGER** does; otherwise tries to drop the trigger and create a new one. Dependencies are not preserved.

If any of these operations would break a dependency, it will fail with an exception.

Syntax The syntax pattern is:

```
{ALTER TRIGGER name} | {CREATE OR ALTER TRIGGER name FOR {table | view}
[ACTIVE | INACTIVE]
[{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
[POSITION number]
AS <trigger_body>;
```

3. As at the time of this writing, there is evidence that, when the embedded model is being used on Windows platforms, the generator increments will be lost if the embedded application crashes without a proper shut-down of the session. This is not a roll-back situation: it reflects a known problem on Windows, which retains changed blocks in its disk cache until files are closed.
4. Not in Firebird 1.0.x.
5. Not available prior to the “2” series

ALTER TRIGGER

The FOR name clause that is used in CREATE OR ALTER TRIGGER is omitted. ALTER TRIGGER can not be used to change the table with which the trigger is associated.

Changing only the header

When you use it to change only a trigger header, ALTER TRIGGER requires at least one altered attribute after the trigger name. Any header attribute omitted from the statement remains unchanged.

The following statement deactivates the trigger SAVE_SALARY_CHANGE:

```
ALTER TRIGGER SAVE_SALARY_CHANGE INACTIVE;
```

If the phase indicator (BEFORE or AFTER) is altered, then the event (UPDATE, INSERT, or DELETE) must also be specified. For example, the following statement reactivates the trigger, VERIFY_FUNDS, and specifies that it fire before an update instead of after:

```
ALTER TRIGGER SAVE_SALARY_CHANGE
ACTIVE BEFORE UPDATE;
```

Changing the body

Any change whatsoever to the trigger body causes the new body definition to replace the old definition. ALTER TRIGGER need not contain any header information other than the trigger's name—recall that header elements are not changed if they are not specified.

For example, the following statement modifies trigger, SET_CUST_NO, that was created with this definition:

```
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
ACTIVE BEFORE INSERT
AS
BEGIN
    IF (NEW.CUST_NO IS NULL) THEN
        NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
    END^
```

We will alter the trigger to have it insert a row into a new table, NEW_CUSTOMERS, each time a new row is inserted into the CUSTOMER table:

```
SET TERM ^;
ALTER TRIGGER SET_CUST_NO
BEFORE INSERT AS
BEGIN
    IF (NEW.CUST_NO IS NULL) THEN
        NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
        INSERT INTO NEW_CUSTOMERS(NEW.CUST_NO, CURRENT_DATE)
    END ^
SET TERM ;^
```

CREATE OR ALTER TRIGGER

This tolerant syntax creates a new trigger if one with the supplied name is not found, or alters an existing procedure of that name.

Simply edit the original CREATE definition as required, inserting the keywords OR ALTER.

“Object is in use” error

As with stored procedures, committing the change will throw the notorious error if any user is currently using the trigger or another object that depends on it.

In any event, the new version of the trigger won't be immediately available on Superserver if the old version is still in the cache. All users must log out and, when they log in again, they will see the new version.

On Classic or Superclassic, the new version will be available to the next client that logs in but not to clients that are already connected.

Inactive/active

Performing ALTER TRIGGER...INACTIVE | ACTIVE does not usually invoke the “Object in use” error unless an existing transaction has a table lock. The deactivation (or reactivation) will not affect transactions that already have the table in their purview. However, it should affect the next transaction that requests a state change on the table.



If your requirements are such that you have triggers needing to be switched off and on regularly, it could be good practice to include some conditioning in those triggers to meet the requirements, rather than exposing the integrity of your data in an arbitrary fashion.

RECREATE TRIGGER

v.2.0 +

With this form, the engine will try to drop the trigger if one exists with the same name and create a new one with the modified code. RECREATE TRIGGER will fail if the existing trigger is in use. Its syntax is identical to CREATE TRIGGER (see [Creating Table-level Triggers](#)) except for the initial verb.

Higher-level Triggers

v.2.1 +

The so-called “database triggers” can be defined to fire at the session or the transaction level.

Writing the higher-level triggers is very much like writing statement level triggers. Of course, statement-level context variables such as the NEW and OLD column variables are not available. The syntax pattern for the trigger headers is different, reflecting the different phasing of the database- and transaction-level events.

Trigger Events

Unlike statement-level triggers, database triggers do not split an event into BEFORE and AFTER phases. However, as with statement triggers, you can define multiple triggers for an event and use a POSITION clause to assign an execution order for them.

Session-level triggers

A session-level trigger's effects last for the duration of the client's session with a database. Triggers can be written for one or both of two events, viz.,

- **ON CONNECT**—triggers fire in an initial transaction started after a database connection is established. If an exception occurs that is not handled by the triggers, the initial transaction is rolled back, the attachment is disconnected and the exception is returned to the client application. As with other triggers, an exception handler can also raise a custom exception and pass it to the end of the trigger, with the same effect, except that you have the opportunity to make the experience less traumatic for the user!

After all **CONNECT** triggers have executed without unhandled exceptions, the initial transaction is committed and the connection is "live".

- **ON DISCONNECT**—When a detach request is received, a transaction is started for the **DISCONNECT** event. Triggers are fired and, if there are uncaught exceptions, the transaction is rolled back, the detach request is executed and the exceptions are swallowed.

If there are no exceptions, the transaction is committed and the detach request is executed.

Transaction-level triggers

A transaction-level trigger's effects last for the duration of a transaction and apply to every user transaction started in the database. Triggers can be written for one or more of three events, viz.,

- **ON TRANSACTION START**—Triggers are fired in the newly-created user transaction. Uncaught exceptions are returned to the client and the transaction is rolled back.
- **ON TRANSACTION COMMIT**—The timing of the **TRANSACTION COMMIT** event depends on whether the transaction is single-phase or two-phase:
 - for a single-phase transaction, immediately the transaction is about to commit.
 - for a two-phase transaction, in the Prepare (pre-commit) phase

Uncaught exceptions roll back the trigger's savepoints and the **COMMIT** request is aborted. The exception is returned to the client.

- **ON TRANSACTION ROLLBACK**—triggers are fired immediately preceding the roll-back of the transaction. Changes done will be rolled back with the transaction. Exceptions are swallowed.

A possible "gotcha"

Since the **ON DISCONNECT** and **ON TRANSACTION ROLLBACK** events have no direct way to inform the client of an exception, clearly an unhandled exception in either of these events results in a lock-out. In some circumstances, that may be exactly what was intended.

In case this Catch-22 situation occurs unexpectedly, there is something you can do to enable you to get in and fix the problem. Some of the Firebird command-line utilities have switches that the database owner or a user with **SYSDBA** privileges can apply to suppress the firing of all the high-level triggers for the duration of a session with that tool:

```
gbak -nodbtriggers
isql -nodbtriggers
nbackup -T
```


Creating and Changing the Higher-level Triggers

Only the database owner or a user with SYSDBA privileges may create, recreate, create or alter, or drop database triggers. Make sure you have exclusive access to the database when performing these DDL operations.

The syntax pattern for creating and changing the higher-level triggers is:

```
{CREATE | RECREATE | CREATE OR ALTER}
TRIGGER <trigger-name>
[ACTIVE | INACTIVE]
ON {CONNECT | TRANSACTION START | TRANSACTION COMMIT |
    TRANSACTION ROLLBACK | DISCONNECT }
[POSITION <n>]
AS
BEGIN
...
END
```

To understand the differences between the styles for modifying the definitions of the triggers, refer to [Changing Triggers](#), earlier in this chapter.



A higher-level trigger's event type cannot be altered. If you need to make a trigger execute in a different event, you must drop it and create a new one redefining the event type.

Dropping Triggers

During database design and application development, a trigger may cease to be useful. To remove a table-level or higher-level trigger permanently, log in as the owner or a user with SYSDBA privileges and use DROP TRIGGER.

The syntax pattern is

```
DROP TRIGGER trigger-name;
```

The trigger name must be the name of an existing trigger. The following example drops the trigger, SET_CUST_NO:

```
DROP TRIGGER SET_CUST_NO;
```



To disable a trigger temporarily, use

```
ALTER TRIGGER trigger-name INACTIVE.
```


ACCESSING OTHER DATABASES FROM PSQL

Moving forward from the “simple” syntax for EXECUTE STATEMENT discussed in Chapter 28, in this chapter we explore the extensions that arrived with Firebird 2.5. In various combinations they can be used connect the DSQL statement to an external database for execution there.

Note that we are not talking about the two-phase, multi-database client transaction that all versions of Firebird support. You can find out about those in the chapters about transactions in Part Five. EXECUTE STATEMENT does not work in a cross-database (two-phase) transaction.

We are not talking, either, about cross-database schemas or namespaces—at the time of this writing, those are still to come, in a future version.

What the extensions provide is the capability for an executing PSQL module to request an autonomous transaction in a database other than the one in which the module is executing. That other database can be on the same host as the CURRENT_CONNECTION database or on a remote host.

Extensions to EXECUTE STATEMENT

v.2.5 +

When using a simple EXECUTE STATEMENT in a PSQL module, the statement supplied to its string argument runs by default inside the CURRENT_TRANSACTION, as do all statements that execute in the module.

Autonomous Transaction

An autonomous transaction is one that executes work independently of the CURRENT_TRANSACTION, during the course of execution of a stored procedure or trigger. For work in the current database it would usually be a block wrapped as an IN

AUTONOMOUS TRANSACTION construct. For work in an external database, the work is executed with EXECUTE STATEMENT having the WITH AUTONOMOUS TRANSACTION and ON EXTERNAL DATA SOURCE attributes . The EXECUTE STATEMENT query content executes in a separate transaction that will be committed automatically if the statement completes without errors. If the statement’s execution is stopped by an exception, the autonomous transaction is rolled back, returning a generalised “eds_” exception from the external data source provider mechanism.



EXECUTE STATEMENT WITH AUTONOMOUS TRANSACTION is not restricted to operations on external databases. It can be used for autonomous operations in the CURRENT_CONNECTION.

See also The IN AUTONOMOUS TRANSACTION DO.. construct in Chapter 28.

The Optional Extension Clauses

Syntax The pattern for the expanded syntax is:

```
...
[FOR] EXECUTE STATEMENT <query-text> [( <input-parameters> )]
-----
-- O P T I O N A L   C L A U S E S --
[WITH {COMMON | AUTONOMOUS} TRANSACTION]
[ON EXTERNAL [DATA SOURCE] <connection-string>]
[AS USER <user-name>] [PASSWORD <password>] [ROLE <role-name>]
[WITH CALLER PRIVILEGES]
-----
[INTO <variables>]
[DO <psql-statement>]
...
```

The order of the optional clauses is not fixed but clauses cannot be duplicated.



Don't forget that <query-text>, <user-name>, <password>, <role> and <connection-string> are string expressions. If they are supplied as literals, they must be enclosed in single-quote characters.

Clauses and arguments

<query-text>—This is the string or string expression that carries the query. It can include input parameters: if it does, the string or variable must be in parentheses. Additionally, if it is a literal string, rather than a variable, then the whole expression (within the parentheses) must be enclosed in single quotes.

<input-parameters>—If the **<query-text>** specifies input parameters, the values assigned to them are listed in this phrase. Parameterisation has options: the scheme and usage are described in the topic Using parameters with EXECUTE STATEMENT in Chapter 28, **Overview of PSQL**.

WITH {COMMON | AUTONOMOUS} TRANSACTION—Specifies whether to execute **<query-text>** in the CURRENT_TRANSACTION (the default, **COMMON**, which need not be specified) or in a separate transaction.

ON EXTERNAL [DATA SOURCE] <connection-string>—Specifies the path to a separate database. **WITH AUTONOMOUS TRANSACTION** is required with this, along with adequate log-in credentials.

If no `ON EXTERNAL DATA SOURCE` clause is present, `EXECUTE STATEMENT` is normally executed within the `CURRENT_CONNECTION` context. This will be the case if the `AS USER` clause is omitted, or it is present with its `<user_name>` argument equal to `CURRENT_USER`.

See *The <connection-string> argument* below, for more details.

AS USER <user-name> PASSWORD <password>—Required if the external data source is specified, unless acceptable log-in credentials are established some other way. Refer to *Authentication*, below. Both arguments require single quotes if supplied as literals.

If `<user-name>` is equal to `CURRENT_USER`, the statement is executed in the `CURRENT_TRANSACTION`. If not, it is executed in the same engine instance using a separate connection, established without Y-Valve and remote layers.

The case of the supplied `<user-name>` must match the name that is stored in `security2.fdb` of the host being connected to, i.e., all upper case if it is not stored with a quoted identifier and case-correct if it is.

In the absence of an **AS USER <user_name>** clause, `CURRENT_USER` is the default.

ROLE <role-name>—For use in case the login credentials require it for the named user to obtain the privileges necessary to execute the statement. The case of the supplied `<role-name>` must match the name that is stored in the database being connected to, i.e., all upper case if it is not stored with a quoted identifier and case-correct if it is. The argument requires single quotes if it is supplied as a literal.

WITH CALLER PRIVILEGES—Makes it possible to have the executable statement inherit the access privileges of the calling stored procedure or trigger. The statement is prepared using any *additional privileges* that apply to the calling stored procedure or trigger.

The effect is the same as if the statement were executed by the stored procedure or trigger directly.

This option is not compatible with the `ON EXTERNAL DATA SOURCE` option.

Transaction Behaviour

The optional clauses available in the extended syntax for setting the appropriate transaction behaviour are `WITH AUTONOMOUS TRANSACTION` and `WITH COMMON TRANSACTION`.

WITH COMMON TRANSACTION

`WITH COMMON TRANSACTION` is the default and simply conveys the “normal” situation as it is in prior versions, whereby the transaction lifetime of `EXECUTE STATEMENT` is bound to the lifetime of the `CURRENT_TRANSACTION`. It does not need to be specified: it is what you get if you omit `WITH AUTONOMOUS TRANSACTION`.

The behaviour for `WITH COMMON TRANSACTION` is as follows:

- Any transaction in an external data source will be a fresh one, started with the same parameters as `CURRENT_TRANSACTION`
- Otherwise, one of the following, in order of preference:
 - the statement is executed inside the `CURRENT_TRANSACTION`, if possible
 - an available transaction in the pool will be reused

- another transaction may be started for it internally within the `CURRENT_CONNECTION`.

External connections made `WITH COMMON TRANSACTION` stay open until the end of `CURRENT_TRANSACTION` and will be committed or rolled back in accord with the fate of `CURRENT_TRANSACTION`. If (and only if) the **<connection-string>** is identical both in case and content, these transactions can be reused by subsequent calls to `EXECUTE STATEMENT`.

WITH AUTONOMOUS TRANSACTION

`WITH AUTONOMOUS TRANSACTION` starts a new transaction with the same parameters as `CURRENT_TRANSACTION`. That transaction will be committed if the statement is executed without exceptions or rolled back if the statement encounters an error, immediately after the statement completes execution or exits due to exception.

Regardless of the outcome of the autonomous transaction, any pending work in the `CURRENT_TRANSACTION` is unaffected.



WITH AUTONOMOUS TRANSACTION statements will “piggy-back” on connections that were opened earlier by statements *WITH COMMON TRANSACTION* if a perfect match is found for the **<connection-string>**. In that event, the reused connection will be left open after the statement has been executed. (It must be, because it has at least one uncommitted transaction!)

External Queries

The `ON EXTERNAL DATA SOURCE` clause with its **<connection-string>** argument specifies that the statement is to be executed in another database.

The <connection-string> argument

The format of **<connection_string>** is the usual one that is passed through the API function `isc_attach_database()`, viz.

[<host_name><protocol-delimiter>] <database-path>

Examples Connecting to a database on a different host, POSIX:

```
myserver:/opt/databases/mydatabase.fdb
```

Or, if a database alias, e.g., `myalias`, is defined:

```
myserver:myalias
```

Connecting to a database on a different host, Windows:

```
winserver:C:\databases\mydatabase.fdb -- TCP/IP
```

```
\\winserver\C:databases\mydatabase.fdb -- WNET
```

```
winserver:myalias
```

```
\\winserver\myalias
```

Database path or alias alone may be passed if the external database is located on the same host machine and the server supports a “serverless” local connection.



Use of a two-phase transaction for the external connection is not available in V.2.5 at the time of this writing.

Character Set

The connection to the external data source uses the same character set as is being used by the `CURRENT_CONNECTION` context.

Authentication

Where server authentication is needed for a connection that is different to `CURRENT_CONNECTION`, e.g., for executing an `EXECUTE STATEMENT` command on an external datasource, the `AS USER` and `PASSWORD` clauses are required. However, under some conditions, the `PASSWORD` may be omitted and the effects will be as follows:

- On Windows, for the `CURRENT_CONNECTION` (i.e., no external data source), trusted authentication will be performed if it is active and the `AS USER` parameter is missing, null or equal to `CURRENT_USER`.
- If the external data source parameter is present and its `<connection-string>` refers to the same database as the `CURRENT_CONNECTION`, the effective user account will be that of the `CURRENT_USER`.
- If the external data source parameter is present and its `<connection_string>` refers to a different database than the one `CURRENT_CONNECTION` is attached to, the effective user account will be the operating system account under which the Firebird process is currently running.

In any other case where the `PASSWORD` clause is missing, only the user name (*isc_dpb_user_name*) will be presented in the attachment parameters and native authentication will be attempted.

An authentication parameter is treated as “missing” if:

- it is absent
- it is `NULL` or an empty string
- it is `USER` and its value is the same as the `CURRENT_USER`
- it is `ROLE` and its value is the same as the `CURRENT_ROLE`

Exceptions

When `ON EXTERNAL` is used, it is an “external provider” that makes the connection—even if it connects to the same database as the `CURRENT_CONNECTION`.

As at the v.2.5.1 implementation, two possible “eds_” exception symbols are returned to the `GDSCODE` context variable for external data source errors: `eds_connection` for any failure to connect and `eds_statement` for any exception that occurs when the statement fails for any reason at all.

To catch them in your handler code, provide `WHEN GDSCODE` traps in the appropriate block levels for both of these symbols (`WHEN GDSCODE eds_connection`, `WHEN GDSCODE eds_statement`) if you want the module to respond specifically, such as passing details of the failure to a log; or include a `WHEN ANY` if the procedure doesn’t need a specific response to an `ON EXTERNAL` error.

Note *If there is no `ON EXTERNAL` clause, exceptions are caught as normal, even if the channel is through an extra connection to the database that the `CURRENT_CONNECTION` refers to.*

For more information, refer to *Trapping and Handling Exceptions* in the previous chapter.

CHAPTER 32

ERROR HANDLING AND EVENTS

In the first part of this chapter we examine how the execution of PSQL modules—both triggers and procedures—can be enhanced to trap errors and handle them within the executing code.

In the second part, we explore the concept of event callbacks—messages ported to a common clearing house on committing the work of a stored procedure or trigger, that clients can listen for on demand.

Exceptions in PSQL

The standard behavior of PSQL modules when an exception occurs is to stop executing, undo all work done since the initial `BEGIN` statement, jump to the final `END` statement and return control to the client, passing one or more error messages. If the module is a trigger, the exception will undo any previously executed triggers and prevent the requested DML changes from being posted.

Types of Exceptions

Three types of exceptions can occur:

- 1 SQL errors—that is, SQL messages having a negative `SQLCODE` or, in v.2.5+, an `SQLSTATE` code of class '02' or higher
- 2 Internal Firebird errors that have to do with concurrency, data, metadata and environmental conditions. They have a nine-digit error code, made up of 335544 followed by a unique three-digit sequence that identifies the `GDSCODE`. Most `GDSCODE`s fall into generic groups beneath `SQLCODE`s and you will usually get both a `SQLCODE` and a `GDSCODE` when an exception occurs
- 3 Custom exceptions that you declare as persistent objects in the database and invoke in code when a specified condition is detected

What is an exception?

An exception is simply a message that is generated when an error occurs.

All of the predefined exceptions—SQLCODE, SQLSTATE and GDSCODE—have text messages associated with them. The default messages are in English, but they don't have to be. Versions of the messages are available in a few other languages—including Pig-Latin!—while others are either work in progress or jobs waiting for volunteers.

Firebird has DDL syntax for creating custom exceptions with text messages up to 1021 bytes long¹. You can extend your custom exceptions at run-time and add more text—including context-specific details—to the message that goes back across the wire².

Creating an exception

Creating an exception is one of the simplest pieces of DDL in the lexicon. The syntax is

```
CREATE EXCEPTION exception-name <message>;
```

exception-name is a regular Firebird identifier of 31 characters or less. It must be unique amongst identifiers for exceptions. In dialect 3, it can be double-quoted and case-sensitive, if you really want to give yourself headaches.

<message> is a single-quoted string of text in character set NONE.

Example

```
CREATE EXCEPTION NO_DOGS 'No dogs allowed!';
COMMIT;
```

A CREATE EXCEPTION statement needs to be committed, just as any other DDL statement does.

Altering or dropping an exception

With SYSDBA privileges or as the owner of an exception that is used in stored procedures, you can alter or drop it at any time. If it is used in a trigger, it can only be altered and then, only to change the text message. No dependencies are stored for exceptions used by stored procedures. This makes it a problem if you drop one and forget to replace it—it is embarrassing to have an exception occur because of a missing exception!

To drop our NO_DOGS exception:

```
DROP EXCEPTION NO_DOGS;
```

To alter it:

```
ALTER EXCEPTION NO_DOGS 'No dogs allowed except Irish Wolfhounds!';
```

Tip

When constructing schema scripts, group your CREATE EXCEPTION statements together in some position that is easy to find during development for modification and self-documentation. Developers often use short prefixes or a systematic naming scheme to accord various sorts of categories to custom exceptions.

Exceptions in Action

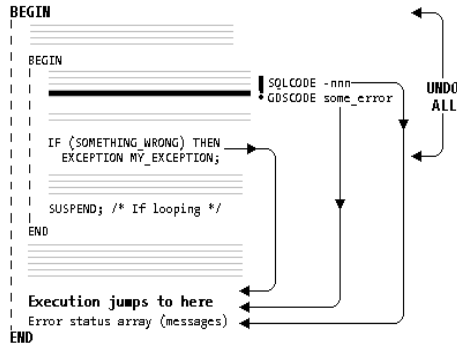
The internally defined exceptions are thrown by the engine in response to corresponding errors that require execution to stop. They cover a very large range of conditions, including every kind of constraint violation, arithmetic and string overflows, references to missing objects, data corruption and so on. The SQLCODE, SQLSTATE and GDSCODE

1. Prior to Firebird 2.0, the length is limited to 78 bytes.
2. But not in v.1.0.x.

exceptions are the same ones that are used when errors occur during dynamic SQL operations. The SQLCODE and GDSCODE errors, with their symbols and English messages, are listed in Appendix VII, **Error Codes**. The SQLSTATE classes, codes and messages are in Appendix VIII.

Custom exceptions, which are available only in PSQL modules, do not need to duplicate the work of the internally-defined ones. Define your exceptions for use when you want your code to detect error conditions that break your business rules. The three exception types are depicted in Figure 32.1.

Figure 32.1 Standard PSQL response to exceptions



We encountered an example in *A multi-table procedure* in Chapter 29, where a custom exception is used in a trigger for the purpose of stopping an event, where letting it continue would break a business rule. In this case a stored procedure was taking care of tidying up dependencies left in the organizational structure by the departure of an employee. It was declared like this:

```

CREATE EXCEPTION REASSIGN_SALES
  'Reassign the sales records before deleting this employee.' ^
COMMIT ^

```

At the point where the exception is to be used, the procedure checks whether the employee appears as the sales rep on any outstanding sales orders. If so, the custom exception is used to end the procedure. Of course, the exception, if it occurs, causes all of the other tasks performed by the procedure to be undone.

```

...
BEGIN
  IF (EXISTS(SELECT PO_NUMBER FROM SALES
    WHERE SALES_REP = :emp_num)) THEN
    EXCEPTION reassign_sales;
  ...
END

```

Note *In selectable stored procedures, output rows that have already been fetched by the client in previous loops through FOR SELECT..DO..SUSPEND are unaffected and remain available to the client. For the mechanism at work here, refer to the topic below, "The WHEN Statement".*

There are cases where it is possible to use the custom exception as a way to intervene, deal with a problem condition and let the procedure continue. We can trap the exception and write code to handle it, right there in the procedure. The next topic examines how this trap-and-fix technique can be used to deal with our reassign_sales exception.

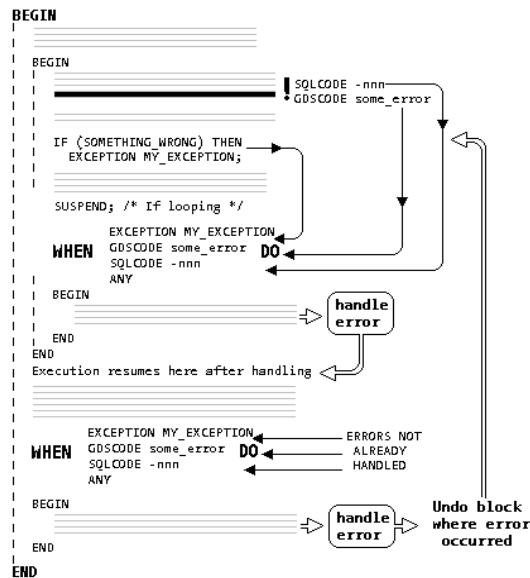
Trapping and Handling Exceptions

PSQL code can trap the errors when they occur and hand them on to exception handler routines.

If an exception is handled in your code—you provide a fix or a workaround for the error and allow execution to continue—then no exception message is returned to the client.

Figure 32.2 illustrates the logic of trapping and handling errors:

Figure 32.2 Error trapping and handling logic



As before, the exception causes execution in the block to stop. Instead of letting execution pass to the final END statement, now the procedure searches through the layers of nested blocks, starting in the block where the error was detected and backing through the outer blocks, looking for some handler code that “knows about” this exception. It is looking for the first WHEN statement that can handle the error.

The WHEN statement

A WHEN statement takes the form

WHEN <exception> DO <compound-statement>

<exception> can be any one of the following:

<exception-name> | GDSCODE code | SQLCODE code | ANY

<compound-statement> is one statement or a number of ordinary PSQL statements in a BEGIN..END block.

Scope of exception types

The paradigm of exception types shown in the syntax pattern represents a scale of scope and granularity.

The custom exception can target any condition you choose, including rules that you may not be able to express in constraints, or choose not to. WHEN statements and handler code that target custom exceptions are best placed within the same block where the error would occur.

Next in granularity is the GDSCODE. In v.1.0.s, it is a context variable of sorts, insofar as the procedure can read the code returned and compare it with the code specified in the WHEN predicate:

```
WHEN GDSCODE foreign_key DO /* could be written WHEN GDSCODE 335544466 */
BEGIN
...
END
```

GDSCODE is a full-blown context variable³. As long as you read it inside the block where the exception is raised, you can capture its number code and store it in a log record.



All GDSCODEs have symbolic constants that are more or less meaningful in English. It is highly recommended to use them for clear self-documentation. If you look in the header file `iberror.h`, in your `/include` directory, you will see that the symbolic constant definitions there have the prefix `isc_`. The prefix must be omitted in WHEN predicates.

Some errors detected by GDSCODE can be fixed inside the scope of the block where they occur. If so, the WHEN statement for handling the error can be included here; otherwise it should go into an outer loop and be handled there.

SQLCODE is quite generic and it does not always reflect an error. SQL operations pass SQLCODE 0 for successful completion and SQLCODE 100 for end-of-file. A range of unused “slots” exists between 1 and 99 for warnings and information messages. The SQLERROR range is all sub-zero numbers greater than -1000. These SQLERROR SQLCODEs tend to be high-level groupings of several GDSCODEs.

SQLSTATE—first supported in v.2.5—is not linked in a formal way to either of the other two context variables. It first became a PSQL context variable in v.2.5.1. It returns 5-character alphanumeric codes consisting of SQL CLASS (2 characters) and SQL SUBCLASS (3 characters). The “error” classes start at ‘02’.

Like GDSCODE, SQLCODE and SQLSTATE are context variables—they can be captured and logged.



With the linked GDSCODE and SQLCODE, you can capture one or the other. You will get the code for one and null for the other.

SQLCODEs are less granular than either GDSCODE or SQLSTATE. In many cases, they are the least likely to yield a condition that can be fixed inside the module. Look up the codes in Appendix VII and notice that a single SQLCODE often groups many GDSCODEs. Usually, they are most useful in the outermost block of the module.

Being implemented according to the ISO standard, SQLSTATE is still quite generic and perhaps not as useful as GDSCODE for getting to the heart of a problem.

ANY is like “an exception of last resort”. It provides a hook for any internally-defined exception that has not otherwise been handled. With the ability to read the various error code variables (provided they are in scope), ANY has better potential to provide a default

3. In all versions except v.1.0.x

handler than it does in lower versions. Of course, the code for a general default handler must be included somewhere in the outer layer of the module, to make it accessible from any of the inner layers.

Placement of WHEN blocks

Always place your WHEN blocks immediately preceding the END statement that will close the block where you want the exception handled. Don't place any other statements—not even SUSPEND or EXIT—between the end of your handlers and the closing END statement. Refer back to Figure 32.2, *Error trapping and handling logic*, which depicts this flow.



If you want to use an EXIT statement immediately before the final END statement of your module, for documentation purposes, that's fine. At that point, EXIT can not affect execution flow.

When a procedure encounters an error in the cursor loop of a selectable procedure, the statements since the last are undone back to the previous SUSPEND. Any rows already output from previous calls to SUSPEND are unaffected and remain available to the client.

SUSPEND should not be used in executable procedures. Let your execution logic determine when blocks end and when errors should be thrown back as exceptions.

Nested exceptions as savepoints

The nested architecture of PSQL module execution blocks means, of course, that PSQL supports implicitly “nested” transactions. Every PSQL module's activity is under the control of the transaction context from which it was invoked. The standard execution flow ensures that the work either completes as a whole or fails as a whole. In the case of stored procedures, an exception cause the entire invocation instance to fail; in the case of triggers, an exception causes the DML and all related events already performed to fail.

Handling exceptions provides the means to partition the execution into stages that can be undone back to specified points without necessarily discarding the entire task of the module. The level of “undoing” is determined by the point at which the error occurs and and proximity of the WHEN handler clause to the error point. The “savepoint”—equivalent to a named savepoint in a client-controlled transaction—is the start of the code block in which the WHEN handler code executes.

Handling the `reassign_sales` exception

Now, back to our `DELETE_EMPLOYEE` procedure. In the *A multi-table procedure* example, when the procedure bumped into a case where the departing employee had orders on file, it threw the custom exception `reassign_sales` and simply stopped, undid its work and sent the exception message back for a human to deal with.

However, we can have the procedure handle it and allow the procedure to complete. For example, the handler could null out the `SALES_REP` key and send a message to another procedure that creates a log table record of each affected sales record.

We begin by creating the log table:

```
SET TERM ^;
CREATE TABLE EMPLOYEE_LOG (
    EMP_NO SMALLINT,
    TABLE_AFFECTED CHAR(31),
    FIELD_AFFECTED CHAR(31),
```

```

FIELD_VALUE VARCHAR(20),
USER_NAME VARCHAR(31),
DATESTAMP TIMESTAMP) ^
COMMIT ^

```

Next, we need to create the procedure that will take care of the logging. It is quite generic, since we'll assume that the same logging procedure might be wanted for other tasks in this system:

```

CREATE PROCEDURE LOG_ACTION (
    EMP_NO SMALLINT,
    TABLE_AFFECTED CHAR(31),
    FIELD_AFFECTED CHAR(31),
    FIELD_VALUE VARCHAR(20))
AS
BEGIN
    INSERT INTO EMPLOYEE_LOG
    VALUES (:EMP_NO, :TABLE_AFFECTED, :FIELD_AFFECTED,
    :FIELD_VALUE, CURRENT_USER, CURRENT_TIMESTAMP);
END ^

```

The last thing left to do is to add the exception-handling code to our DELETE_EMPLOYEE procedure:

```

RECREATE PROCEDURE DELETE_EMPLOYEE (
    :emp_num INTEGER)
AS
    DECLARE VARIABLE PO_NUMBER CHAR(8);
BEGIN
    IF (EXISTS(SELECT PO_NUMBER FROM SALES
    WHERE SALES_REP = :emp_num)) THEN
        EXCEPTION reassign_sales;

```

At this point, if the exception occurs, the following statements are bypassed and execution jumps to the first WHEN statement that can handle the exception.

```

UPDATE department ...
SET ...
...
...
DELETE FROM employee
WHERE emp_no = :emp_num;

```

Here is the handler block. First, it loops through the SALES table and sets the SALES_REP to null on all records in which our departed employee's code appears there. On each iteration of the loop, it calls the logging procedure, passing the employee's code along with the details of the affected Sales record:

```

WHEN EXCEPTION REASSIGN_SALES DO
BEGIN
    FOR SELECT PO_NUMBER FROM SALES
    WHERE SALES_REP = :emp_num
    INTO :PO_NUMBER
    AS CURSOR C
DO

```

```

BEGIN
    UPDATE SALES SET SALES_REP = NULL
    WHERE CURRENT OF C;
    EXECUTE PROCEDURE LOG_ACTION (
        :emp_num, 'SALES', 'PO_NUMBER', :PO_NUMBER);
END

```

After the loop is finished, the main procedure calls itself once more, to complete the processing that was skipped previously because of the exception:

```

EXECUTE PROCEDURE DELETE_EMPLOYEE1 (:emp_num);
END
END^
COMMIT ^

```

Error logs

If it is important to keep an error log, keep in mind that exceptions eventually raised to the client cause all of the work done in the module to be undone. If you are logging to a database table, the log records disappear along with the other undone work. For conditions where handlers fix or swallow every error, an internal log table will work just fine.

If you need a log that will survive an unhandled exception, use an external table. For details, refer to the topic *[Using External Files as Tables](#)* in Chapter 15.

SQLCODE and GDSCODE

You can trap the numeric error code that is passed to an internally-defined exception in the context variables `SQLCODE`, `SQLSTATE`⁴ or `GDSCODE`. This provides a very compact way to log the current exception as part of your exception handling routine.

Internally defined exceptions have an `SQLCODE`, an `SQLSTATE`⁵ and a `GDSCODE`. Your code can access one: the others will be unavailable.



Any time you try to access any of these codes outside the handler block, it will return a zero value.

The following code block framework ends with a series of exception handlers. The first two handle `SQLCODE` errors by handing them on to custom exceptions. These custom exceptions may be handled in an outer block or their purpose may be to abort the procedure and return a useful message to the client.

If neither of the targeted exceptions occurs but some other, unpredicted exception does, the `WHEN ANY` statement picks it up. Its handler calls a stored procedure to write a log record, passing the `SQLCODE`—along with others taken from the context of the block—as input:

```

BEGIN
...
    WHEN SQLCODE -802 DO
        EXCEPTION E_EXCEPTION_1;
    WHEN SQLCODE -803 DO
        EXCEPTION E_EXCEPTION_2;

```

4. `SQLSTATE` is not supported as a context variable in versions lower than v.2.5.1.

5. ditto


```

WHEN ANY DO
    EXECUTE PROCEDURE P_ANY_EXCEPTION(SQLCODE, other inputs...);
END

```

Re-raising an exception

Suppose you want to trap and log unpredicted error to an external log table before allowing the exception to take its course and abend the procedure or trigger. You can re-raise an exception: that means you can provide some handling for an exception and finish the handler a bare `EXCEPTION` statement to raise it to the final `END` statement. Execution stops and control passes back to the client, with the exception code or name and the applicable message in the error status array.

In your handler, you pick up the `GDSCODE` or `SQLCODE` and some other context variables, write the log record and then re-raise the exception:

```

BEGIN
...
    WHEN ANY DO
    BEGIN
        EXECUTE PROCEDURE P_ANY_EXCEPTION(SQLCODE, other inputs...);
        EXCEPTION;
    END
END ^

```

Exceptions in triggers

Custom exceptions in triggers have the power to enforce business rules. The example Employee database has a rule that customers who have had their credit stopped are flagged by column `ON_HOLD`, which is constrained to be either `NULL` or `'*'`. When a `SALES` record is inserted or an existing, unshipped one is updated for such a customer, the order is to be refused if the `ON_HOLD` flag is not null. Another rule says that an order that has already been shipped can not be changed.

When inserting or updating sales order records we can write `BEFORE` triggers that raise exceptions if the rules are violated, and block the operation.

For either Firebird version, we can write a `BEFORE INSERT OR UPDATE` trigger to enforce these rules.⁶

We create exceptions for two conditions:

```

CREATE EXCEPTION E_CANT_ACCEPT
    'Operation refused. REASON: Customer is on hold.' ^
CREATE EXCEPTION E_CANT_EXTEND
    'Operation refused. REASON: Order already shipped.' ^
COMMIT ^
CREATE TRIGGER BIU_SALES0 FOR SALES
ACTIVE BEFORE INSERT OR UPDATE POSITION 0 AS
BEGIN
    IF (INSERTING) THEN
    BEGIN
        IF (EXISTS (SELECT 1 FROM CUSTOMER

```

6. For v.1.0.x you will need a `BEFORE INSERT` and a `BEFORE UPDATE` trigger.

```

        WHERE CUST_NO = NEW.CUST_NO
        AND ON_HOLD IS NOT NULL)) THEN
    EXCEPTION E_CANT_ACCEPT;
END
END
ELSE /* updating */
BEGIN
    IF (OLD.ORDER_STATUS = 'shipped') THEN
        EXCEPTION E_CANT_EXTEND;
    ELSE
        BEGIN
            IF (EXISTS (SELECT 1 FROM CUSTOMER
                WHERE CUST_NO = NEW.CUST_NO
                AND ON_HOLD IS NOT NULL)) THEN
                EXCEPTION E_CANT_ACCEPT;
            END
        END
    END
END ^

```

Run-time Exception Messaging

More options are available for writing exception handlers. The static exception message, defined by CREATE EXCEPTION, can be extended replaced with a run-time string to provide a much better context for the user to identify problem data.⁷

In the next example, we use the run-time message extensions to enforce the same rules as the trigger in the previous example.

The exception:

```

CREATE EXCEPTION E_REFUSE_ORDER
    'Operation refused. REASON: ' ^

```

The trigger:

```

CREATE TRIGGER BA_SALES0 FOR SALES
ACTIVE BEFORE INSERT OR UPDATE POSITION 0
AS
    DECLARE VARIABLE ORDER_STATE SMALLINT = 0;
BEGIN
    IF (UPDATING AND OLD.ORDER_STATUS = 'shipped') THEN
        ORDER_STATE = 1;
    IF (
        EXISTS (SELECT ON_HOLD FROM CUSTOMER
            WHERE CUST_NO = NEW.CUST_NO
            AND ON_HOLD IS NOT NULL)
        AND (INSERTING OR ORDER_STATE = 0)) THEN
        ORDER_STATE = 2;
    IF (ORDER_STATE = 1) THEN
        EXCEPTION E_REFUSE_ORDER 'Order ' || NEW.PO_NUMBER || ' already shipped.';
    END
END

```

7. Not in v.1.0.x, though.

```

ELSE
  IF (ORDER_STATE = 2) THEN
    EXCEPTION E_REFUSE_ORDER
      'Order ' || NEW.PO_NUMBER || '. Customer ' || NEW.CUST_NO || ' is on hold.';
  END ^

```

In the error status array, the client will receive the name of the exception, along with its statically-defined message and the run-time message.

Error Codes Listings

Appendix VII, **Error Codes**, lists the internally-defined exceptions, including SQLCODEs, GDSCODEs, the symbols for the GDSCODEs and the English-language messages current at the release of Firebird 2.5.0.

Appendix VIII lists the **SQLSTATE** classes and codes with their English language messages.



When a Firebird binary is built, the English-language messages are extracted from an internal database. The SQLCODEs are stored but the GDSCODEs are calculated “on the fly”. The file firebird.msg, in your Firebird root directory, is built from that database, as a binary tree that the client and the server refer to when the server is running.

Events

Firebird events provide a signaling mechanism by which stored procedures and triggers can pass an alert to client applications when other applications commit changes to data. The client applications are set up to “listen” for specific events through a server-client interface, avoiding the system cost of polling for changes.

Client subsystems that poll the server for news of database state changes are not rare in the world of relational database subsystems. However, Firebird's event notification model does not expend network or CPU resources on polling or timers. It is a subsystem of the server that is maintained on and by the server. A client “registers interest” in an event and, when it is ready, it signals that it is waiting for notification.

When a transaction commits, notifications of any events that occurred are posted to those listening client applications that are waiting. The client application can then respond to the event in some manner.

Uses for Events Notification

The Firebird events notification provisions can meet numerous application requirements that call for the means to respond rapidly to changes in database state performed by others using the system. The techniques can be used in combination with external telecoms, process control, scheduling and messaging technologies to automate time-critical and state-critical response flows.

The possibilities are limitless in terms of scope and application. Some examples are:

- background data replication services are prompted to expect a new item
- a ticketing application uses the scheme as a signal to refresh open datasets in other booking offices whenever a seat allocation or a timetable change happens.

- an inventory application flashes a "stock out" message to the Purchasing department when an inventory item goes under its minimum stocking level
- retail chains are notified when a new price list is loaded
- a monitoring device on processing machinery signals the store when the levels of materials are running low

Typical originators of events are database state-changing operations—INSERT, UPDATE and DELETE operations that have been committed successfully, involving trigger code that sends a POST_EVENT call-sign. A top-level stored procedure may also send a POST_EVENT call-sign that will appear when the transaction commits.



Events are success feedback: they never appear from rolled-back transactions. Applications would typically use exceptions to deal with failures.

Elements of the Events Mechanism

Event signaling happens in triggers and stored procedures through the medium of the PSQL statement POST_EVENT.

POST_EVENT is but one piece of the mechanism—in isolation, it does nothing. It is merely a call-sign that applications listen for. It transports no information about the database event it signals—it is up to the application to provide its own context for each event on the basis of its call-sign.

The event mechanism itself consists of several interacting server-side and application pieces.

Server-side elements

The server-side elements are:

- one or more triggers or stored procedures that invoke POST_EVENT statements
- an internal event table—the destination of the POST_EVENT calls—that maintains a list of events posted to it by procedures and triggers, for the duration of the transactions in which the events occur
- an internal Event Manager subsystem that maintains a list of listeners and waiters. It acts as “traffic cop” to marshall and match up events with listeners

Application elements

On the application side, the mechanism needs

- an application that is capable of registering an interest in events
- other applications that actually perform the DML operations that the listening application is interested in.

Of course, the listening application also needs its own mechanism for responding to events.

Interface elements

Events transport from server to client uses a different pair of ports than that used for the main client/server channel (which is usually port 3050). The server and the client library find a random pair of ports to use for event traffic.



If your firewall strategy disallows random port selection by applications, this auxiliary port can be specifically configured—see the `firebird.conf` parameter `RemoteAuxPort` in Chapter 34, **Configuration Parameters in Detail**.

The software element is a routine in the client layer known as an *event callback function*. It is client-based code that is called by the server to inform the client of events immediately the transaction that posted an awaited event has committed.

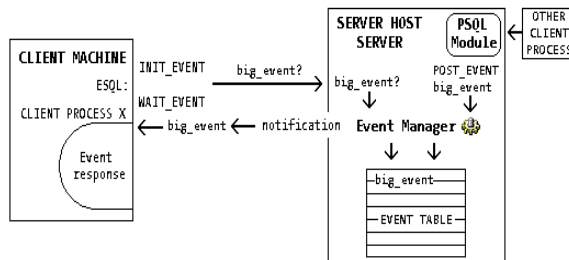
For dynamic applications that want to listen *synchronously* (q.v.), as ESQL applications do⁸, the callback function lives in the client library. Dynamic applications can—and usually do—listen *asynchronously* (q.v.). For this, they need to supply a custom callback function, known as an *asynchronous trap* (AST).

Synchronous listening

Figure 32.3 illustrates the “bare-bones” event model that is implemented in the ESQL language for embedded applications with the `EVENT_INIT` and `EVENT_WAIT` statements. Dynamic SQL has no equivalent SQL statements. For dynamic SQL applications, the same synchronous event model is implemented in the API through the `isc_wait_for_event()` function.

An ESQL application uses `EVENT_INIT` to signal that it is listening for an event and `EVENT_WAIT` to await the notification. It listens for notifications through an auxiliary port-to-port channel on the network, using the main connection channel's database handle. Once `EVENT_WAIT` is called, execution of the client application is suspended until the event notification arrives.

Figure 32.3 Synchronous signaling



A client out in the network posts an update to a row in MYTABLE. It is received by the server and executed. During the AFTER UPDATE phase, a trigger posts an event named `big_event`, to notify the event manager that it has completed the update.

The event manager adds the event to its list of events. . . At this point the update is uncommitted and the event manager does nothing further. In its list of listeners, it sees that process X is listening for that event. Process X will now wait, twiddling its thumbs, until one or more events named `big_event` are committed.

On the COMMIT, the event manager sends Process X, and any other listeners waiting for `big_event`, a notification that `big_event` happened. Even if the transaction caused `big_event` to be posted many times, the waiting client gets a single notification.

If no processes have registered an interest in `big_event`, the event manager simply ignores the `POST_EVENT` call. Any processes currently signaling `EVENT_WAIT` for `big_event` receive the notification immediately. If any processes have registered an interest in

8. For embedded (ESQL) applications, the precompiler, *gpre*, generates the code for this callback function.

big_event, but are not waiting, the event manager retains the event until they either signal wait or cancel their interest. Once the interested applications have lost interest, big_event will be erased from the table.

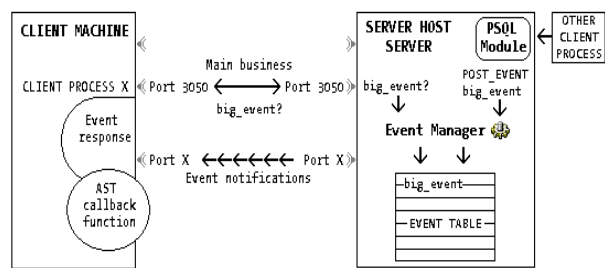
An application can wait on up to 15 events per EVENT_INIT request. It can spread events over multiple EVENT_INIT requests but, with synchronous events, it can only wait on the handle of one EVENT_INIT request at a time.

Asynchronous signalling

Synchronous signaling clearly has its limitations. In particular, it requires the application to wait an indefinite time to get notification. This limiting model was extended to support asynchronous signaling.

Under this model, an application process still registers interest and waits and listens, but it is able to continue its own execution and make database requests while awaiting the notifications. The application has its own, client-side *event queue* to manage. Figure 32.4 depicts the elements of the setup for asynchronous listening.

Figure 32.4 Asynchronous events mechanism



A stock brokering application, for example, requires constant access to the Stocks database to provide brokers with real-time information as prices fluctuate; but it also needs to watch particular stocks continuously and trigger off the appropriate Buy or Sell procedure when certain events occur.

Dynamic SQL applications use API calls to implement events listening, both synchronous and asynchronous. There is no language equivalent in DSQL and setting up the application interface from raw ingredients is rather complex.

An application registers interest in events by way of an events parameter buffer (EPB) that is populated by a call to the *isc_event_block()* function. One EPB can register up to 15 events, specifying different EPBs and event_list buffers for each call. The named events need to match—case-sensitively—the events that will be posted. Applications that need to respond to more than 15 events can make multiple calls to *isc_event_block()*.

Synchronous listening via the API

Setting up synchronous listening through the API is similar to what is needed for asynchronous signaling except that it calls this *isc_wait_for_event()* function, rather than *isc_que_events()*. As with the ESQL equivalent, EVENT_WAIT, program execution is suspended during the wait. The *isc_wait_for_event()* function listens for the wake-up notification that will occur when the server “pings” the callback function.

Asynchronous listening

Before you can use the API continuous-signaling function, *isc_que_events()*, you need a callback function (asynchronous trap, or AST) at the client, for the server to call when an event is posted.

The AST function

The AST function has to provide some form of global flag to notify the application when the server has called it. It has to process the server's event list into buffers that the application can access for its own management of the event queue. It must take three arguments:

- a copy of the list of posted events
- the length of the events_list buffer
- a pointer to that buffer



*The **API Guide** from the documentation set issued by the Inprise (Borland) company, that released the ancestral InterBase 6 beta as open source, has guidelines for writing an AST function. Links to these documents can be found at the bottom of the page www.firebirdsql.org/en/reference-manuals/ at the Firebird official website.*

The *isc_event_block()* function accepts into its *isc_callback* parameter a pointer to the AST function and, into its *event_function_arg* parameter, a pointer to the first argument of the AST. This argument generally accepts event counts as they change.

When the application calls the function *isc_que_events()* to signal events that it wants to wait for, it passes a pointer to the AST callback function, along with a list of up to 15 events. The application calls the *isc_event_counts()* function to determine which event occurred.

Multiple *isc_que_events()* calls can be operating simultaneously in a single client/server process. Applications switch off waiting with calls to *isc_cancel_events()*.



*The event block setup details for synchronous listening via *isc_event_wait()* are similar. Events are not continuous, as they are with the asynchronous *isc_que_events()* technique. Synchronous signaling does not use an AST function.*

Event alerters

Fortunately for nearly all of us, the pieces for implementing events in client applications have been encapsulated in classes and components for most of the application development interfaces, drivers and tools that support Firebird. Comprising the AST, the encapsulated API *isc_event** function calls and event parameter blocks and the client-side management of the event buffers, they are usually referred to as *event alerters*.



The term is somewhat confusing in list forums and literature, since the triggers and stored procedures that post the POST_EVENT calls are also often referred to generically as event alerters.

Using POST_EVENT

To use a POST_EVENT alerter in a stored procedure or trigger, use the following syntax pattern:

```
POST_EVENT <event-name>;
```

The parameter, **<event_name>**, can be either a quoted literal or a string variable. The string is case-sensitive. It can begin with a numeral but it cannot contain spaces. Event names are restricted to 15 characters.

When the procedure is executed, this statement notifies the event manager, which stores the alert in the event table. On committing, the event manager alerts applications waiting for the named event. For example, the following statement posts an event named “new_order”:

```
POST_EVENT 'new_order';
```

Alternatively, using a variable for the event name allows a statement to post different events, according to the current value of the string variable, e.g. event_name.

```
POST_EVENT event_name;
```



Although POST_EVENT is an SQL statement, the event name argument should not be prefixed by a colon.

An Example

The following script creates a trigger that posts an event to the event manager whenever any application inserts data in a table:

```
SET TERM ^;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
    POST_EVENT 'new_order';
END ^
SET TERM ; ^
```

Trigger or procedure?

POST_EVENT is available to both triggers and stored procedures, so how does one decide which is the better place to post events?

A good rule of thumb is to use triggers when applications need know about row-level events—either single-row or multiple, depending on the scope of the transactions—and procedures to signal those events whose impact on applications is broader.

It is only a guideline: often, procedures have row-level scope and yet the interested client wants to know about a particular operation when it happens to be performed by that procedure. A POST_EVENT from a trigger in this case would be unable to tell the listener anything about the context of the event and the designer might wish to use the procedure-based event call-sign to ascertain which kind of caller was responsible for the work. Alternatively, the designer might wish to place the event in a trigger to ensure that a particular DML action is consistently signaled, regardless of the context in which it is executed.



The
Firebird Book
A Reference for Database Developers

SECOND EDITION

PART VII



Configuring Firebird

33

CONFIGURING FIREBIRD AND ITS ENVIRONMENT

Firebird does not require the intense and constant reconfiguration that many other heavy-duty RDBMS systems do. However, a range of configuration options is available for customizing a Firebird server and the environment in which it runs when special needs must be met.

Default Configuration

The defaults configured when Firebird is installed should suit most development environments.

Finding the Firebird Root Directory

The root directory of your Firebird installation is used in many ways, both during installation and as an attribute that server routines, configuration parameters and clients depend on. Because several ways exist to tell the server where to find a value for this attribute, developers and system administrators should be aware of the precedence trail that the server follows at startup, to determine it correctly.

- 1 FIREBIRD environment variable: On any platform, the first place the server looks is the (optional) global environment variable FIREBIRD. If it finds this variable, its value is used unconditionally.
- 2 If the FIREBIRD environment variable is not present, the next signpost on the trail applies to Windows platforms only. It seeks the Registry key
`HKEY_LOCAL_MACHINE\SOFTWARE\Firebird Project\Firebird Server\Instances.`
 and looks for the field `DefaultInstance`. If it finds a valid directory path in this field, this is the value used. Other platforms do not have an equivalent signpost.

- 3 If the root directory is still not detected, then the interim root directory is assumed to be the level above the running process (. . \ on Windows, .. / or the link to /proc/self/exe on POSIX, as applicable)
- 4 The startup procedure now looks in this location for the firebird.conf file. If it finds firebird.conf, it looks for the **RootDirectory** parameter. If this is present, its value becomes the final root directory; otherwise the interim value in step 3 becomes final.



If firebird.conf is not found in this location, it may mean that the root directory has been incorrectly detected because of a non-standard installation. The engine must find the root directory files. If you encounter security or filesystem errors when logging in or during runtime, you should review your installation paths to ensure that the steps above will correctly resolve the location of the root files and sub-directories.

Modifying the Firebird Root Location

From v.2.5 forward, it is possible to build Firebird for deployment on POSIX platforms such that its root structure is relocatable. In this build style, the installation directory can be deduced from the executables and libraries and need not refer to the default, hard-coded root in /opt/firebird/.

Of course, in this mode, the practice of copying utilities to non-standard locations and relying on libraries being loaded from /opt/firebird would not work “out of the box”. Symbolic links would be required.



In a future release—possibly v.3.0—this feature will become enabled by default. Although builds from sources are outside the scope of this book, if you want it now, the secret is to pass the argument --enable-binreloc to autogen.sh.

Environment Variables

A small set of environment variables is available for optional use to suit particular conditions or requirements.

The effects of setting environment variables is similar on all platforms. Platform differences arise in where and how they are set and unset. On Windows, the type of environment variable to set and the method of setting it varies from one version of Windows to another.

Windows

Table 33.1 shows which type to set (if applicable) and where and how to set them.

Table 33.1 Setting environment variables on Windows

Windows version	Variable type	Description	In Registry
Server and Professional editions	User Variables for <user>. Makes variables available to applications being run by <user> and only if <user> is logged on	Use to restrict visibility of variables to the appointed user. System Properties dialog, from the System applet on the Control Panel or by right-clicking the My Computer icon on the desktop and selecting Properties. Select Advanced Environment Variables New.. Windows NT: Select Advanced Environment New..	Yes
	System Variables Available to the entire system (all services, all users).	Use if running Firebird as a service Select Advanced Environment Variables New.. (or Edit, if applicable)	Yes
	SET commands written from a command window. Variables can be accessed only by processes started from the command line.	Useful to set an environment variable in a very temporary fashion, e.g., ISC_USER and ISC_PASSWORD for simplifying access with command line tools during an admin task. Use SET <variable_name>=<variable_value> to set the variable; SET <variable_name>= , leaving value blank, to unset.	No
Windows 9X, ME	Not applicable	Use Notepad and set environment variables in autoexec.bat or config.sys. Format is SET <variable_name>=<variable_value> For example, SET FIREBIRD=C:\PROGRA~1\FIREBIRD To view all current environment variables, type SET in a command window	No

POSIX

On Linux, MacOSX and other POSIX, the easiest way to define environment variables is to add their definitions to the system-wide default shell profile.

The root user can also choose to either

- issue `setenv()` commands from a shell OR shell script
- for temporary use, set and export a variable from a shell, e.g.,
`export ISC_USER=SYSDBA`

The Variables

EDITOR

Sets the path to the non-default text editor that is to be used by the *isql* utility. The default editors—which do not need to be set explicitly—are *vim* on Linux and MacOSX and *Notepad* on Windows.

FIREBIRD

The FIREBIRD environment variable (INTERBASE in version 1.0.x), if configured, is used during both installation and runtime, on all platforms, to locate the root directory of the Firebird server installation. If present, it overrides all other settings—installation kit defaults, Windows Registry settings, *firebird.conf* configuration, operating system global paths defaults, etc.

During installation, FIREBIRD points to the root directory where Firebird is (to be) installed. The value must point to a fully-qualified path that exists within the host machine's own physical filesystem. At start-up, the server reads non-default settings from a configuration file named *firebird.conf* (or *ibconfig/isc_config* in version 1.0.x) which must be located in the directory assigned to the FIREBIRD (or INTERBASE) variable. This directory must be the parent of the */bin* directory where the Firebird binaries reside. It is also the default location of the message and lock files, *firebird.msg* (*interbase.msg*) and the *hostname.lck* file(s) (where *hostname* is the name of host server machine).



*You can specifically configure different locations for *firebird.msg* (*interbase.msg*) and *firebird.lck* (*interbase.lck*) by setting the environment variables *FIREBIRD_MSG* (*INTERBASE_MSG*) and *FIREBIRD_LOCK* (*INTERBASE_LOCK*). See below.*

If the FIREBIRD variable is not configured, the defaults are used: */opt/firebird* on Linux/UNIX platforms or *C:\Program Files\Firebird\Firebird_n_n*, (e.g., *2_5*) on Windows platforms. If Firebird is installed in a path that is different to the default and the FIREBIRD variable is configured, components can read the variable to find the installation path.

On Windows, client applications can also find the installation path by reading the field *DefaultInstance* under the Registry key that is installed in a client-only installation.

HKEY_LOCAL_MACHINE\SOFTWARE\Firebird Project\Firebird Server\Instances

For more discussion of the trail followed by the server to locate its files, refer to the topic [The Firebird root directory](#), above. Look up the topic [Performing a Client-Only Install](#) in Chapter 2 for information about this topic.

FIREBIRD_LOCK

FIREBIRD_LOCK (INTERBASE_LOCK in v.1.0.x) sets the location of the shared memory files, including lock files. The Value must point to a fully qualified path that exists within the host machine's own physical filesystem.

FIREBIRD_MSG

FIREBIRD_MSG (INTERBASE_MSG in v.1.0.x) sets the location of the Firebird message file. The Value must point to a fully qualified path that exists within the host machine's own physical filesystem.



FIREBIRD_LOCK and FIREBIRD_MSG are independent of each other and can be set to different locations.

FIREBIRD_TMP

By default, Firebird will use the global temporary file space, usually configured as a system default by the environment variable TMP (see below). The FIREBIRD_TMP (v.1.0.x INTERBASE_TMP) environment variable is one means by which a custom location for Firebird's sort files can be configured. The Value must point to a fully qualified path that exists within the host machine's own physical filesystem.

Other options are available for defining the location of these files. See the topic describing configuration of the Firebird configuration parameter **TempDirectories** (*tmp_directory* in Firebird 1.0.x), in the next chapter, **Configuration Parameters in Detail**.

HOME

If set, defines the location of the .qli_startup file¹. Note that *qli* can also substitute any environment variable using `$(variable name)` in code.

ISC_INET_SERVER_HOME

Sets the working directory for Classic server *ONLY*. Make certain this variable is not set if you are running Superserver or Superclassic.

ISC_MSGS

Sets where `firebird.msg` is located, overriding FIREBIRD_MSG.

ISC_PATH

Sets a default path for locating or (from v.2.5) creating databases. Normally, you would set the `firebird.conf` parameter **DatabaseAccess** with a *Restrict* argument to control where databases should be searched for. If you set this variable, it will override any setting made for this parameter.

ISC_USER and ISC_PASSWORD

The effect of this dangerous pair of environment variables is to give SYSDBA access to the Firebird server and its databases, via the command-line tools or other client applications, to anyone who can log in to the host machine. They are a convenience for developers.

If you do not provide a user name and password when you connect locally to a database or when you run the command-line utilities such as *gbak*, *gstat*, and *gfix*, Firebird looks to see if the ISC_USER and ISC_PASSWORD environment variables are set; if they are, Firebird lets you log in without entering a password. Never allow these variables to remain configured on a server that has production databases running unless your server room is physically very secure!

LC_MESSAGES

Sets where where a localized version of `firebird.msg` is located.

TMP (or TEMP on Windows NT)

A variable that can be used to designate the default location of temporary space globally on the server machine. It will be used by Firebird—in competition with other applications—to store temporary sort files if dedicated sort space is not otherwise configured.

1. The *qli* (Query Language Interpreter) utility is a command-line tool for accessing data using Firebird's internal *gdml* query language. It still works with Firebird and is distributed in the standard installations. If you are interested, the document at http://www.ibphoenix.com/files/qli_syntax.pdf contains an outline of the *gdml* syntax.



Sort space is space on disk where the engine stores the intermediate output sets, in run-time temporary files, for queries that have to be sorted or aggregated. Firebird stores these files in RAM, if it can, and pages them out to disk sort space only if resources become exhausted.

v.1.0.x Firebird 1.0.x uses only disk files for such storage.

The global environment variable TMP (or TEMP on some systems) points to a directory path on the server where applications should store temporary files. Firebird will attempt to store temporary sort files here if the FIREBIRD_TMP environment variable (see above) is not defined and no other sort space has been configured in the Firebird configuration file (see TempDirectories and tmp_directory, below).

ISQL script files on a client

The interactive command line SQL utility, ISQL, provides the capability to “record” sequences of interactive SQL commands to a script file, using the OUTPUT switch. On a client machine, the TMP setting is the only place to control the space where these script files will be stored if an absolute path specification is not supplied. If the TMP location is not set, a Firebird client uses any temporary directory it finds defined for the local system, usually the /tmp filesystem on a Linux/UNIX client, or the user’s temp folder on a Windows client.

VISUAL

Sets the path to the non-default text editor that is to be used by the *qli* utility. The default editors—which do not need to be set explicitly—are *vim* on Linux and MacOSX and *Notepad* on Windows. However, if VISUAL is not set and EDITOR is set, then *qli* will use that editor.

The Firebird Configuration File

The Firebird configuration file, named `firebird.conf`, is present in all Firebird versions since v.1.5. It is required to be present in the root directory of the Firebird installation.

When the Firebird server startup process reads the configuration file, it adjusts its runtime flags to any non-default values contained in the configuration file.

The file will not be read again until next time the server is restarted.

The default configuration parameters and their values are listed in the configuration file, commented out by “#” comment markers. It is not necessary to uncomment the defaults in order to make them visible to the server’s startup procedure.

The configuration file can be edited with any plain text editor, e.g. *vim* (Linux) or *Notepad* (Windows). Do not copy the file from a Windows machine to a Linux one, or vice versa, or from either of these platforms to MacOSX. Each system stores line breaks differently.



Changes made to `firebird.conf` do not take effect until the server process is restarted.

Entries in `firebird.conf` are in the form:

```
parameter_name = value
```


Parameter Syntax

The syntax takes very simple forms, with no strange requirements for quotes or other punctuation, except that lists of tree-roots for file-related parameters are separated by semi-colons.

Comments

A line of comment text can be placed anywhere in the file. The comment marker is the hash symbol (#).

All characters to the right of the # symbol are treated as comment. This means you can use a comment marker to convert a parameter sentence into a comment; that is, to “comment out” a sentence, or to comment out the right-hand part of a sentence.



Notice that most parameters carry a value, with the sentence containing the parameter commented out. The value shown initially is the default, that is used automatically by the engine. When changing a parameter to a different value, remember to delete the # comment marker.

Examples

```
# This is a comment
# DefaultDbCachePages = 2048    # This is an end-of-line comment
```

Values

Values for the configuration parameters come in three types: integer, boolean and string.

Integer Parameters

As expected, an integer parameter takes an integer value. Take note of the comments associated with these, as some specify a count of bytes, rather than KB or MB.

Example

```
#DatabaseGrowthIncrement = 134217728
```

The default for this is 128MB, i.e., $128 * 1024 * 1024 = 134217728$ bytes.



For those with insufficient fingers, conversion tables for KB-to-bytes and MB-to-bytes can be found at the end of firebird.conf.

Boolean Parameters

The convention used for Boolean values is zero (0) represents False while any non-zero value represents True. Ah - so many choices! For clarity and consistency, try to stick to 1 for your True value in all cases.

String Parameters

String parameters take simple, unquoted strings. From the engine's point of view, white space is not especially significant. However, if a space character is a significant element of the string you are supplying, you must supply it in its rightful position.

Examples

```
# RootDirectory = /opt/firebird
# RemotePipeName = pipe47
```

Editing Parameters

Use a text editor that creates text that is ‘native’ to the platform, i.e., one that correctly handles line breaks and new lines. To set any parameter to a non-default setting, delete the comment (`#`) marker and edit the value.

- **parameter_name** is a string that contains no whitespace and names a property of the server being configured.
- **value** is a number, Boolean (1=True, 0=False) or string that specifies the value for the parameter



On Linux, you should assume that parameter names are case-sensitive.

You can edit the configuration file while the server is running. To activate configuration changes, it is necessary to stop and restart the service.

Version Differences

By and large, the set of parameters available in `firebird.conf` just accumulates from version to version. However, over time a few have been changed, deprecated or entirely dropped. Where this has happened, it is noted.

Nevertheless, if you include a parameter that is not supported in the release version you are running, it won’t break anything: it will just be ignored.



By an unfortunate oversight in the build system, `firebird.conf` did not exhibit the version it belonged to until versions 2.1.4 and 2.5.1. If you find that information missing from the header text in the configuration file on servers you are running, it is suggested you add your own identifier to the comments.

Firebird 1.0.x and InterBase 6

In the very old versions, the name of the configuration file depends on the operating system:

- On Linux/UNIX the name is `isc_config`
- On Windows, the name is `ibconfig`.

Parameter names and syntaxes are not interchangeable with those in `firebird.conf`. The format, size and number of parameters are more restrictive. The `ibconfig/isc_config` format is

```
parameter_name    value
```

Each line of the file is limited to 80 characters. The white space between the name and the value can be tabs or spaces, as desired, to please the eye. As in `firebird.conf`, unused parameters and installation defaults are commented with `'#'`.

“Missing” parameters

In Firebird 1.0.x, some optional parameters, requiring settings that could not be configured as defaults, were omitted from the configuration file.

Many new parameters have been added to versions over the years. Although most of the original IB6/Firebird 1.0 parameters are still supported, there has never been any back-porting of new parameters to those old files. However, if a missing “new” parameter

pertinent to `ibconfig` or `isc_config` is needed and is applicable to the old server, it can be added.

Parameters in Detail

For the full documentation of each parameter, see the next chapter, **Configuration Parameters in Detail**.

Configuring the TCP/IP Port Service

By default a Firebird server listens on port 3050 for TCP/IP connection requests from clients. Its registered port service name is ***gds_db***. The good news is that, if you can go with these defaults, you have to do nothing on either server or clients to configure the port service.

You can use a different port, a different port service name, or both. You might need to do this if port 3050 is required for another service, for example, a concurrently running `gds_db` configured for a different version of Firebird or for an InterBase® server. There are several ways to override the defaults. Both the server and the clients must be configured to override the port service name or number, or both, in at least one of these ways:

- 1 in the client connection string
- 2 in the command used to start the server executable
- 3 by activating the `RemoteServicePort` or `RemoteServiceName` parameters in `firebird.conf` (V.1.5 onward)
- 4 in the daemon configuration (for Classic on POSIX)
- 5 by an entry in the Services file

Before examining each of these techniques, it will be helpful to look at the logic used by the server to set the listening port and by the client to set the port that it should poll for requests.

How the server sets the listening port

The server executable has an optional command-line switch (`-p`) by which it can signify either the port number it will be listening on or the name of the port service that will be listening. At this point, if the switch is present, either port number 3050 or the port service name (`gds_db`) is replaced by the argument supplied with the `-p` switch.

Next—or first, if there is no `-p` switch—the server checks `firebird.conf` to look at the parameters ***RemoteServiceName*** and ***RemoteServicePort***:

- If both are commented with `"#"` then the defaults are assumed and no further resolution occurs. Any `-p` argument stands and the “missing” argument remains as the default.
- If ***RemoteServiceName*** has been uncommented, but not ***RemoteServicePort***, then the port service name is substituted, but only if it has not been overridden already by the `-p` switch.

- If **RemoteServicePort** has been uncommented, but not **RemoteServiceName**, then the port number is substituted only if it has not been overridden already by the `-p` switch.
- If both **RemoteServicePort** and **RemoteServiceName** are uncommented, then the **RemoteServiceName** takes precedence if it has not already been overridden by a `-p` argument. If there is already a port service name override, the **RemoteServiceName** value is ignored and the **RemoteServicePort** value overrides 3050.
- At this point, if an override of either the port number or the service name has been signaled, the server proceeds to check the Services file for an entry with the correct combination of service name and port number. If a match is found, all is well. If not, and the port service name is not `gds_db`, the server will throw an exception and fail to start. If `gds_db` is the port service name and it cannot be resolved to any other port, it will map to port 3050 automatically.

If the default port number or service name is to be overridden, then you may need to make an entry in the Services file. To understand whether it will be necessary with your override choices, follow through the steps outline below in the topic **Configuring the services file**.

Using the `-p` switch override

Starting the server with the optional `-p` switch enables you to override either the default port number (3050) or the default port service name (`gds_db`) that the server uses to listen for connection requests. The switch can override one, but not both. You can use the `-p` switch in combination with a configuration in `firebird.conf` to enable an override to both the port number and the port service name.

Syntax for TCP/IP

```
server-command <other switches> -p port-number | service-name
```

For example, to start the Superserver as an application and override the service name `gds_db` with `fb_db`:

```
fbserver -a -p fb_db
```

Or, to override port 3050 to 103050:

```
fbserver -a -p 103050
```

Syntax for WNet redirection

On a WNet network, replace the `-p` switch argument syntax above with the following “backslash-backslash-dot-at” syntax:

```
fbserver -a -p \\.\@fb_db
```

or

```
fbserver -a -p \\.\@103050
```

Classic on POSIX: the `inetd` or `xinetd` daemon

With Firebird Classic server on Linux or UNIX, the *inetd* or *xinetd* daemon is configured to listen on the default port and broadcast the default service name. The installation script will write the appropriate entry in the configuration file `/etc/inetd.conf` or `/etc/xinetd.conf`.

Problems attaching to a Classic server are often due to missing or bad port service entries in this file. You can check the current entry by opening it in a plain text editor, e.g. *vim*, and

editing it if necessary. Following is an example of what you should see in `xinetd.conf` or `inetd.conf` after installing Firebird Classic on Linux:

```
# default: on
# description: FirebirdSQL server
#
service gds_db
{
    flags REUSE KEEPALIVE
    socket_type = stream
    wait = no
    user = root
    # user= @FBRUser@
    log_on_success = USERID
    log_on_failure = USERID
    server = /opt/firebird/bin/fb_inet_server
    disable = no
}
```

If you configured the port service to be different to the defaults, you will need to alter `xinetd.conf` or `inetd.conf` accordingly. Restart *xinetd* (or *inetd*) with `kill -HUP` to make sure the daemon will use the new configuration.



*Connection requests will fail if both *xinetd* (or *inetd*) and *fbserver* attempt to listen on the same port. If your host machine has Classic installed alongside Superserver or Superclassic, it will be necessary to set things up so that each server version has its own service port.*

Using a configuration file parameter

You can configure either ***RemoteServiceName*** or ***RemoteServicePort*** in `firebird.conf` to override either the default port number (3050) or the default port service name (`gds_db`) that the server uses to listen for connection requests.

The engine will use one ***RemoteService**** parameter, but not both. If you configure both, it will ignore ***RemoteServicePort*** in all cases, except where the server start command was invoked with the `-p` switch supplying an override to the port service name. Thus, you can use the `-p` switch and a ***RemoteService**** parameter, in combination, to override both port number and service name.

If the default port number or service name is to be overridden, then you need to make an entry in the Services file.

GOTCHA!

*If you uncomment *RemoteServiceName* or *RemoteServicePort*, but leave the default values intact, they will be treated as overrides. It will then be necessary to make an explicit entry in the services file for the default port service settings.*

Setting up a client to find the service port

If you set up your server with the installation defaults (service `gds_db` listening on port 3050) then no configuration is required. If the server is listening on a different port or is using a different port service name, the client application and/or its host machine need some enabling configuration to help the Firebird client library to find the listening port.

The connection string used by a client can include information for polling the server's listening port in various ways. Clients can optionally use a local copy of `firebird.conf`. Changes may also be needed in the client's services file.

Using the connection string

If only the port name or the service name has been reconfigured, then include the alternative port number or service name in the connection string. This works for all versions of Firebird.

Syntax for TCP/IP connections

To connect to a database named server named 'hotchicken' that is broadcasting on port 3050 with the service name `fb_db`, the connection string would be:

For POSIX:

```
hotchicken/fb_db:/data/leisurestore.fdb
```

Or, if the service name is `gds_db` and the port number is 103050:

```
hotchicken/103050:/data/leisurestore.fdb
```

For Windows, example host name 'winserver':

```
winserver/103050:D:\data\leisurestore.fdb
```

```
winserver/fb_db:D:\data\leisurestore.fdb
```

Notice that the separator between the server name and the port is a slash, not a colon. The colon before the physical path string is still required.

Syntax for WNet connections

On a WNet network, use UNC-style notation:

```
\\winserver@103050\d:\leisurestore.fdb
```

or

```
\\winserver@fb_db\d:\leisurestore.fdb
```

If the server's port number or service name is an override, then you need to make an entry in the Services file (discussed below).

Using port syntax with database aliases

To connect through a non-default port with a database alias, affix the port number or service name to the server name, not to the alias. For example, suppose the database alias is stored in `aliases.conf` as

```
hotstuff = /data/leisurestore.fdb
```

Your application's connection string for server 'hotchicken' would be:

```
hotchicken/fb_db:hotstuff
```

or

```
hotchicken/103050:hotstuff
```

Using a copy of `firebird.conf`

You can optionally include a client-side copy of `firebird.conf` in the firebird root directory and configure ***RemoteServiceName*** or ***RemoteServicePort*** to help the client to locate the server port.

- You can configure one of these two parameters to extend the override provided for the other one through the connection string (above); or to override only the **RemoteServiceName** or the RemoteServicePort without using the connection string to do it.
- If you need to avoid passing the port service name or the port number in the connection string and the server is using non-defaults for both, you can configure both **RemoteServiceName** and **RemoteServicePort**. You would use this technique if your client application needed to retain the capability to connect to old InterBase or Firebird 1.0 servers.

Location of Firebird artifacts on clients

When you rely on `firebird.conf` on client machines, it is important that the client library knows where to find it. You will need to set up a Firebird root directory and inform the system of its location. Set the FIREBIRD environment variable to do this.

Windows clients can alternatively use a Registry key. With a correct client setup, you can also make use of a local version of the message file, `firebird.msg` for cases when you are using the command-line utilities and the Services API remotely.

Configuring the services file

You do not need to configure a service port entry for your Firebird server or clients if the server uses the installation defaults, `gds_db` on port 3050. If `gds_db` is the port service name and it cannot be resolved to any other port, it will map to port 3050 automatically.

If you are configuring the service port for a different port or service name, both the server and the client must be explicitly updated to reflect the reconfiguration. On both Linux and Windows, this information is stored in the `services` file.

Locating the services file

- On Windows, this file is `$sysdir$\drivers\etc\services`. (On Windows 9X/ME, it is `C:\windows\services`.)
- On Linux/UNIX it is usually `/etc/services`.

Open the file using a text editor and either add a line or edit the existing `gds_db` entry. The format is:

```
<service name> <port number>/<protocol> [aliases...] [#<comment>]
```

For example:

```
fb_db 3050/tcp #Firebird server 2.5
```

or

```
gds_db 103050/tcp #Firebird server 2.5
```

Embedded Server

Configuring the “embedded server” model of Firebird is, in every significant way, the same as configuring a full server. Embedded is to be regarded as a deployment model, not a component of a development environment. In fact, it is highly recommended to develop your embedded application using the appropriate full server version in the development

environment, particularly if your platform is Windows and the release version is lower than v.2.5. Which server model is “appropriate” depends on release version and platform.

- From Firebird 2.5 onward, the embedded libraries for all platforms implement a specialised form of Superclassic. The embedded server should be configured as Classic.
- From Firebird 1.5 to 2.1.x
 - on Windows, the embedded server implements Superserver
 - on Linux and MacOSX, the embedded server implements Classic
- In Firebird 1.0.x, embedded server is not supported on Windows. On Linux, it is Classic.

Review the topic *Installing an Embedded Server* in Chapter 2 for more specific comments about conditions that might affect configuration.

CHAPTER 34

CONFIGURATION PARAMETERS IN DETAIL

Entries in `firebird.conf` are in the form:

```
parameter_name = value
```

Many parameters have default values. The default value for a parameter appears in `firebird.conf` commented out by a ‘#’ marker.

Refer to the previous chapter, **Configuring Firebird**, for a more detailed description of the form and syntax of the entries.



Remember that no changes take effect until the next time the server process is restarted.

Settings for All Platforms and Servers

The following parameters are applicable to all platforms and server models, except where noted.

AuditTraceConfigFile

v.2.5

This parameter points to the name and location of the file that the Firebird engine is to read to determine the list of events required for the next system audit trace. By default, the value of this parameter is empty, indicating that no system audit tracing is configured.



The file `fbtrace.conf`, found in Firebird's root directory, is a template for composing an audit trace configuration file. It contains the full list of available events, with format, rules and syntax.

For more information, see the topic **Trace and Audit Services** in Chapter 38, **Monitoring and Logging Features**.

Authentication

v.2.1

Used for enabling Windows ‘trusted user’ authentication on Windows-based servers, v.2.1 and above. Although Firebird has always catered for host user access on non-Windows platforms, it is not available for Windows in versions prior to v.2.1.

The parameter takes a string argument which must be one of ‘native’, ‘trusted’ or ‘mixed’.

- *trusted* makes use of the ‘trusted user’ authentication that Windows server platforms conduct and blocks native Firebird logins. Under the right conditions, this may be the most secure way to authenticate on Windows.



On other platforms, this setting has the similar effect of blocking native Firebird logins and enabling platform user access directly.

- *native* sets the traditional Firebird server authentication mode, requiring users to log in using a user name and password defined in the security database.
- *mixed* allows both.

The installation default is *native*.



Take care when configuring *mixed* or *trusted*: security screening is not handed over to Firebird native authentication after a platform user has got inside the gates. It can leave a significant security hole if administrators use it under the mistaken assumption that trusted user authentication adds some kind of extra security layer.

Changes

Until v.2.1.2, *mixed* was the installation default mode for **Authentication**. Because of the risks it could expose vulnerable servers to, a deliberate change was made at the v.2.1.2 sub-release to set the installation default to *native*.

Under all sub-releases of v.2.1, configuring *mixed* or *trusted* mode confers SYSDBA privileges on Windows domain administrators automatically. In Firebird 2.5 and beyond, there is more to it. Firebird 2.5 introduced improved features for enabling trusted Windows users to obtain escalated privileges on the Firebird server and within databases. The parameter discussed here—**Authentication**—must be explicitly configured to either *trusted* or *mixed* before it becomes possible to enable the additional v.2.5 features.

Refer to topics in Chapter 36, **Protecting the Server and Its Environment** and Chapter 37, **Database-level Security**, pertaining to the new RDB\$ADMIN role in ODS 11.2 databases and mapping SYSDBA privileges to domain administrators.



The **Authentication** parameter is to be deprecated in v.3.0 and removed in the following major version. It will be superseded by a new parameter, **AuthServer**, in which the used plug-in authentication module[s] shall be enumerated.

Complete Boolean Evaluation

v.1.5

Establishes the Boolean evaluation method (complete or shortcut). Its type is Boolean, with default 0 (False), i.e., the server will “short-cut” a Boolean evaluation expression involving the AND or OR predicates, by returning a result as soon as a True or False is obtained that cannot be affected by the results of any further evaluation.

Under very rare (usually avoidable) conditions, it might happen that an operation inside an OR or an AND condition remains unevaluated due to the shortcut behavior, with potential to affect the outcome of the original result. If you have the misfortune to inherit an application that has such characteristics in its SQL logic, you might wish to use this

parameter to force complete evaluation until you have the opportunity to perform surgery on it.



Don't overlook the fact that this flag affects all Boolean evaluations performed in all databases on the server.

ConnectionTimeout

v.1.5, enhanced in v.2.5, v.2.1.4 and v.2.0.6

Seconds to wait before abandoning an attempt to connect (integer). The default is 180 seconds.

Originally, this parameter was implemented to affect remote protocols. However, on heavily loaded v.2+ Windows systems, where XNET is the subsystem used for the local (“serverless”) protocol, local connect could fail due to the client timing out while waiting for the server to set the *xnet_response_event*. To help with this problem, the ConnectionTimeout parameter was enhanced to affect XNET connections as well, in addition to TCP/IP.



The enhancement noted above was implemented in v.2.5 and subsequently backported to v.2.1.4 and v.2.0.6. It is thus NOT implemented in prior 2.1.x and 2.0.x sub-releases, nor in v.1.5.x.

DatabaseAccess

v.1.5

This parameter enables the tightening of control over access to database files and supports the database-aliasing feature. This parameter provides options to restrict the server's access to aliased databases only, or to only databases located in specific filesystem trees.

DatabaseAccess may be Restrict, Full or None.

- *Restrict* allows you to configure the locations of attachable database files to a specified list of filesystem tree-roots. Supply a list of one or more tree-roots, separated by semi-colons, to define one or more permissible locations.

For example,

Unix: `/db/databases;/userdir/data`

Windows: `D:\data`

From v.2.5 forward, if Restrict is configured, Firebird uses the first tree root in the list as the default location for creating a database.

Relative paths are treated as relative to the path that the running Firebird server recognises as the root directory. For example, on Windows, if the root directory is `C:\Program Files\Firebird`, then the following value will restrict the server to accessing database files only if they are located under `C:\Program Files\Firebird\userdata`:

`Files\Firebird\userdata`:

`DatabaseAccess = Restrict userdata`

- *Full* (the default) permits database files to be accessed anywhere on the local filesystem. However, it is not recommended
- *None* permits the server to attach only databases that are listed in `aliases.conf`.

It is strongly recommended that you set this option and make use of the database-aliasing feature. Database aliasing, is described, with examples, in Chapter 4, **Operating Basics**.

DatabaseGrowthIncrement

v.2.1

For control of disk space preallocation, this parameter represents the upper limit for the size, in bytes, of the chunk of disk that will be requested for preallocation as pages for writes from the cache. The default is: 134,217,728 bytes (128 MB).

When the engine needs to initialize more disk space, it allocates a block that is 1/16th of the space already allocated, but not less than 128 KB and not greater than the DatabaseGrowthIncrement value. The DatabaseGrowthIncrement value can be raised to increase the maximum size of newly-allocated blocks to more than the default 128 MB.



- The lower limit of the block size is purposely hard-coded at 128 Kb and cannot be reconfigured.
- Space is not preallocated for database shadow files.
- Preallocation is disabled for a database that has the “No reserve” option set.

Disabling Preallocation

To disable preallocation, i.e., have the engine revert to the older behaviour of requesting blocks one page at a time when space is required for a write, set **DatabaseGrowthIncrement** to zero.

DeadlockTimeout

v.1.5

Integer determining how long, in seconds, the Lock Manager will wait after a conflict before purging locks from dead processes and performing an extra scan for deadlocks. The default is 10 seconds.

Normally, the engine detects deadlocks instantly and would have no reason to use this parameter. It is a release when something goes wrong with conflict resolution and the engine needs to intervene.



Do not be tempted to set it too low and thus raise the number of extra deadlock scans to a level that would slow performance down generally.

DefaultDbCachePages

v.1.5

Sets the server-wide default (integer) number of database pages to allocate in memory for each database. The configured value can be overridden at database level.

The default value for SuperServer is 2048 pages. For Classic, it is 75.

SuperServer and Classic allocate and use cache memory differently. There is no "formula" that can be applied to set an optimal default cache size to suit all needs. However, the factors are discussed in greater detail in Chapter 15, Creating and Maintaining a Database in the topic The database cache.

v.1.0 **database_cache_pages** in isc_config/ibconfig

DummyPacketInterval

v.1.5

v.1.0 **dummy_packet_interval** in ibconfig/isc_config

This is an old InterBase timeout parameter which sets the number of seconds (integer) the server should wait on a silent client connection before sending dummy packets to request acknowledgment. It is set by default to 0 (disabled) on Firebird 1.5 and above and to 60 on Firebird 1.0.x.



This parameter should not be enabled on Windows at all and it is not recommended on other operating systems.

To disable this setting on Firebird 1.0, open `ibconfig` (Windows) or `isc_config` (other systems) and add the line:

`dummy_packet_interval=0`

Normally, Firebird keeps track of active connections using the `SO_KEEPALIVE` socket option, with a default timeout period of two hours. If you need to alter the timeout period, adjust server settings to suit:

- On POSIX servers, modify the contents of `proc/sys/net/ipv4/tcp_keepalive_*`.
- For Windows, obtain instructions from the article at <http://support.microsoft.com/default.aspx?kbid=140325>

FileSystemCacheThreshold

v.2.5 (in v.2.1, MaxFileSystemCache)

Sets a threshold determining whether Firebird will allow the page cache to be duplicated to the filesystem cache or not. If this parameter is set to any (integer) value greater than zero, its effect depends on the current default size of the page cache: if the default page cache (in pages) is less than the value of ***FileSystemCacheThreshold*** (in pages) then filesystem caching is enabled, otherwise it is disabled.



This applies both when the page cache buffer size is set implicitly by the `DefaultDBCachPages` setting or explicitly as a database header attribute. It applies to all platforms.

Thus,

- To disable filesystem caching always, set ***FileSystemCacheThreshold*** to zero
- To enable filesystem caching always, set `FileSystemCacheThreshold` to an integer value that is sufficiently large to exceed the size of the database page cache. Remember that the effect of this value will be affected by subsequent changes to the page cache size.

The default setting is 65536 pages, i.e. filesystem caching is enabled.



If the configured cache size affecting a particular database is greater than the `FileSystemCacheThreshold` then the setting for `FileSystemCacheSize` (see below) will have no effect on that database.

`MaxFileSystemCache`, introduced in Firebird 2.1, is not a valid parameter in Firebird 2.5 and above.

FileSystemCacheSize

v.2.5

Controls the maximum amount of RAM used by a Windows file system cache on 64-bit Windows XP or a Microsoft Server 2003 host with Service Pack 1 or higher.

At the V.2.5 initial release, it had no effect on POSIX host systems.

The setting for this parameter is an integer expressing the percentage of the total physical RAM that is available to the OS. To be valid, settings must be within the range 10 (per cent) to 95 (per cent), or explicitly set to 0 to enforce the host caching settings. Numbers outside that range will assume the default, which is 30 (per cent).

Windows Security Privileges

The OS user needs the *SeIncreaseQuotaPrivilege* in order to adjust the filesystem cache settings. This right is built in for users with Administrator privileges and for service accounts and it is also granted to the Firebird service account explicitly by the Windows installer.

Under other conditions, e.g., embedded, or where the Firebird server is run as an application, or in a custom service installation, the user may not have that privilege. The process startup does not fail as a result of this misconfiguration: it will write a warning to the `firebird.log` and start-up will simply proceed with the host OS settings.

GCPolicy

v.2.0

Sets the garbage collection policy for Superserver installations. The possible settings are *cooperative*, *background* and *combined*.

This means that, for Superserver, it is possible to configure whether the engine performs garbage collection on databases in separate worker threads (*background*) or by touching every record in the next user transaction involving affected tables (*cooperative*) or by combing both methods (*combined*).

Defaults are *combined* for Superserver and *cooperative* for Classic and Superclassic.



This setting has no effect on a Classic or Superclassic server installation, since Classic supports only cooperative garbage collection.

LegacyHash

v.2.0

This parameter enables you to configure Firebird 2 and above to reject an old DES hash always in an upgraded security database. If you don't use the security database upgrade procedure, this parameter does not affect Firebird operation.

A DES hash cannot arrive in `security2.fdb`.

Refer to Chapter 5, **Migration Notes**, for instructions on upgrading your existing Firebird 1.5 `security.fdb` (or a renamed `isc4.gdb`) to the security database layout for Firebird 2 and higher servers.

The default value is 1 (true), i.e., always reject the old DES hash.

LockAcquireSpins

v.1.5

v.1.0 ***lock_acquire_spins*** in `isc_config/ibconfig`

This setting is relevant only on SMP machines running Classic/Superclassic.

Only one client process may access the lock table at any time. A mutex governs access to the lock table. Client processes may request the mutex conditionally or unconditionally. If it is conditional, the request fails and must be retried. If it is unconditional, the request will wait until it is satisfied. ***LockAcquireSpins*** (integer) establishes the number of attempts that will be made if the mutex request is conditional.

The default is 0 (unconditional). There is no recommended minimum or maximum.

LockHashSlots

v.1.5, enhanced at v.2.1

v.1.0 ***lock_hash_slots*** in `ibconfig/isc_config`

This parameter is used for tuning the lock hash list. Under heavy load, throughput might be improved by raising the number of hash slots, to disperse the list in shorter hash chains. The value is integer; prime number values are recommended.

This parameter and the **LockMemSize** (see the next topic) should be evaluated at the same time, using the *Lock Print* tool. If the lock hash chains are longer than 20 on average, the number of hash slots is too small. If you need to increase the number of hash slots, you should increase the lock table size by the same percentage.

In Firebird 2.1 the default was raised from 101 to 1009.

The previous maximum number of hash slots available was 2048. At v.2.1, it was raised to 65,536. However, because the actual setting should be a prime number, the exact supported maximum is 65,521 (the biggest prime number below 65,536).

The minimum is 101.

LockGrantOrder

v.1.5

v.1.0 **lock_grant_order** in ibconfig/isc_config

When a connection wants to lock an object, it gets a lock request block which specifies the object and the lock level requested. Each locked object has a lock block. Request blocks are connected to those lock blocks, as either requests that have been granted, or pending requests.

In old InterBase versions, locks were granted as soon as they became available, a policy that could reduce the availability of locks and cause “lock request starvation”. Later versions implemented lock allocation on the basis of “first come, first served”, which improved the general efficiency of responses to lock requests.

The **LockGrantOrder** parameter enables “switching off” the newer policy of queuing lock requests and thus emulating the old InterBase behaviour. The default is 1 (keep the queuing policy).

Change it to 0 to emulate the old behaviour but treat it as a deprecated setting and don't try to set it unless you need it for debugging some modification to the database engine..

LockMemSize

v.1.5, enhanced at v.2.1

v.1.0 **lock_mem_size** in ibconfig/isc_config

This integer parameter represents the number of bytes of shared memory allocated to the memory table used by the lock manager.

Classic/Superclassic

For a Classic or Superclassic server, the **LockMemSize** gives the initial allocation, which will grow dynamically until memory is exhausted (“Lock manager is out of room” does not mean somebody went for coffee!) This parameter's value is related to the size of the database cache, since each page will require a separate lock in the table.



When the number of pages of database cache is set too high for the resources available, it often causes problems with the lock memory table.

Superserver

In Superserver, the memory allocated for the lock manager does not grow.

Defaults

- Firebird 1.5 and 2.0 :: The default size on Linux and Solaris is 98,304 bytes (96K) and on Windows, 262,144 (256K).
- Firebird .2.1 onward :: 1 MB on all platforms.

LockSemCount

Not in v.2.5 or newer versions

This parameter does not exist in the Firebird 2.5 or newer configuration options. Its function has been obviated by the revamped lock management in the later engines.

v.1.0 ***any_lock_sem_count*** in `isc_config/ibconfig`

Integer parameter, specifying the number of semaphores available for interprocess communication (IPC) between Classic processes. The default is 32. Set this parameter on a Classic server to raise or lower the number of available semaphores.

LockSignal

Not in v.2.5 or newer versions

This parameter, applicable to the Classic server, does not exist in the Firebird 2.5 or newer configuration options. Its function has been obviated by the revamped lock management in the later engines.

In Classic, when one process holds a lock on a page or other resource and a second process needs access, the second process signals the first. The default signal used (16, integer) depends on the perating system :

- Windows: BLOCKING_SIGNAL 101
- POSIX: BLOCKING_SIGNAL SIGUSR1

LockSignal lets you change the signal used to indicate lock conflicts. An indication for doing so is when another process on the system, using the same signal, is failing to pass signals along or is getting signals it cannot interpret. A solution may be to elect another signal.



Firebird can tolerate “noisy” signals, i.e., those that are in use by more than one service. When it receives a signal, it passes it to other handlers for the same signal. If it does not know what to do with a signal it receives, normally it would merely ignore it.

v.1.0 In `isc_config/ibconfig`, ***any_lock_signal*** and ***v4_lock_signal***

Change the signal used by setting either of these parameters. Setting both is a better choice.

MaxUserTraceLogSize

v.2.5

Stores the maximum total size of the temporary files to be created by a user trace session using the new Trace functions in the Services API. When the log file size reaches this limit, the trace session is suspended automatically until an interactive user service has read and deleted some log files.

The default limit is 10 MB. Use this parameter to raise or lower the maximum total size of the temporary files storing the output.

OldColumnNaming**v.1.5.3**

Allows the server to revert to the column naming behaviour prior to v.1.5.3 whereby, in SELECT expressions, the engine would not attempt to supply run-time identifiers, e.g., CONCATENATION, for derived fields when the developer neglects to provide identifiers.

The installation default is 0 (disabled).



This setting affects all databases on the server and will potentially produce exceptions or unpredicted results where mixed applications are implemented.

OldSetClauseSemantics**v.2.5**

Before Firebird 2.5, the SET clause of the UPDATE statement assigns columns in the user-defined order, with the NEW column values being immediately accessible to the subsequent assignments. This does not conform to the SQL standard, which requires the starting value of the column to persist during execution of the statement.

From v.2.5 forward, only the OLD column values are accessible to any assignment in the SET clause.

As the v.2.5+ semantics could break existing code, setting OldSetClauseSemantics true (i.e., to 1) reverts to the legacy behaviour. The installation default is 0—the server uses the conformant behaviour.

This parameter is provided as a temporary solution to resolve backward compatibility issues. It will be deprecated in future Firebird versions.



Enabling OldSetClauseSemantics affects all databases on your server.

Redirection**v.2.5**

Parameter for controlling redirection of remote requests. It controls the multi-hop capability that was broken in InterBase 6 and is restored in Firebird 2.0.

About Multi-hop

When you attach to some database using multiple hosts in the connection string, only the last host in this list is the one that opens the database. The other hosts act as intermediate gateways on port gds_db (by default, port 3050).

Multihop is potentially a bad security risk. This resurfaced capability is disabled by default. Setting the **Redirection** parameter to 1 enables it.



*If you are considering enabling multi-hop capability, please study the warning topic in Chapter 36, **Protecting the Server and its Environment**.*

RelaxedAliasChecking**v.2.0**

If enabled, permits a slight relaxation of the restrictions on mixing relation aliases and table names in a query. For example, with **RelaxedAliasChecking** set to true (=1), the following query will succeed, whereas it would fail in v.2.0.x, or in v.2.1 with the parameter set to its default of 0:

```
SELECT ATABLE.FIELD1, B.FIELD2
FROM ATABLE A JOIN BTABLE B
```

ON A.ID = B.TABLE.ID

Understand that this is a temporary facility whose purpose is to provide some headspace for migrating systems using legacy code that exploited the tolerance of InterBase and older Firebird server versions to non-standard SQL usage. Expect it to be permanently removed from a future release.



Do not enable this parameter if you have no “offending” code in your applications or PSQL modules.

RelaxedAliasChecking is not applicable to v.1.5.

RemoteAuxPort

v.1.5, enhanced in v.2.5

Passing event notification messages back to the network layer through randomly selected TCP/IP ports shows up in some types of installation as a persistent source of network errors and conflicts with firewalls, sometimes to the extent of causing the server to crash. With RemoteAuxPort you can configure a single TCP Port for all event notification traffic.

The value is an integer. The installation default (0) retains the traditional random port behaviour. To dedicate one specific port for event notifications, use an integer which is an available port number.



In versions prior to v.2.5, assigning a dedicated port for events is available only to Superserver, thus limiting the use of events through a firewall or a secure tunnel to that model. From v.2.5 forward, it works for Classic and Superclassic as well

RemoteBindAddress

v.1.5, enhanced in v.2.5

By default, clients may connect from any network interface through which the host server accepts traffic. This parameter allows you to bind the Firebird service to incoming requests through one single IP address (e.g. network card) and to reject connection requests from any other network interfaces. This helps to solve problems in some networks when the server is hosting multiple subnets.

The value is a string. For versions prior to v.2.5, the value must be in a valid dotted IP format. From v.2.5 forward, it may optionally be the host name of the host on which the server is running.

Default value (not bound) is empty.



Before availing yourself of the option to use the host name in preference to the NIC address, take care that the host name specified is not associated concurrently with more than one IP address, anywhere! In particular, check the etc/hosts file on all nodes in the network, including the host station itself.

RemoteFileOpenAbility

v.1.5 (POSIX), v.2.5 available on Windows

Boolean parameter which, if set True (1), allows the engine to open files which reside on a partition on a drive that is not physically connected to the host machine. In versions prior to Firebird 2.5, it is available only on a POSIX NFS (networked filesystem) mounted partition. From v. 2.5, it is available on Windows shares as well.



RemoteFileOpenAbility is not safe for database files—except possibly a read-only database—because the filesystem is beyond the control of the local system. It should not be enabled for the purpose of opening any read/write database whose survival matters to you.

RemoteFileOpenAbility is NOT intended for enabling two servers to accept connections simultaneously to the same database!

The default is 0 (False, disabled) and you should leave it that way unless you are very clear about its effects.

RemoteFileOpenAbility is intended to allow shadows on mapped locations that have high availability. Although usage of shadows is rare these days, other specific, well-tested, safe purposes might be established for using it. An example given was a database kept under lock-and-key on a USB device that could be plugged in to a diskless workstation for performing an occasional, isolated security task.

RemoteServicePort

v.1.5

The two parameters **RemoteServiceName** and **RemoteServicePort** each provides the ability to override the TCP/IP protocol defaults. This one overrides the TCP/IP port number used to listen for client database connection requests, if one of them differs from the installed default port 3050.

Change one of the entries, not both. The **RemoteServiceName** (below) is checked first for a matching entry in the services file. If there is a match, the port number configured for **RemoteServicePort** is used. If there is not a match, then the installation default, port 3050, is used.



If a port number is provided in the TCP/IP connection string, it will always take precedence over RemoteServicePort.

RemoteServiceName

v.1.5

This is the symbolic name of the service as broadcast by the server. If the `firebird.conf` file is optionally included in a client-only installation (see [xxx]), the client will use it to find the service name if necessary. See also **RemoteServicePort** (above). For more information, refer to the topic **Configuring the Port Service** in the previous chapter.

Default = `gds_db`

RootDirectory

v.1.5

This value is a string, the absolute path to a directory root on the local filesystem. It should remain commented unless you want to force the startup procedure to override the path to the root directory of the Firebird server installation, that it would otherwise detect for itself. Since Firebird 1.5, servers follow a predefined route to find objects that they expect to be the root directory. The logic of this route is explained in the previous chapter, **Configuring Firebird**.

TcpRemoteBufferSize

v.1.5

The engine reads ahead of the client and can send several rows of data in a single packet. The larger the packet size, the more rows are sent per transfer. Use this parameter—with caution and complete comprehension of its effects on network performance!—if you need to enlarge or reduce the TCP/IP packet size for send and receive buffers. It affects both the client and server.

Value is an integer (size of packet in bytes) in the range 1448 to 32768. The installation default is 8192.

TcpNoNagle

v.1.5

On Linux, by default, the socket library will minimize physical writes by buffering writes before actually sending the data, using an internal algorithm (implemented as the

TCP_NODELAY option of the socket connection) known as Nagle's Algorithm. It was designed to avoid problems with small packets, called tinygrams, on slow networks.

By default, TCP_NODELAY is enabled (value 0) when Firebird Superserver is installed on Linux. On slow networks, disabling it can actually improve speed on slow networks. Watch out for the double negative—set the parameter True (=1) to disable TCP_NODELAY and False (0) to enable it.

In releases up to and including v.1.5, this feature is effective only for Superserver. From v.2.0 onward, it is disabled (=1) by default and deprecated.

v.1.0 **tcp_no_nagle** in `isc_config`—was not available on Windows at all.

TempBlockSize

v.2.1 (SortMemBlockSize in v.1.5 and 2.0)

This integer parameter allows you to configure, in bytes, the minimum size of each memory block allocated for use by the in-memory sorting module. The installation default is 1,048,576 bytes (1MB). You can reconfigure it to any size up to the currently configured maximum value set by the **TempCacheLimit** (formerly **SortMemUpperLimit**) parameter (see below).



This value can be regarded as the granularity of memory allocation for sorts.

TempCacheLimit

v.2.1 (SortMemUpperLimit in v.1.5 and 2.0)

The maximum amount of memory, in bytes, to be allocated by the in-memory sorting module. The installation default (integer) is 67,108,864 bytes (64 MB) for SuperServer and 8,388,608 (8 MB) for the Classic and Superclassic servers.



For Classic and Superclassic, the default is too large unless only a handful of clients are connected. Bear in mind that increasing either the block size or the maximum limit on Classic affects each client connection/server instance, or each attachment thread on Superclassic, and will ramp up the server's memory consumption in linear proportions.

TempDirectories

v.1.5

When the size of the internal sort buffer is too small to accommodate the rows involved in a sort operation, Firebird needs to create temporary sort files on the server's filesystem. By default, it will look for the path specified in the environment variable `INTERBASE_TMP`. If that variable is not present, it will try to use the root of the `/tmp` filesystem on Linux/UNIX, or `C:\temp` on Windows NT/2000/XP. None of these locations can be configured for size.

Firebird provides a parameter for configuring the disk space that will be used for storing these sort files and other temporary sets. It is prudent to use it, to ensure that sufficient sort space will be available under all conditions.

All `CONNECT` or `CREATE DATABASE` requests share the same list of temporary file directories and each creates its own temporary files. Sort files are released when the sort is finished or the request is released.

Supply a list of one or more directories, separated by semi-colons (;), under which sort files may be stored. Each item may include an optional size argument, in bytes, to limit its storage. If the argument is omitted, or is invalid, Firebird will use the space in that directory until it is exhausted, before moving on to the next listed directory.

Examples Unix: `/db/sortfiles1 100000000;/firebird/sortfiles2`

Windows: E:\sortfiles 500000000

Relative paths are treated as relative to the path that the running server recognizes as the root directory of the Firebird installation. For example, on Windows, if the root directory is C:\Program Files\Firebird, then the following value will tell the server to store temporary files in C:\Program Files\Firebird\userdata\sortfiles, up to a limit of about 477 Mb:

TempDirectories = userdata\sortfiles 500000000

v.1.0 *tmp_directory* in isc_config/ibconfig

The syntax for the older *tmp_directory* value is to include one *tmp_directory* line for each directory in which you want temporary sort files to be stored. Specify number of bytes available in the directory, and, within double-quotes, the path to a directory which exists on a physical drive with sufficient spare capacity. There is no restriction on the name used for the directory. You can list multiple entries, one per line, and the spaces do not need to be contiguous or confined to a single storage device.

For example, the following entries constitute a list in a single configuration file:

```
tmp_directory 6000000 "d:\fbtemp"
tmp_directory 12000000 "f:\fbtemp"
tmp_directory 4000000 "w:\backwash"
```



For this old syntax, the pathname must be enclosed in double quotes, or the server will ignore the entry. Space will be used according to the order specified. If space runs out in a particular directory, Firebird creates a new temporary file in the next directory from the directory list. If there are no more entries in the directory list, Firebird displays an error message and stops processing the current request.

Settings Applicable to Microsoft Windows

CpuAffinityMask

v.1.5

v.1.0 *cpu_affinity* in isc_config/ibconfig,

With Superserver on Windows there can be a problem with the operating system continually swapping the entire SuperServer process back and forth between processors on SMP machines. In support lists, this is referred to as “the see-saw effect” and, on affected systems, it can have a severe effect on performance. This parameter must be used to set Firebird SuperServer’s processor affinity to one specific CPU.

CpuAffinityMask and cpu_affinity take one integer, the CPU mask. For example:

```
CpuAffinityMask = 1
cpu_affinity = 1
```

only runs on the first CPU (CPU 0).

```
CpuAffinityMask = 2
cpu_affinity = 2
```

only runs on the second CPU (CPU 1).

```
CpuAffinityMask = 3
cpu_affinity = 3
```

runs on both first and second CPU (not recommended).



Older versions of Firebird can run on Windows9x or ME. On those platforms, this parameter has no effect, as it uses an NT API call. Windows9x flavors do not utilize multiple processors.

Calculating the affinity mask value

You can use this flag to set Firebird's affinity to any single processor or (on Classic server) any combination of the CPUs installed in the system.

Consider the CPUs as a set numbered from 0 to *n*-1, where *n* is the number of processors installed. Let *i* be the zero-based position of a CPU in the set.

To get a mask value *M* for a CPU that you want to be affinity, raise 2 to the power of *i*. If there is more than one CPU to be affinity, add the *M* values together. The affinity mask, *A*, is the sum of the *M* values.

For example, to select the first and fourth processors (processor 0 and processor 3 in the set) calculate as follows:

$$A = 2^0 + 2^3 = 1 + 8 = 9$$



Firebird servers may not support the Hyperthreading feature of some CPU/motherboard setups. To avoid balancing problems, it may be necessary to disable hyperthreading at system BIOS level.

The default CPU affinity mask is 1 (processor 0). On HT systems with a single, physical CPU it not advisable to change the default.

GuardianOption

v.1.5

Boolean parameter used on Windows servers to determine whether the Guardian should restart the server every time it terminates abnormally. The installation default is to do so (1=True).

The alternative (setting 0) is for Guardian to start the server just once, i.e., to disable the Guardian's restart behaviour.

IpcName

v.1.5

Default value: FirebirdIPI

The name of the shared memory area used as a transport channel in local protocol.

v.1.0 The default value—FirebirdIPI—is not compatible with Firebird 1.0 nor with InterBase®. Use the value InterBaseIPI, if necessary, to retain compatibility with an existing application that refers to the shared memory (IPC space) by name.

MaxUnflushedWrites

v.1.5

Introduced in v.1.5 to handle a bug in the Windows server operating systems, whereby asynchronous writes were never flushed to disk except when the Firebird server underwent a controlled shutdown. Hence, on 24/7 systems, asynchronous writes were never flushed at all.

This parameter determines how frequently the withheld pages are flushed to disk when Forced Writes are disabled (asynchronous writing is enabled). Its value is an integer which sets the number of pages to be withheld before a flush is flagged to be done next time a transaction commits. Default is 100 in Windows installations and -1 (disabled) in installations for all other platforms.

If the end of the **MaxUnflushedWriteTime** cycle (see below) is reached before the count of withheld pages reaches the MaxUnflushedWrites count, the flush is flagged immediately and the count of withheld pages is reset to zero.

MaxUnflushedWriteTime

v.1.5

This parameter determines the maximum length of time that pages withheld for asynchronous writing are flushed to disk when Forced Writes are disabled (asynchronous writing is enabled). Its value is an integer which sets the interval, in seconds, between the last flush to disk and the setting of a flag to perform a flush next time a transaction commits. Default is 5 seconds. in Windows installations and -1 (disabled) in installations for all other platforms.



Asynchronous writes were never supported in Windows 9x or ME. On non-Windows platforms, the default value of this parameter is (-1), which disables the effect.

ProcessPriorityLevel

v.1.5

Priority level/class for the server process. This parameter replaced the **server_priority_class** parameter of pre-1.5 releases—see below—with a new implementation.

The values are integer, as follows:

- 0 - normal priority,
- positive value - high priority (same as **-B[oostPriority]** switch on instsvc.exe configure and start options)
- negative value - low priority.



All changes to this value should be carefully tested to ensure that they actually cause the engine to be appropriately responsive to requests.

v.1.0 **server_priority_class** in ibconfig

Relevant on Windows NT/2000 only: priority of Firebird service on Windows NT or Windows 2000. 1 = low priority, 2 = high priority. Default 1.

RemotePipeName

v.1.5

String parameter, the name of the pipe used as a transport channel for Windows Named Pipes networking, often referred to as “NetBEUI”. The named pipe is equivalent to a port number for TCP/IP.

The default value—interbas—is compatible with Firebird 1.0 and InterBase®.

UsePriorityScheduler

v.2.0

Setting this parameter to zero disables switching of thread priorities completely. Its default setting is 1 (enabled).

It affects only the Win32 Superserver. If you have problems with computer response time when running Superserver stand-alone on a workstation, setting this parameter to zero to turn off the thread scheduler may be effective in gaining more frequent CPU slices for Superserver threads.

Settings Applicable to POSIX Platforms

BugCheckAbort

v.2.1

Provides the capability to make the POSIX server stop trying to continue operation after a bugcheck and instead, to call `abort()` immediately and dump a core file. Since a bugcheck usually occurs as a result of a problem the server does not recognise, continuing operation with an unresolved problem is not usually possible anyway, and the core dump can provide useful debug information.

In the more recent Linux distributions the default setups no longer dump core automatically when an application crashes. Users often have troubles trying to get them working. Differing rules for Classic and Superserver, combined with a lack of consistency between the OS setup tools from distro to distro, make it difficult to help out with any useful “general rule”.

Code has been added for Classic/Superclassic and Superserver on Linux to bypass these problems and automate generation of a core dump file when an `abort()` on `BUGCHECK` occurs. The Firebird server will make the required `'cwd'` (change working directory) to an appropriate writable location (`/tmp`) and set the core file size limit so that the `'soft'` limit equals the `'hard'` limit.

By default the setting is disabled (0) for production release versions and enabled (1) for debug versions.

Configuring external locations

Having external code and data that are accessed by the server can present a security vulnerability if the server's filesystem is inadequately protected from intruders or is exposed through holes in the network. These external pieces can be made less vulnerable by configuring restrictions on where the Firebird engine may access them. The capability to deny access to unrecognized locations helps in the overall task of securing the filesystem and the network.



Configure few locations, rather than many, to reduce the scope of the engine's search and the degree access control maintenance required.

The Firebird configuration file—discussed earlier in this chapter—provides settings for restricting access to external function libraries, BLOB filter modules and data files linked to tables defined using `CREATE TABLE <table-name> EXTERNAL`.

UdfAccess

v.1.5

This parameter is used to restrict access to external function libraries and BLOB filter modules, perceived as a potential target for malicious intruder attacks. You can elect one of three levels of access to all such modules, to be applied server-wide.

In times gone by, it was regarded as a benefit to be able to store external code modules in multiple filesystem locations. The world has changed. It is now recommended that they be limited to a single tree or, in very exposed situations, disallowed altogether.

UdfAccess may be *None*, *Full* or *Restrict*.

None disallows all use of user-defined external libraries. It is the installation default on some distributions.

Full permits external libraries to be accessed anywhere on the system. When Full access is enabled, the full file path and name must be included in the MODULE_NAME clause of the DECLARE_EXTERNAL_FUNCTION statement that declares the function to the database. It is not recommended at all nowadays.

Restrict (the installation default) restricts the location of callable external libraries to specific filesystem locations. By default, the search will begin in the /UDF directory beneath your Firebird root. To locate external function libraries or BLOB filter modules elsewhere in the local filesystem, supply a list of one or more directory tree-roots, separated by semi-colons (;), within and beneath which these modules may be stored.

For example,

Unix: /db/extern;/mnt/extern

Windows: C:\ExternalModules

Relative paths are treated as relative to the path that the running server recognizes as the root directory of the Firebird installation. For example, on Windows, if the root of the Firebird installation is C:\Program Files\Firebird\Firebird_1_5, then the following value will restrict the server to accessing external files only if they are located in C:\Program Files\Firebird\Firebird_1_5\userdata\extern:

UDFAccess = Restrict userdata\ExternalModules



Unless you have a special reason to locate external code modules in a tree with a custom name, it is suggested that you keep it simple by using the default relative setting, Restrict = UDF.

v.1.0 external_function_directory in isc_config/ibconfig

This parameter can be used in v.1.0.x (Superserver only) to specify an arbitrary number of locations for external function libraries, BLOB filters and/or character set modules. If this configuration parameter does not exist, Firebird checks the sub-directories ..\udf or ..\intl beneath the path that the running server recognizes as the root directory of the Firebird installation.

Examples:

```
external_function_directory <double-quoted directory path>
external_function_directory "/opt/firebird/my_functions"
external_function_directory "/opt/extlibs/lang"
external_function_directory "d:\udfdir"
```

ExternalFileAccess

v.1.5

This parameter provides three levels of security regarding external files accessed from within the database through tables. The value is a string, which may be *None*, *Full* or *Restrict*.

None (the default value) disables any use of external files on your server.

Full permits external files to be accessed anywhere on the system.

Restrict provides the ability to restrict the location of external files for database access to specific path-trees. Supply a list of one or more directory tree-roots, separated by semi-colons (;), within and beneath which external files may be stored.

For example,

Unix: /db/extern;/mnt/extern

Windows: C:\ExternalTables

Relative paths are treated as relative to the path that the running server recognizes as the root directory of the Firebird installation.

For example, on Windows, if the root that the running server recognizes as the root directory of the Firebird installation is C:\Program Files\Firebird, then the following value will restrict the server to accessing external files only if they are located in C:\Program Files\Firebird\userdata\ExternalTables:

```
ExternalFileAccess = Restrict userdata\ExternalTables
```

The following entry on POSIX will restrict access to only files located in or beneath /exportdata or /importdata:

```
ExternalFileAccess = Restrict /exportdata;/importdata
```

For more information about external files, refer to the topic *Using external files as tables* in Chapter 15, *Tables*.



In Firebird 2 and above, the first path cited is used as the default when a new external file is created

V.1.0.x ***external_file_directory*** in `ibconfig`

On Windows only, for concentrating external files into one or more restricted locations. There is no limit to the number of directories that can be in the search list. Make a one-line entry per directory as follows:

```
external_file_directory <double-quoted directory path>
external_file_directory "d:\x-files"
```

Deprecated Settings

OldParameterOrdering

v.1.5

Version 1.5 addressed an old InterBase bug that caused output parameters to be returned to the client with an idiosyncratic ordering in the XSQLDA structure. The bug was of such longevity that many existing applications, drivers and interface components have built-in workarounds to correct the problem on the client side.

Releases 1.5 and later reflect the corrected condition in the API and are installed with `OldParameterOrdering=0 (False)`. In Firebird 1.5, 2.0 and 2.1, set this Boolean parameter True (1) if you need to revert to the old condition for compatibility with existing code.

Unavailable from Firebird 2.5 forward.

server_working_size_max

server_working_size_min

Two old, deprecated memory parameters that were inherited in `ibconfig` for older Firebird releases and were unsupported. They have never been present in `firebird.conf`.

CreateInternalWindow

This was an old option that was intended to be used to run multiple server instances. It has been removed.

DeadThreadsCollection

The `DeadThreadsCollection` parameter was an old `InterBase` parameter that was never implemented effectively. It is silently ignored in `Firebird 2.0`. In `Firebird 2` and above, dead threads are released “on the fly”, making configuration unnecessary.

CHAPTER 35

CONFIGURING AND MANAGING DATABASES

Topics in this chapter are split into two parts: the first concerns reconfiguring attributes of the database, while the second explains how *gfix*—with occasional help from *gbak*—is used for database configuration and housekeeping tasks.

The *gfix* Tool Set

gfix is a client application that provides a collection of tools for managing and configuring databases. Its tools are operated by way of commands issued from your system's command-line shell, either directly, one at a time, or in an automated sequence using a script or batch file. It has no interactive interface.

With *gfix* you can

- initiate a database shutdown to get exclusive access
- take a database off-line and put it back on-line
- perform a sweep
- change the sweep interval
- switch between synchronous (forced) and asynchronous writes
- change a read-write database to read-only and vice-versa
- set the size of the database cache
- find and commit or recover limbo transactions
- mend corrupted databases and data in certain conditions, in combination with *gbak*
- activate and drop shadow databases

Using the Tools

Before starting up the server in order to do extended work locally on a database, you can add the two operating system variables `ISC_USER` and `ISC_PASSWORD` to avoid having to type the SYSDBA or owner user name and password in every command. They are:

```
shell prompt> SET ISC_USER=SYSDBA
shell prompt> SET ISC_PASSWORD=heureuse
```

For security reasons, you should remove these environment variables as soon as you finish your task. It is not recommended that you configure these variables beyond the scope of your current shell or make them permanent on the system.

Exclusive Access

Exclusive access by the SYSDBA or equivalent, or the database Owner, is required for most of the tasks you do with *gfix* because they affect the way the Firebird server behaves with the database. In a development environment, it is simple enough to detach all your connections explicitly and proceed with your task. In a production environment, it will be necessary to put the database off-line to all other users—known as a **shutdown**—before you proceed to log in with that exclusive access.

Because exclusive access is a prerequisite to any database-wide configuration you might do, and shutting down the database is often a prerequisite to obtaining exclusive access, we start a little out of sequence with a management task: using shutdown to get a database off-line and putting it back on-line.

Shutting Down a Database

Uses *gfix -sh* and *-o*

Database shutdown is not the same as shutting down the server. The server stays running when a database is shut down.

A database is implicitly “in a shut-down state” when no connections are active. An explicit shut-down condition can be imposed, using *gfix* with the `-sh[ut]` switch, to enable the privileged user or the database owner to get exclusive access. Once this explicit shutdown state is achieved, it remains until explicitly de-activated by *gfix -o[nline]*. The two operations are referred to as “shutting down a database” and “putting a database on-line”.

The *gfix* `-shut` and `-online` command switches

The syntax pattern for using the *gfix -sh[ut]* and *-o[nline]* switches is:

```
gfix <switch> [<mode>] [<options>] <db_name> [-user <user-name> -pa[ssword] <password>]
<switch> ::= {-sh[ut] | -o[nline]}
<mode> ::= {normal | multi | single | full}
<options> ::= {-force <timeout> | -tran | -attach}
```

The `<db_name>` parameter can be the fully-qualified absolute database path or the database alias. User name and password may be omitted if the environment variables `ISC_USER` and `ISC_PASSWORD` are set or you are logged in to the host as a superuser. They must be supplied if *gfix* is being run remotely or through the local loopback server (localhost).

Database shutdown modes

- The “normal” mode is when the database is on-line, accepting requests for attachments and transactions
- The “multi” mode is when the database is shut down but **privileged users** are permitted to make unlimited new attachments.
- The “single” mode is when the database has been shut down and only one user attachment is permitted.
- The “full” mode is when the database is shut down and no attachments are permitted at all.

Switching from one mode to another

The modes can be switched sequentially, in either direction as long as it is logical, according to this sequence only:

normal \longleftrightarrow multi \longleftrightarrow single \longleftrightarrow full \longleftrightarrow normal

“Multi” is the default target mode when the **-shut** switch is used, while “normal” is the default target mode for **-online**. That means it is not essential to specify the mode when taking an on-line database off-line or returning a database from “full” or “multi” shutdown mode back on-line.

The rule of thumb for a mode change that is “logical” is to use the **-shut** switch to increase the level of protection from other users and the **-online** switch to increase the proximity to “normal mode”. Trying to use the **-shut** switch to bring a database one level “more online” or the **-online** switch to make a database “more protected” results in an illogical sequence that will return an exception.

For example, this attempt to move from exclusive access mode to “multi” mode fails:

```
gfix -shut single -force 0 our_db
```

followed by

```
gfix -shut multi -force 0 our_db
```

This attempt to put an online database into an off-line mode using the **-online** switch fails:

```
gfix -online our_db
```

followed by

```
gfix -online full our_db
```

Failure, too, for this attempt to use the **-online** switch to change the off-line mode from “multi” (implicit, default for **shut -force**) to exclusive single-user mode:

```
gfix -shut -force 0 our_db
```

followed by

```
gfix -online single our_db
```



In versions prior to the “2” series, the “single” and “full” modes are not implemented. This means that, although a properly shut-down database under v.1.5 or v.1.0 will exclude non-privileged users, things can get complicated if the Owner and SYSDBA are both logged in. The onus will be on the privileged administrator doing the shutdown to ensure that s/he is going to be the only user (human or not) logging in while the database is off-line.

Qualifying arguments for `gfix -shut`

The `-shut` switch comes with a choice of three qualifiers to specify the strategy for the shutdown: `-at[tach] n`, `-tr[an] n` and `-force n`. You must use one and only one argument. In each case, *n* sets a timeout period in seconds.

- `-at[tach] n` is used to prevent new database connections. It doesn't force existing connections off but it blocks any new ones. If no processes are still connected when the timeout period of *n seconds* expires, the database will be in the shutdown state. If there are still processes connected, the shutdown is cancelled.
- `-tr[an] n` is used to prevent new transactions from starting. It doesn't forcibly end existing transactions but it disallows any new ones from being started. If no processes are connected when the timeout period of *n seconds* expires, the database will be in the shutdown state. If there are still transactions active, the shutdown is cancelled.
- `-force n` will force the database into a shutdown state *after n seconds*, regardless of any connected processes or active transactions. This is a drastic operation that could cause users to lose their work. It should be used with caution.



If you need to resort to the `-f[orce]` switch to kill a rogue query, at least be kind to your well-behaved users and use `-at[tach]` or `-tr[an]` first to give them the opportunity to save work and exit gracefully from their applications.

Examples

```
gfix -sh -at 300 localhost:service_mgr our_db
```

initiates a shutdown into “multi” mode that will take effect in five minutes, if all users detach from the database.

```
gfix -sh -force 600 our_db
```

will force all non-privileged users off the system in 10 minutes. Any transactions still running will be rolled back and users will lose their uncommitted work.

```
gfix -shut single -force 0 our_db
```

attempts to get exclusive access while the database is off-line in “multi” mode. It will return an error if another privileged user is already in the process of securing exclusive access.

```
gfix -shut full -force 0 our_db
```

works the same, except that no users, not even the calling superuser, will be attached if the shut-down succeeds. That user can then attempt to get exclusive access using

```
gfix -online single our_db
```

The command to put the database back on-line for all users is:

```
gfix -online our_db
```



You can also use `gfix -online` to cancel a scheduled shutdown, i.e. one that has not completed its timeout.

Exclusive access

From the “2” series forward, once the database is off-line, a privileged user or the Owner can attempt to attach and switch the mode to “single” to obtain immediate exclusive access:

```
gfix -shut single -force 0 our_db
```


This call should bump off any other privileged users that are still logged in. However, if another privileged user already has the database in “single” mode, the call will fail.

Firebird 1.X

Because the older versions of *gfix* do not support the “single” mode, actually getting exclusive access may need cooperation from others if you have more than one human or a scheduled job that runs with SYSDBA or Owner privileges. Note the following:

- if either the owner or SYSDBA was already logged in when the shutdown took effect, the server will not block the other from logging in once the shutdown is in effect
- once either SYSDBA or the Owner logs in after the shutdown, the other will be blocked from logging in. That’s good. If the *same* user wants to log in again, it will be permitted. That’s not so good. Remember, we are talking about a user here, not necessarily a human. Given the carelessness about the use of SYSDBA and other privileged logins on some systems, we are not necessarily talking about just one superuser, either!

The onus on the SYSDBA or Owner user who needs exclusive access to ensure that either itself or the other is not logged in somewhere, using a visual admin tool, an SQL monitor, another command-line tool, a cron or batch job or even another *gfix* option, for example. Once you get exclusive access, keep it exclusive: don't start up more than one application.



On Windows, privileged users other than SYSDBA and Owner are not relevant in Firebird 1.X or 2.0.x.

Server shutdowns and restarts

Whenever you need to shut down the server in a production environment, you will normally want to use **gfix -shut** to shut down individual databases in a controlled way first.

Be aware that shutting down or restarting the server does not have any effect on the on-line/off-line mode of any databases. If a database is in an off-line mode when the server is stopped, it will still be in that mode when the server is next started.



If you file-copy a database that is in an off-line mode, the copy will be off-line when you try to connect to it.

Configuration Options

A number of *gfix* command options permit certain configuration settings to be set or changed for a database. To perform these commands, SYSDBA or owner privileges and exclusive access are required.

Default Cache Size

Uses *gfix -b*

Using **gfix -buffers** is the preferred way to set the default cache size for an individual database. Some points important to remember are:

- if you increase the page size, the cache size will rise accordingly. You should ensure that you take the amount of physical RAM on the machine into consideration when altering cache size. Once the cache reaches the point where it is too large to be kept in RAM, it

will begin swapping out to disk, thereby quite defeating the benefit of having a cache at all.

- on a Classic or Superclassic server, every client gets its own cache. Even the default cache size of 75 pages will be too large on some under-resourced systems if the database is using a big `page_size`.
- Superserver uses one cache for all users. For 32-bit systems, where the total RAM available to a process is limited to 2 GB, the default cache size of 2048 pages for the default `page_size` of 4096 bytes is a recommended starting point and 40-50 Mb should be considered a practical limit.

Syntax for setting the database cache size

```
gfix -b[uffers] n db_name
```

where *n* is the number of page-sized buffers to be reserved in RAM for each cache.

Example

```
gfix -b 5000 d:\data\accounts.fdb
```

If the database `page_size` is 8192, a cache of 5000 pages will allocate a cache of around 40 Mb.

Setting cache at server level instead

The cache size can alternatively be set at server level using the configuration parameter *DefaultDbCachePages* in the previous chapter. If you decide to use this method of reserving cache, use `gfix -buffers` to set the database-level cache to zero, e.g,

```
gfix -b 0 d:\data\accounts.fdb
```

Other cache options

gfix has a switch `-c[ache] n` that is currently unused, reserved for future implementation.

Forced Writes

Uses `gfix -w`

Forced Writes is synonymous with synchronous writes. When the behaviour is synchronous (“Forced Writes enabled”), new records, new record versions and deletions are physically written to disk immediately upon success of the request¹. With asynchronous writes (“Forced Writes disabled”), new and changed data are retained in the system file cache, relying on the flushing behavior of the operating system to make them permanent on disk.

The term “disabling Forced Writes” means switching the write behavior from synchronous to asynchronous.

The command syntax pattern is

```
gfix -w[rite] {sync | async}
```

Enabling and Disabling Forced Writes

To enable Forced Writes, use the `-write` switch with the `sync` argument:

```
gfix -w sync d:\data\accounts.fdb
```

```
gfix -w sync /usr/data/accounts.fdb
```

1. In Firebird, new record versions and delete stubs are stored in the database, even though they are not committed until the transaction commits. This step, which occurs at statement level, is often referred to as “posting”.

To disable, use the `async` argument:

```
gfix -w async d:\data\accounts.fdb
gfix -w async /usr/data/accounts.fdb
```

Firebird is installed on Windows and Linux with Forced Writes enabled. In a very robust environment with highly reliable UPS support, a DBA may disable Forced Writes to reduce I/O and improve performance. When Forced Writes is disabled in less dependable environments, the database becomes susceptible to data loss and even corruption in the event of an uncontrolled shutdown.

Force Writes on Linux servers

A Firebird core developer discovered, quite late in the game, that Forced Writes could never have actually worked on Linux. It had been implemented using a call that Linux kernel developers had documented but had forgotten to implement.

The mechanism was recoded in Firebird and is fixed from v.2.0.4 forward.

Forced Writes on Windows servers

Windows is less dependable than other operating systems with regard to cache flushing. It appears that, if applications do not explicitly request the Windows system to flush the cache, it may defer all writes until the database file is closed. In Firebird 1.0.x, if Forced Writes is disabled on a 24/7 Windows server, flushing may never occur.

For using a database with forced writes off, there are a couple of parameters in `firebird.conf` to specify a “flush cycle” that at least ensures things are written to disk with a predictable regularity. Refer to the parameters *MaxUnflushedWrites* and *MaxUnflushedWriteTime* in the previous chapter.

Very old Windows platforms

Forced writes are not applicable to Windows 95. On Windows 98 and ME servers you should never disable forced writes.

SystemRestore Feature on Windows

Windows releases and editions since Windows ME and XP have a feature named System Restore, which causes the operating system to update its own filesystem backup of files with certain suffixes each time a file I/O operation occurs. System Restore is not a substitute for forced writes.

Converted InterBase 6 databases

Be aware that a Firebird database that started life in InterBase® 6.x (commercial or Open Edition) or earlier will have been created with Forced Writes disabled by default.

Access Mode

Uses `gfix -mo`

Use the `gfix -mo[de]` option to switch the access mode, for all connections to the database, between read-only and read-write. A read-only database can not be written to at all—not even by SYSDBA, its Owner or any server process.

Syntax pattern for setting the access mode

```
gfix -mo[de] {read_write | read_only} db_name
```

To switch a database from read-write to read-only:

```
./gfix -mo read_only /data/accounts.fdb
```

To switch from read-only to read-write:

```
./gfix -mo read_write /data/accounts.fdb
```

Page Fill Capacity

Uses **gfix -u**

Firebird fills database pages so that the ratio of data stored per page does not exceed 80 per cent. The purpose is to enhance the availability of space for new record versions on existing pages by keeping some in reserve to compensate for the variations in actual space used by and released from obsolete record versions. Over time, this strategy helps to level out the number of pages used to store a table's data. In the long view, the fewer pages the engine has to visit for reads and writes, the better the I/O performance.

It makes sense to fill pages to full capacity in a database that you plan to distribute as a read-only database, as a catalog or a demonstration, for example.

The database needs to be in read-write mode for this command to work. The command switch is `-u[se]` and it takes one of two arguments:

```
gfix -u[se] {reserve | full}
```

- `reserve` ('reserve some space') sets the page use to 80 per cent
- `full` ('use all space') sets the page use to 100 per cent

To enable 'use all space' use the command

```
./gfix -use full /demos/catalog.fdb
```

To disable 'use all space' and return to the 80 per cent fill ratio, set the access mode to read-write if it is read-only and use the command

```
./gfix -use reserve /demos/catalog.fdb
```

Sweep Interval

Uses **gfix -h**

Sweep interval is an integer setting in the database that specifies the threshold for a particular set of conditions that will trigger off an automatic *sweep*, a process that can deal with the obsolete record versions left behind by updates and deletes, commonly referred to as "garbage" or "back versions". For more detailed information about garbage and the ways Firebird deals with it, see the topics [Garbage collection](#) and [Sweeping](#), later in this chapter.

Databases are created with a default sweep interval of 20,000. It represents a theoretical high limit of transaction throughput at which the engine would signal the need for an automatic sweep to attempt reducing it.

It is a subtle but important distinction that the automatic sweep does not occur every 20,000 transactions. Garbage collection ("GC") goes on behind the scenes all the time: back versions that are not interesting to any active transactions are cleaned up behind the scenes by another process. Garbage build-up comes into the orbit of the automatic sweep when the regular GC cannot cope with the volume of build-up and the engine detects a large enough "gap" between the oldest transaction whose garbage is still "interesting" and the newest transaction that is still active. The automatic sweep will be flagged if this "gap" becomes equal to or greater than the sweep interval.

Changing the sweep interval

Usually, raising or lowering the sweep interval has little or no effect on performance. However, if your applications frequently roll back transactions, you might encounter extraordinary build-ups of garbage that GC cannot remove. If you are seeing an increase in transaction start-up times between sweeps, lowering the sweep interval might help to reduce the build-up of rollback artifacts.

If the sweep interval is too low, application performance might tend to go down because of too-frequent sweeping. Raising the sweep interval could help improve overall performance in this case.

The option switch for setting the sweep interval is `-h[ousekeeping] n`, where *n* represents the count (interval) that you want to change to:

```
gfix -h 10000 /data/accounts.fdb -user SYSDBA -pas heureux
```

sets the new sweep interval for `accounts.fdb` to 10000.

Disabling automatic sweeping

You might consider disabling the automatic sweep if the occasional, unpredictable delays imposed by automatic sweeps are causing unacceptable slow-downs for users. Disabling it is not recommended unless back version housekeeping is being managed effectively by alternative means, such as monitoring statistics and running regular manual sweeps and/or *gbak* backups.

Automatic sweeping can be disabled by setting a sweep interval of zero.

```
./gfix -h 0 /data/accounts.fdb -user SYSDBA -pas heureux
```

or (Windows)

```
gfix -h 0 d:\data\accounts.fdb -user SYSDBA -pas heureux
```

sets the sweep interval to zero, thus disabling automatic sweeping.

Sweeping is discussed in more detail in the [Management Tools](#) section below.

SQL Dialect

Uses **gfix -sql** and other tools

With this switch, you can change a dialect 1 database to Firebird's native dialect 3 format. The database then stops being dialect 1 and obeys the full syntax rules of Firebird SQL and can accept all of Firebird's data types.

BUT (aren't there always some of those?) the database retains any existing data and definitions in accordance with dialect 1. Some traps lie here, especially with respect to fixed numeric types.

In short, changing the dialect with *gfix* is neither a quick-fix migration tactic nor the most bomb-proof way to migrate from dialect 1 to dialect 3. A quick change with *gfix* means a slow route to a fully-migrated database. Experienced users recommend, instead, to extract a schema script from your dialect 1 database, modify the definitions to suit your needs and then to use a datapump tool to move the old data across.

Still, if you want to do it this way anyway, here's how.

Syntax **gfix -sql[_dialect] n db_name**

where *n* must be either 1 or 3. In reality, of course, it should be 3—there is no reason to change a database from dialect 3 to dialect 1 and it's highly likely such a move has never been tested to see what it might break.

Example

To change the dialect of the database to 3:

```
./gfix -s 3 /data/accounts.fdb
```

or, on Windows, you might do:

```
gfix -sql_dialect 3 d:\data\accounts.fdb
```

For a detailed description of the dialect 1 issues and how to convert the data in a dialect 1 database to dialect 3, see the topic *Dialect 1 Databases* in Chapter 5, **Migration Notes**.

You might also find the original InterBase 6.0 Beta **Get Started** guide useful for extra information about the dialect differences, although the migration technique described there is not recommended, as it was never fully implemented. Archived copies of this document can still be found by searching the web, for example at

<http://www.ibphoenix.com/files/60GetStart.zip>.

Page Size

Uses `gbak -create`

The page size of a database is set initially by the optional `page_size` parameter in the CREATE DATABASE statement. It is the size of a chunk of hard disk space that the server requests to be allocated by the operating system whenever more space is needed. If the `page_size` parameter is not used in the CREATE statement, the default page size is 4 KB (or 2KB and 1KB in versions 1.5 and 1.0, respectively).



From v.2.5 forward, that 4KB default is the minimum page size that can be specified for a new database. In older versions, it is possible—albeit probably not sensible—to create databases with page sizes of 1KB or 2KB. Firebird 2.5+ servers can read database with those page sizes.

If you need to configure a database to have a larger page size, there is one and only one way to do it: by backing up with *gbak* and then restoring with the new page size specified in the optional `page_size` parameter.

Restoring a Database with a New Page Size

Suppose you have a database backup file named `my_db.fbk` in firebird's home directory. The database header information stored in that backup records that the page size of the database should be 4KB. You want it to be 8KB. Following the recommended safe practice for restores, you would restore it in a manner similar to the following.

In `aliases.conf`, in the Firebird root directory, create an alias for the restored database:

```
my_db_test = /home/firebird/Documents/my_db.fdb
```

Save `aliases.conf`.

If you are not sure which user is the Owner of the database, look it up in `RDB$DATABASE` and make note of it. You will need the password of that user for this task. Let's suppose the user is `SYSADMIN` and the password is `'rhubarb'`.

Now, proceed with the restore, entering the whole of this command in one line:

```
/bin/gbak -c /home/firebird/my_db.fbk my_db_test
-page_size 8192 -user SYSDBA -password rhubarb
```

That's all. You can check the results afterwards using `gstat -h` or the SHOW DATABASE command in interactive *isql*. If all is well, take the old database off-line, rename it, copy the newly created `my_db.fdb` to your active database directory. If necessary, use `gfix -online` to put it on-line.

Management Tools

At the beginning of the chapter we discussed how to put a database off-line for maintenance tasks and how to get it back on-line again. In this section we look at the other database management tools available in the *gfix* toolset—with help sometimes from *gbak*.

Garbage collection

The Firebird server maintains an inventory of transactions. Any transaction that is present in the inventory in any state except *committed* is known as an *interesting transaction*. The oldest of these “interesting” transactions (Oldest Interesting Transaction—OIT) marks the starting point for a condition known as “the gap”.

The opposite end of the “gap” is the oldest transaction that was still active (Oldest Active Transaction, OAT) when the previous sweep completed: it shows in the ***gstat -h*** output as “Oldest Snapshot” (OST). The “gap” that the engine detects is thus the difference between the OIT and the OST. When the size of the “gap” reaches the number specified as the sweep interval, a flag is set to make an automatic sweep occur next time a new transaction starts.



*The **gstat** tool is discussed in [Collecting Database Statistics—gstat](#) in Chapter 38, [Monitoring and Logging Features](#).*

Firebird performs garbage collection—“GC”—automatically, to limit the database growth from obsolete record versions. GC frees up space allocated to an outdated version of a row as soon as possible after that row version becomes uninteresting to any transactions that involved it. Transactions kick off GC when they encounter back versions of rows discarded by other transactions. Deleted rows and abandoned versions left after rollbacks escape this garbage collection. Rows that are infrequently touched will cause back versions to accumulate, too.

Garbage collection also happens whenever the database is backed up using *gbak*, since *gbak*'s data-copying task touches every row in every table. However, *gbak* doesn't perform a full sweep. Like the regular GC, it leaves deleted and rolled-back back versions alone. Sweeping is the only way to get rid of these, short of restoring the database from a backup.

Sweeping

Uses *gfix -sweep*

Firebird's multi-generational architecture creates the situation where multiple versions of data rows are stored directly on the data pages. Firebird keeps the old versions when a row is updated or deleted. In the normal course of events, obsolete record versions created by updates are cleaned up by cooperative or background garbage collection.

However, under some conditions, these old versions can get “stuck” and accumulate, causing the database file(s) to grow out of proportion to the size of accessible data. Sometimes, having “stuck transactions” will impact performance.

Sweeping is a systematic way to remove outdated rows from the database and prevent it from growing too large. By default, Firebird databases are always set up to be swept automatically when certain conditions occur. However, because performance can be

affected during a sweep, sweeping can be tuned to optimize its benefits while minimizing its impact on users.

It can be a positive tuning strategy to disable automatic sweeping and take charge of it yourself. You can monitor the database statistics and perform manual sweeps, either on an “as required” basis or at scheduled times. You can, for example, include a sweep command in a *cron* script or scheduled batch file.

For information about how database statistics reports can help in the analysis of the sweeping requirements in your database, refer to the section Collecting Database Statistics—*gstat* in Chapter 38.

Performing a manual sweep

A manual sweep can be done at any time to release space held by back versions, especially record versions left behind by rollbacks and deletions. It is common to schedule sweeps at times of low activity on the database server, to avoid competing with clients for resources.

You may wish to do your own sweeping if

- you are monitoring the “gap” and want to choose the appropriate time to housekeep the sticky back versions
- you think occasional updates of infrequently-visited records might have built up a backlog of uncollected garbage
- a large run of deletions has been done and you want to shift the garbage promptly

To start an immediate sweep:

```
gfix -sweep C:\data\accounts.fdb -user SYSDBA -pas masterkey
```

or (POSIX):

```
./gfix -sweep /data/accounts.fdb -user SYSDBA -pas masterkey
```

Exclusive access for manual sweeps

Sweeping a database does not strictly require it to be shut down—it can be done at any time—but it can impact system performance and should be avoided at busy times.

There is a benefit if a sweep is performed with exclusive access and with all clients’ work committed. Under these conditions, not only is more memory available to the sweep operation, but the sweep is able to do a thorough cleanup of the states of data records and transaction inventory. Unresolved transactions are finally rendered obsolete and the resources being used to track them are freed.

Analysing and Repairing Logical Corruption Uses gfix

In day-to-day operation, a database is sometimes subjected to events that pose minor problems to database structures. These events include:

- Abnormal termination of the server
- The integrity of the database is not affected by an abnormal termination. However, if Firebird has already assigned a data page for uncommitted changes requested by clients, the page becomes “an orphan”. While orphan pages are quite benign, they occupy unassigned disk space that should be returned to free space. Validation can find and release these spaces.
- Write errors in the operating system or hardware

Write errors usually create problems for database integrity. They can cause data structures such as database pages and indexes to become broken or lost. At worst, these corrupt data structures can make committed data irrecoverable. Sometimes, validation may be able to help find these broken pieces and eliminate them.

When to validate a database

You should validate a database:

- whenever a *gbak* backup is unsuccessful.
- whenever an application receives a “corrupt database” error.
- periodically, as a regular housekeeping routine, to monitor for corrupt data structures or misallocated space.
- any time you suspect data corruption.

The command-line utility *gbak* can be used in conjunction with *gfix* to perform a sequence of validation and repair steps.

Performing a database validation

Database validation requires exclusive access to the database. Without exclusive access, you get the error message:

```
OBJECT database_name IS IN USE
```

To validate a database, simply enter the command:

```
gfix -v db_alias_name -user SYSDBA -password heureux
```

Validation will silently locate and free any unassigned pages or misallocated structures it finds. It will report any corrupt structures but does not attempt to mend them. To have *gfix* report faults but no attempt to free the spaces, include the `-n[o_update]` switch:

```
gfix -v -n etc.
```

You can have the validation ignore checksum errors by adding the `-i[gnore]` switch:

```
gfix -v -i etc.
```



Even if you can restore a mended database that reported checksum errors, the extent of data loss may be difficult to determine. If this is a concern, you may want to locate an earlier backup copy and restore the database from it.

Repairing a corrupt database

If you suspect you have a corrupt database, it is important to follow a proper sequence of recovery steps in order to avoid further corruption. For a detailed description of the recommended recovery procedure, see Appendix IX, **Database Repair How-to**.

For a description of the *gfix* switches use during this recovery procedure, refer to Table 35.1, Summary of *gfix* switches at the end of the chapter.

Transaction recovery

Uses *gfix* -l, -p, -t, -c, -r

gfix provides tools for recovering transactions left in limbo after a connection is lost during a multi-database transaction.

Two-phase commit

A transaction that spans multiple Firebird databases is committed in two steps, or phases. This two-phase commit guarantees that, if the transaction cannot complete the updates to all of the databases involved, it will not update any of them.

In the first phase of a two-phase commit, Firebird prepares a sub-transaction for each database involved in the transaction, and writes the appropriate changes to each database.

In the second phase, following the same order in which it prepared and wrote them, Firebird marks each sub-transaction as committed.

Limbo transactions

Limbo transactions are sub-transactions that remain unresolved if something traumatic happens to one or more database connections during the second phase of the two-phase commit: for example, a network fault or a power failure. All of the transactions involved in the multi-database transaction are “in limbo” because the respective servers cannot tell whether they should be committed or rolled back.

Consequently, some records in the databases involved may become inaccessible until explicit action is taken to resolve the limbo transactions with which they are associated.

Transaction recovery

With *gfix*, you have a number of options for inquiring in the databases about and resolving limbo transactions after the traumatic failure of a two-phase commit. The process of identifying a limbo transaction and either committing it or rolling it back is known as transaction recovery.

You can attempt to recover all limbo transactions or you can perform the recovery, transaction by transaction, using the transaction ID of each individual transaction.



These tools will not recover single-database transactions that were left incomplete by some external event. They work strictly on two-phase transactions.

Finding limbo transactions

To list the IDs of all limbo transactions, along with an indication of what would happen to each if an automatic two-phase recovery were requested, use the `-l [ist]` switch (that's “I” as in “list”):

```
gfix -l db_name
```

Prompting for recovery

Use the `-p[rompt]` switch together with `-l [ist]` to have *gfix* list the limbo transactions one by one and prompt you for COMMIT or ROLLBACK action:

```
gfix -l -p db_name
```

Automated two-phase recovery

Since limbo transactions result from either uncompleted commits or uncompleted rollbacks, the server knows how each should end. Hence, automatic recovery is merely a way of confirming that you want *gfix* to tell the server to proceed with the original intentions as they stood when the two-phase commit was interrupted.

The `-t[wo_phase] {ID | all}` switch initiates an automated two-phase recovery.

Use the **all** option to perform a two-phase recovery for all limbo transactions:

```
gfix -t all db_name
```

Use the **ID** option by entering the transaction ID of a single transaction for which you want a two-phase recovery performed:

```
gfix -t nnnnnn db_name
```

where *nnnnnn* is the ID of the targeted transaction.

Specifically committing or rolling back

To attempt to resolve limbo transactions by committing them, use the `-c[ommit]` {ID | all} switch. To recover **all** limbo transactions in this manner, enter:

```
gfix -c all db_name
```

To resolve a **specific** limbo transaction by attempting to commit it, enter:

```
gfix -c nnnnnn db_name
```

where *nnnnnn* is the ID of the targeted transaction.

To attempt to resolve limbo transactions by rolling them back, use the `-r[ollback]` {ID | all} switch.

To resolve **all** limbo transactions in this manner, enter:

```
gfix -r all db_name
```

To resolve a **specific** limbo transaction by attempting to roll it back, enter:

```
gfix -r nnnnnn db_name
```

where *nnnnnn* is the ID of the targeted transaction.



In case you missed it, you can use the `-list` switch to find out the IDs of the limbo transactions.

Managing Database Shadows

Creation of shadows is performed through DDL statements. However, a dead database cannot talk DDL. *gfix* has the utilities for operating on the shadow, to activate it as a replica of the database it has been shadowing and to kill it when it becomes unavailable following the demise of that database.

The concept and procedure of shadowing are discussed in [Database Shadowing](#) in Chapter 39, *Backing Up Databases*.

Activating a shadow

The *gfix* switch for activating a shadow when a database dies is `-ac[tivate]`. Syntax is:

```
gfix -ac <path-to-first-shadow-volume>
```

Suppose the shadow's first volume, `employee.sh1`, is in a directory `/opt/dbshadows`, you would activate it with this command:

```
./gfix -ac /opt/dbshadows/employee.sh1
```

A database shadow “knows” the name and location of the database it has been shadowing and, in the case of multi-file shadows, how to find the succeeding volumes. In this example, our activated `employee.fdb` and any secondary files will appear where the shadowed database was.



Make sure you move or rename the “dead” database files before you activate the shadow.

Dropping unavailable shadows

Once a shadow has been activated, it is no longer a valid shadow and it becomes unavailable for further shadowing.

The switch for dropping unavailable shadows is `-k[i11]`. Syntax is:

```
gfix -k[i11] db_name
```

To kill unavailable shadows for `employee.fdb`:

```
./gfix -k /opt/firebird/examples/empbuild/employee.fdb
```



The server will not create new shadows for a database that has been reconstituted from a shadow.

Summary of *gfix* switches and options

Table 35.1 Summary of *gfix* switches

Switch/Option	Task	Purpose
<code>-ac[tivate] shadow-file</code>	Shadowing	Used with the primary shadow file path, to activate a shadow
<code>-at[tach] n</code>	Shutdown	Used with <code>-shut</code> to prevent new database connections during timeout period of <i>n</i> seconds. Shutdown will be cancelled if there are still active connections after <i>n</i> seconds.
<code>-b[uffers] n</code>	Database page cache	Set default database cache buffers for the database to <i>n</i> pages. This is the recommended way to set the default database cache size.
<code>-ca[che] n</code>	Not used	
<code>-c[ommit] {ID all}</code>	Transaction recovery	Commit limbo transaction specified by ID or commit all limbo transactions.
<code>-fo[rce] n</code>	Shutdown	Used with <code>-shut</code> to force shutdown of a database after <i>n</i> seconds—a drastic solution that should be used only as a last resort.
<code>-full</code>	Data repair	Used with <code>-v[alidate]</code> to check record and page structures, causing unassigned record fragments to be released.
<code>-h[ousekeeping]</code>	Control auto sweeping	Change threshold for automatic sweeping to <i>n</i> transactions. Default is 20,000. Set <i>n</i> to 0 to disable automatic sweeping.
<code>-i[gnore]</code>	Data repair	Ignore checksum errors when validating or sweeping.

Switch/Option	Task	Purpose
-k[ill] db_name	Shadowing	Used with the database file path, to kill any unavailable shadows
-l[ist]	Transaction recovery	Display IDs of each limbo transaction and indicate what would occur if -t[two_phase] were used for automated two-phase recovery.
-m[end]	Data repair	Mark corrupt records as unavailable, so that they will be skipped during a subsequent validation or backup
-n[o_update]	Data repair	Used with -v[alidate] to validate corrupt or misallocated structures, reporting them but not fixing them.
-o[nline]	Shutdown	Cancels a -shut operation that has been scheduled, or rescinds a shutdown that is currently in effect.
-pa[ssword] password	Remote access	Submits password for accessing database. For most gfix operations, this must be the password of the SYSDBA, the database owner or (on POSIX) a user with root privileges.
-p[rompt]	Transaction recovery	Used with -l[ist] to prompt for action during transaction recovery
-r[ollback] {ID all}	Transaction recovery	Roll back limbo transaction specified by ID or roll back all limbo transactions.
-s[weep]	Housekeeping	Force an immediate sweep of the database.
-sh[ut]	Shutdown	Shut down the database. Requires to be qualified by either -at[ach], -f[orce] or -tr[an] n and database path or alias.
-sql[_dialect] n	Migration	n=3. Changes SQL dialect of the database from 1 to 3. It does not change data or convert existing data types.
-t[two_phase] {ID all}	Transaction recovery	Perform automated two-phase recovery, either for a limbo transaction specified by ID or for all limbo transactions.
-tr[an] n	Shutdown	Used with -shut to prevent new transactions from starting during timeout period of n seconds. Shutdown will be cancelled if there are still active transactions after n seconds.
-use {reserve full}	Page fill capacity	Enable or disable full use of the space available on database pages. The default reserve fills pages using a fill ratio of 80 percent. Switching to full uses all space available.
-user username	Remote access	Submits username for accessing database. For most gfix operations, this must be SYSDBA, the database owner or (on POSIX) a user with root privileges.

Switch/Option	Task	Purpose
-v[alidate]	Data repair	Locates and releases pages that are allocated but unassigned to any data structures. Also reports corrupt structures.
-w[rite] {sync async}	Forced writes	Enable or disable forced (synchronous, buffered) writes. sync enables, async disables.
-z	Information	Report version of gfix and Firebird server.

gfix error messages

Table 35.2 *gfix* error messages

Error Message	Causes and Suggested Actions to Take
Database file name <string> already given	A command-line option was interpreted as a database file because the option was not preceded by a hyphen (-) or slash (/). Correct the syntax.
Invalid switch	A command-line switch or option was not recognized.
Incompatible switch combinations	You specified at least two options that do not work together, or you specified an option that has no meaning without another option (for example, -full on its own).
More limbo transactions than fit. Try again.	The database contains more limbo transactions than gfix can print in a single session. Commit or roll back some of the limbo transactions, then try again.
Numeric value required	The -housekeeping option requires a single, non-negative argument specifying number of transactions per sweep. That argument does not have a default.
Please retry, specifying <string>	Both a file name and at least one option must be specified.
Transaction number or “all” required	You specified -commit, -rollback, or -two_phase without supplying the required argument.
-mode read_only or read_write	The -mode option takes either read_only or read_write as an argument. Check spelling.
“read_only” or “read_write” required	The -mode option must be accompanied by one of these two arguments.



The
Firebird Book
A Reference for Database Developers

SECOND EDITION

PART VIII



Administering & Securing Firebird

CHAPTER 36

PROTECTING THE SERVER AND ITS ENVIRONMENT

Other than user passwords, Firebird has no facility to encrypt and decrypt data that pass through its client interface. It does provide certain restrictions on the use of Firebird tools that access databases but, ultimately, at the software level alone, no system is safe from a raider who is determined to access your databases without authority.

Securing the Environment

In this section we draw your attention to some areas where you need to take precautions with your Firebird server and client environments. It is in no sense a blueprint for solving all of the environmental security issues that might affect your server and your network. In short, if security is a serious concern for your deployments, then treat it seriously. Research it, recognize potential risk areas and be prepared to consult specialists.

Physical Security

The overriding factor in physical security is to protect these security-sensitive machines from any physical contact by unauthorized hands. All else is to no avail if someone can get to the machine and steal it; or someone can open the door of the rack-room and steal the hard drives attached to the server.

Keep servers and sensitive or critical client machines behind well-locked doors. If you have FAT32 partitions on servers or workstations, anyone logging into the machine locally can access anything on them. If possible, lock resources like CD-ROM, floppy and zip drives and disable ports that can accept bootable devices such as USB keys and external hard drives. Set boot options in the BIOS to prevent boot-ups from removable media. Password-protect the BIOS to prevent unauthorized changes to boot options. Password-protect all servers and workstations.

Use Securable Filesystems

Remote database users (client applications) do not need filesystem permissions on databases. However, you should ensure that the appropriate permissions are imposed for using external applications to write to and read from external data files that are linked to tables. You can combine OS permissions and the server's **ExternalFileAccess** configuration to limit the risk exposure.

Users of embedded applications, including a “serverless” Classic connection on POSIX, do need rights to paths, the database file and other files, such as lock files, logs, external tables and so on. The same applies to any user account that runs a Firebird server as an application. Apart from those necessities, focus on all aspects that potentially put users in the way of Firebird assets and follow these rules-of-thumb:

- Use the maximum possible restrictions supported by your operating system and your chosen filesystem to protect the Firebird tree and other directories that are accessed by the Firebird server engine.
- Don't store Firebird software files, databases, scripts, backups or externally-accessed data files on FAT32 partitions. On Windows, don't enable ordinary users to access these partitions via shares. Any share permissions on NTFS partitions housing Firebird-related files and executables should be as limited as possible. Additionally, the most restrictive possible object permissions should be in force.
- Use group accounts in preference to individual accounts; avoid multi-group membership where possible.

Protect Backups

Back up regularly, compress backups to transportable media and store them securely off-site.

Some false assumptions

- A file-copy is bound to be corrupt, so it is no use to an intruder.
Do not assume that a file copy of a running database will be corrupt and unusable. An illicit copy may well prove to be useable, especially if update activity on the database is relatively infrequent, or if the attacker can try copying repeatedly.
- gbak files are not databases so they are no use to an intruder.
Don't leave backup files and database archives lying around the network where network cruisers can find them. A stolen copy of a gbak file can be restored with full visibility on any other Firebird server. That means: if you let me steal a copy of your database or you let me get your *gbak* file, I can restore it on my server. Because I am SYSDBA, I can open it and see everything.

Platform-based protection

The degree of platform-based protection you are able to apply to your database server installation depends on two general factors: how well the operating system platform and its filesystem can protect your system (software as well as data) and how secure your system has to be. The second may well be prescriptive for choosing the first.

Unless you have overpowering reasons to do otherwise, you should run the Firebird server as a service.

If possible, use a special user account to start the Firebird service. This is implemented by default on Linux since v.1.5 and some other POSIX builds, with the user `firebird`. On Windows, and for Firebird 1.0.x on any platform, it will be necessary to set up this special user account yourself and to be logged in as this user when using the installer to install Firebird.



Firebird 1.5 introduced an additional, optional Login switch to allow installer scripts to include the capability to make a “real user” the logged-in user for installing the service at boot-up. It is recommended that, for this purpose, the scripts create a “firebird” user with restricted privileges and set up the service install accordingly.

Restrict OS Logins

Require passwords for all logins and disable login caching on Windows servers and workstations. On networks, disable the ability for users and groups to change their own login settings. Require strong passwords. Enforce account lockout on all servers and workstations that connect to databases.

Enforce the use of ordinary, traceable accounts for normal work; restrict root or Administrator logins to administrative sessions. Eliminate ‘guest’, ‘world’ and ‘everyone’ accounts.

Monitor failed logins, failures to log out, failed file and program object accesses, failed user privileges, unusual shutdowns and boot-ups.

POSIX

Linux, UNIX and other POSIX-compliant platforms are often preferable to Windows when security is a major concern. The technologies for securing these platforms are mature and widely understood by implementors. Filesystem security and trusted access are inherent in design requirements that are determined by public standards. That is not to imply that merely installing a database server on a POSIX-conformant platform is a guarantee of security. It says that the elements needed for setting up secure systems that are reliably secure are present and capable of being implemented.

Microsoft Windows Platforms

Windows server installations are so infamously hard to secure that intense security requirements may simply rule out Windows altogether as a deployment platform for database servers where on-site security expertise and monitoring are not available.

Educate Yourself!

If you are a Windows application developer who is responsible for deploying Firebird securely on a Windows host, be sure to educate yourself on the risks and capabilities associated with each Windows host variant that you deem suitable for your software.

In an old-but-good lab presentation entitled ‘Hardening Windows 2000’, network security guru Philip Cox, of System Experts Corporation, begins by outlining **Four Steps to Practical Win2K Security** as:

- 1 Locate Windows system
- 2 Insert *nix CD
- 3 Reboot
- 4 Follow installation prompts

Yes, they were words spoken in jest, accompanied by the obligatory ‘smiley’. Cox's paper provides a seriously useful outline for system administrators for whom running Windows servers is the only option. He is the primary author of an authoritative and engagingly frank book about Windows server security.

Microsoft itself publishes a number of free white papers with detailed, practical instructions for implementing security on its server platforms and keeping abreast of its frequent security patches. The website <http://activewin.com> is a useful source for coherent descriptions of the many security patches that have been released and continue to be released for various Windows versions.

Windows Server Platforms

On Windows “service-capable” versions, services are available to remote clients even when no user is logged in at the server—the correct condition in which to leave an unattended server. Versions intended for single-user or home network use may be supplied with this safeguard disabled. Check thoroughly that the Windows version proposed for your deployment is one that enables this capability.

When running as a service, Firebird, like most Windows services, runs under the localsystem account. Localsystem is a built-in account that, on NT 4 and lower server versions, had few powers. On later Windows server platforms, localsystem was vested with an extraordinary level of access privileges for local system resources, including privileges that can not be granted even to members of the Administrators group.

By contrast, applications are run from regular user accounts and require the user to be logged-in to the server host. Any standard Windows user can start the Firebird server as an application.

File protection

On Windows, restricting the directory (folder) locations where database files and server artifacts live and protecting access to them are strongly recommended. Implement both object and share permissions on every file that network users can potentially reach. To be capable of protection, the directories and files must be on NTFS partitions, in trees dedicated to the purpose, and made readable only by a suitably privileged account or group.

When read access is thus restricted, remote clients must connect to the server through the TCP/IP protocol, not through Windows Networking (often referred to as ‘WNET’ OR ‘NetBEUI’).

The DatabaseAccess configuration parameter

For all versions of Firebird except the v.1.0 series, you can (and should) restrict the locations from which the server may read database files, by configuring the **DatabaseAccess** parameter in `firebird.conf` (q.v.).

Windows 95/98 and ME

When security is an issue, Windows 95/98 and ME systems should not be considered for use as Firebird hosts. They lack support for either services or file-level security. Anyone with access to the filesystem can readily make a duplicate copy of the database with a file-copying or archiving command or simply use the GUI features (drag ‘n’ drop, copy/paste, etc.) to steal a Firebird database.

At best, configuring DatabaseAccess will restrict the directory locations from which the server is allowed to read database files. However, the FAT32 filesystem is open to the world and offers no protection from accidental or malicious overwriting of databases, external function code or other Firebird-related files.



Support for Windows 9x and ME was officially dropped at Firebird 2.0.

Execution of arbitrary code

On a poorly protected system, all current Firebird versions provide the opportunity for arbitrary code to be executed, through the medium of external function libraries, BLOB filters and custom character set implementations. These external code modules run in the same address space as the server process, and with the same privileges.

Accordingly, it is important to protect the server from any possibility of its accessing external files and modules that have been written and planted by unauthorized users.

Firebird 1.0.x

In Firebird 1.0.x, you can configure specific directories to store external code modules and externally mapped data files and apply OS-level restrictions to prevent unauthorized access. It is strongly recommended to make use of this capability on filesystems that are capable of supporting it with filesystem access permissions.

However, a Firebird 1.0.x server can access external code and data anywhere on the filesystem that is under the host machine's control.

Firebird 1.5 and later versions

From V.1.5, the locations of external executables and other external objects can be tightly configured to ensure that the server will throw exceptions if it gets a request to access external objects in wrong places. For details about these configurations, refer to the notes about external objects in Chapter 34, *Configuration Parameters in Detail*.

Special risks with Windows services

Services running under localsystem profile are considered to be part of the trusted code base (TCB)—they have the same level of implied trust as Windows itself. Running the Firebird service on many Windows server platforms carries a recognized risk with regard to malicious exploits designed to execute arbitrary code. The importance of assuring the programmatic integrity of Firebird's external executables, even under very secure conditions, is crucial. In a Windows environment that is left open to exploits, the fire danger rises to "extreme".

In short, Windows operating system software on its own provides no credible guarantee of security for database servers, either within a local network or beyond. Potential exploiters must be stopped by tightly-configured user access control within the LAN and dependable, third-party firewall products to detect and block attacks.

Windows embedded server

The Windows embedded server library is, of course, designed to run on machines that do not run a full server. If you have a copy of fbembed.dll—renamed or not—housed anywhere on a full server machine, the server security is at risk. Here's why.

Embedded server does not use server authentication to verify that a user logging in to a database has a right to be doing so. Most application interfaces require a user name and password. Any user name will do, and any password—nothing is verified. The internal security of a database designed for embedded server use has to be protected with SQL permissions (see Chapter 37, *Database Security*) that limit which database objects can be accessed by which user names. That in itself is a can of worms on a stand-alone machine that is physically available to passers-by.

However, on a machine that is running a full Firebird server and also has the embedded server software on board, a malicious program could be set to run under an embedded server when databases are shut down, connect to the security database as SYSDBA, steal or damage the user records, read encrypted passwords, and generally please itself by visiting other databases as SYSDBA.

In versions prior to Firebird 2.5, an application connecting to a database through fbembed.dll locks the database file exclusively. A malicious application could hold its connections on databases—including the security database—indefinitely.

Embedded server applications and clients run in the address space of the OS user. To avoid having a malicious embedded server program being dropped into the system, be rigorous about restricting user and group access in the filesystem spaces where databases and Firebird system files live.

Wire Security

Much of the communication between the client and the server carries sensitive information which can be quite easily “sniffed” by someone eavesdropping network communications. For example, the encrypted password could be sniffed and used to gain unauthorized access to the server.

It is vital, therefore, to ensure that all pieces of the network path between any client and the server are trusted.

Add-on products can be purchased which provide encrypted network tunnelling solutions to block potentially insecure pathways.

Web and Other n-Tier Applications

Relying on default user names can have unforeseen effects, such as unintentionally bestowing the privileges of the database owner, or even the server process owner, on ordinary users. It is strongly recommended that you have your server application enforce input of a user name and password before any calls are made to the Firebird server process.

Use Dedicated Hosts

Avoid sharing the host machine with other services, especially vulnerable ones such as web and ftp servers that potentially invite anonymous logins. Shut down all services not required to run Firebird. On Windows, restrict access to the Registry by network users on database servers.

Establish Trustworthiness

Firebird can admit client connections to servers that bypass Firebird user authentication and use the operating system user and permissions scheme instead. On non-Windows systems it is a long-time feature that was inherited from InterBase® virtually undocumented. The system admin may mistakenly assume that the security database is the ultimate gatekeeper, whilst exposing the Firebird server to unprotected access by users that are assumed to be trustworthy.

When users log in without passing a Firebird user name and password, the authentication routine substitutes the current operating system identity for the Firebird user identity. If the OS user has root or Administrator privileges through some oversight in the network security scheme, there is a gaping hole.

In order for users to be allowed access to Firebird databases via their OS user credentials, it is essential to define a trusted host relationship between the server and each client workstation.



The environment variables ISC_USER and ISC_PASSWORD must be eliminated from production systems.

Firewalls

Placing your server machines behind a firewall is recommended, for obvious reasons. It may be less obvious that providing firewall protection to client processes is also a good idea. It is possible for a rogue user running on a trusted client machine to feed incorrect information to the server and gain privileged access to its databases. Windows clients are especially prone to compromise in this respect.

A Linux/UNIX server can be configured to recognize trusted clients explicitly. From there, the server implicitly trusts a process running on a trusted client.

Server Multi-hop

The restoration of the server redirection (‘multi-hop’) capability from Firebird 2.0 onward throws up a potential new vulnerability. For that reason, it is controlled by a parameter (**Redirection**) in `firebird.conf`, which is disabled by default. You should not enable it without clearly understanding how it exposes your servers.

These days, the ability to redirect requests to other servers is dangerous. Suppose you have one carefully protected firebird server, access to which is possible from the Internet. If this server has unrestricted access to your internal LAN by redirection, it will serve as a gateway for incoming requests like

```
firebird.your.domain.com:internal_server:/private/database.fdb.
```

Knowing the name or IP address of some internal server on your LAN is enough for an intruder: login access to the external server is not even needed. Such a gateway easily overrides a firewall that is protecting your LAN from outside attack.

Denial-of-Service Attacks

The original code that Firebird inherited had a large number of string copy commands that did not check the length of the data they were requested to copy. Certain of these overruns could be manipulated externally by passing large strings of binary data into SQL statements or pushing random garbage into the server port (currently 3050). Use of these functions is a common technique for malicious buffer overrun attacks intended to bring down servers.

As Firebird has progressed through its versions, most of these potential overrun vulnerabilities have been identified, removed and back-ported to the latter sub-releases of Firebird 1.5 and Firebird 2.0. Actually, it would be great to say that all have been

eliminated. However, new ones still do appear occasionally and, at the time of writing, only Firebird 2.1 and higher versions will ever get fixed now!

These vulnerabilities are more easily exploited if the server and client processes are not running on trusted networks and/or are not adequately firewalled.

Defensive programming can help to pre-empt D-o-S attacks on your system. Validating the lengths of strings from web input, for example, may be extremely useful.

Obviously, if you are not using currently supported versions of Firebird, you should at least ensure that your production sites are using the most recent sub-release.

Managing User Access

Users gain access to Firebird databases through an “authentication gateway” that is controlled by a specific Firebird server installation. A user that crosses this threshold gains access to all of the databases available to that server. Any user can create an object in any database but, by default, no user gets any sort of access to any object in a database.

User authentication in some form is required whenever a remote or local client connects to a Firebird database. The solitary exception is when the client connects through an embedded server application on Windows: no user/password authentication is performed during the embedded client/server connection process in this situation. However, for user permissions on the objects defined in the database, applications will need to pass the appropriate corresponding user name and the relevant **ROLE** parameter in the connection structures (DPB or SPB).

What a user can access in a database is controlled by granting SQL privileges, a topic that is discussed in the next chapter, **Database-level Security**.

The Security Database

The mechanism controlling user authentication is the security database that lives in the root directory of any Firebird server installation. It contains definitions of all users that have access to the Firebird server. A case-sensitive password must be defined for each user and used to gain access to the server.

The name of the security database differs from major version to major version:

- In Firebird 2 and higher it is `security2.fdb`
- In Firebird 1.5 it is `security.fdb`
- In Firebird 1.0 it is `isc4.gdb`

On all server installations except the Windows embedded server, it must be located in the Firebird root directory.



The embedded server on Windows does not use the security database.

Storage of USER Records

Prior to Firebird 2, the table that stores the authentication records is called USERS. In the older `security.fdb` and `isc4.gdb`, the SYSDBA can access that table directly and the *gsec* utility queries and updates it directly.

From Firebird 2.0 onward, the USER table is replaced by one named RDB\$USERS and is not accessible by any user, even SYSDBA. All access to RDB\$USERS is done via a view named USERS.

User Maintenance

The tool for maintaining the user table is *gsec*, a command-line application that can also be called to run in an interactive shell.

From Firebird 2.5 onward, DDL is available to suitably privileged users, to add, alter and delete users when connected to any user database. The detail can be found in the topic [Using DDL to Manage User Accounts](#) in Chapter 13, **Data Definition Language—DDL**.

Usage of *gsec* and the CREATE/ALTER/DELETE USER statements are discussed in detail further along in this chapter.

Migration of the Security Database

The security databases for the different major versions have differing internal structures, although an `isc4.gdb` database (from Firebird 1.0 or InterBase 6.0) can simply be backed up for a migration from Firebird 1.0 and then restored under the v.1.5 server as `security.fdb`. For a discussion of migrating security databases through the versions, refer to Chapter 5, [Migration Notes](#).

Firebird “Native” Users

The user table in the security database contains the login credentials of every ordinary user. For the purely “native” Firebird user, a record contains the user name and an encrypted string which is a hash of that user’s password. Optional fields are available for the user’s real name and, for POSIX users, their platform user and group IDs.

Platform Users

Platform users that are already logged in to the host computer on a version of Firebird that recognises trusted users can pass straight through the “authentication gateway” without needing to use the native Firebird login. For POSIX, this capability is available on all Firebird versions. For Windows, trusted user authentication became available at v.2.1 and was somewhat reimplemented for v.2.5.

Linux, MacOSX and Other POSIX

A platform user that is trusted by the host server’s access control mechanism can access databases without having to supply a native Firebird user name and password. However, any privileges in databases that have been granted to the native user will become available to the platform user only if its platform credentials are associated with a user record through use of the optional User ID (UID) and/or Group ID (GID) columns in the user record.

Windows: Trusted User Authentication

On Firebird versions 2.1 and higher, Windows “trusted user” security can be applied for authenticating Firebird users on a Windows host. When a client connects to a Firebird 2.1 or higher server on a Windows host that is configured for trusted user authentication, simply omitting the username and password from the database or services parameter blocks will automatically cause Windows trusted user authentication to be applied.

Illustration

Suppose you have logged in to the Windows server SRV as user 'John'. If you connect to server SRV with *isql*, without specifying a Firebird user name and password:

```
isql srv:employee
and do:
SQL> select CURRENT_USER from rdb$database;
you will get something like:
USER
=====
SRV\John
```



Where trusted user authentication is supported, Windows users can be granted SQL privileges on database objects and roles in the same way as regular Firebird users.

Configuring Trusted User Authentication

The parameter in *firebird.conf* for configuring the authentication method on Windows is **Authentication**. Its three possible values are **native**, **trusted** and **mixed**.

The default, **native**, configures authentication so that only Firebird security logins are allowed, providing full compatibility with previous Firebird versions and subverting malicious exploits on networks where Windows security is soft:

```
#Authentication = native
```

To use only Trusted User authentication and ignore Firebird login security altogether—which may offer better security than **native** if security on the Windows network is properly hardened—change the setting to

```
Authentication = trusted
```

The most accommodating setting, **mixed**, allows both Windows Trusted User authentication and Firebird's native authentication:

```
Authentication = mixed
```



*Trusted User authentication was enabled by default in Firebird 2.1 and 2.1.1., by defaulting the **Authentication** parameter to 'mixed'. It proved to be a problem for some types of deployments where Windows security was not well controlled. From v.2.1.2 onward, to reduce the risk of trusted user authentication becoming an inadvertent source or exposure, the default became 'native'.*

Environment Variables ISC_USER and ISC_PASSWORD

To retain the legacy behaviour, when the *ISC_USER* and *ISC_PASSWORD* variables are set in the environment, they are picked and used instead of trusted authentication. However, trusted authentication can be coerced to override the environment variables if they are set—discussed next.

Forcing Trusted Authentication

For the situation where trusted authentication is needed and there is a likelihood that the `ISC_USER` and `ISC_PASSWORD` variables are set, from v.2.1 onward, the parameter `isc_dpb_trusted_auth` can be included in the DPB.

Most of the Firebird command-line utilities support this parameter by means of the switch `-tru[sted]`. The usual rules for abbreviating switches apply, except to the *qli* and *nbackup* utilities.

```
Example    C:\Pr~\bin>isql srv:db          -- log in using trusted authentication
           C:\Pr~\bin>set ISC_USER=user1
           C:\Pr~\bin>set ISC_PASSWORD=12345
           C:\Pr~\bin>isql srv:db          -- log in as 'user1' from environment
           C:\Pr~\bin>isql -trust srv:db    -- log in using trusted authentication
```

qli and *nBackup*

The *qli* and *nBackup* utilities do not follow the usual pattern for switches and abbreviations: they use single-letter switches that are somewhat arcane. The switch of interest for *qli* is `-K`. The facility to force trusted authentication is yet to be implemented for *nBackup*. When it appears, expect its switch to be `-K` also.

Size of Platform User Names

Windows rules for full domain user names allow names longer than the maximum 31 characters allowed by Firebird for user names. The 31-character limit is enforced and, from V.2.1, logins passing longer names are disabled. This will remain the situation until the mapping of OS objects to database objects is implemented in a later Firebird version.

Privileged Users

As from version 2.5, Firebird recognises three types of users that have automatic access to all objects in all databases. They are:

- the “system database administrator” user SYSDBA
- platform “superusers”, including root on Linux and other POSIX platforms and, with conditions, domain Administrators on Windows
- ordinary users that have been given the privilege to use the RDB\$ADMIN role globally and which log in to a database using that role



The recognition of superusers on Windows is not available in Firebird 2.0 and prior versions.

Additionally, ordinary users can have elevated privileges on some or all objects in a specific database through either

- having been granted the RDB\$ADMIN role in that database and logging in with that role (Firebird 2.5 and higher)

or

- being the Owner of those objects



The database owner does not have any automatic privileges with respect to objects in the database that were created by another user, not even after a restore that changes the database owner. Database developers, especially those working in teams, should take care to settle on one user that will always be the owner of all objects and of the database and to make sure that all DDL requests are submitted by that user.

The SYSDBA User

SYSDBA has full destructive¹ rights to all databases on the server and is the owner of the security database.

All new installations of Firebird on Windows install the SYSDBA user into the security database, with the default password *masterkey*. Obviously, this is widely known and is not intended to be secure. It should be changed at the first opportunity.

On Linux and many other POSIX platforms, the installers generate a random password for SYSDBA. You will find it in the text file *SYSDBA.password*, located in the */bin* directory of the installation. Once the password has been noted and changed, that file should be deleted.



The SYSDBA password should not be distributed to ordinary users and it most certainly should not be hard-wired into application code.

Platform “Superusers”

Platform superusers are those users that are logged into the host with privileges equivalent to *root* on POSIX:

- on Linux and other POSIX platforms, the *root* user, along with any other users that have root privileges on the host server
- on a suitably configured Windows server, users with domain Administrator privileges

Linux, MacOSX and Other POSIX

The *root* user and others that have root privileges, via *su* or otherwise, are admitted with global SYSDBA privileges in all versions of Firebird.

Windows Administrators

If a local Administrator or a member of the built-in Domain Administrators group connects to Firebird using trusted authentication, he/she will be connected the same privileges as SYSDBA. It is therefore not necessary to grant specific SQL privileges to these users. Naturally, it also means taking care to be precise about dispensing domain privileges to your network users.

Entering User Credentials via SQL

Firebird 2.5 introduced DDL syntax to enable user accounts on the server to be managed by submitting SQL statements when logged in to a regular database.

It can be used by a privileged user as a direct alternative to *gsec*. It is also available to ordinary users to change their own passwords and update their personal information.

1. Meaning SYSDBA or equivalent can alter or drop objects at will, even the database itself.

Simple Syntax

The statements are CREATE USER, ALTER USER and DROP USER. CREATE USER and ALTER USER have optional parameters for including and removing escalated privileges, which are discussed a little later.

We look here at the simplest forms. The square brackets are not part of the syntax: they appear in the patterns to designate optional arguments.



Notice that, unlike the USER argument, which is an identifier, the other arguments are strings and must be single-quoted.

Adding a new user

The SYSDBA, or a user with escalated privileges in both the current database and the security database, adds a new user with this pattern:

```
CREATE USER user-name PASSWORD 'password' [FIRSTNAME 'firstname']
[MIDDLENAME 'middlename'] [LASTNAME 'lastname']
```



The PASSWORD argument is required when creating a new user. It should be the initial password for that new user. The user can change it later herself, using ALTER USER or gsec.

Example of adding a user

```
CREATE USER fluffy PASSWORD 'test';
```

Modifying an existing user

Once the user exists, a privileged user (or the user FLUFFY himself) can use ALTER USER to add some more information about FLUFFY. At least one of PASSWORD, FIRSTNAME, MIDDLENAME or LASTNAME must be present:

```
ALTER USER fluffy FIRSTNAME 'Foufou' LASTNAME 'Curlychops';
```

FLUFFY changes his password:

```
ALTER USER fluffy PASSWORD 'McPoodle';
```

Deleting a user

The SYSDBA, or a user with SYSDBA-equivalent privileges in both the current database and the security database, can delete a user:

```
DROP USER FLUFFY;
```

Using CREATE/ALTER USER to escalate privileges

The CREATE USER and ALTER USER statements also include two optional arguments that enable a user with SYSDBA privileges to grant the RDB\$ADMIN role in the security database to an ordinary user, thereby assigning global SYSDBA privileges to that user. The arguments are GRANT ADMIN ROLE and REVOKE ADMIN ROLE.

Assigning the RDB\$ADMIN role

The pattern for assigning the RDB\$ADMIN role in the security database by this method is:

```
CREATE | ALTER USER user-name [other-parameters]
[ {GRANT | REVOKE} ADMIN ROLE];
```

For example, to enable user FLUFFY to log in to any database with the RDB\$ADMIN role and its privileges:

```
ALTER USER fluffy GRANT ADMIN ROLE;
```

In future, when FLUFFY logs in to a database with RDB\$ADMIN as his role, he will have the same global privileges as SYSDBA. His privileges include WITH GRANT OPTION and WITH ADMIN OPTION, which means FLUFFY will be able log into any database, supplying 'RDB\$ADMIN' as the argument to the ROLE parameter and assign and rescind these and normal object privileges to and from other ordinary users, just as the SYSDBA would.

Rescinding the RDB\$ADMIN role

FLUFFY can only use these escalated powers as long as they remain assigned to him. To remove his privilege to use the RDB\$ADMIN role, use the REVOKE ADMIN ROLE argument:

```
ALTER USER fluffy REVOKE ADMIN ROLE;
```

More about privileged users

How this facility fits in with the rest of the server and database access control scenario is discussed further in the next two chapters.

The gsec Utility

Firebird provides a command-line utility named *gsec* as an alternative interface for maintaining the security database. It has its own shell for interactive use; or *gsec* commands can be run directly from an operating system command shell or an executable script (shell script or batch file).

Pre-V.2.5 Versions

In versions prior to Firebird 2.5, *gsec* is the only interface to the security database. For write access, its use is restricted to the SYSDBA and, on POSIX, the *root* user and any user that has Superuser privileges on the platform.

Any authentic user can run *gsec* to display user records but only the SYSDBA user can modify the data stored in them. The user name and password will be required unless

- the environment variables ISC_USER and ISC_PASSWORD are visible to the command shell in which you are running;

or

- you are logged in as a platform superuser



In Firebird versions below v.2.1, Windows system users are not recognised in any guise by the authentication module. Thus, Windows domain administrators are not recognised as superusers.

Current Version

In Firebird 2.5 and above, the range of users that can use *gsec* for write operations on the security database encompasses privileged Windows domain Administrators and users with global RDB\$ADMIN role privileges, in addition to those who have write permissions in earlier versions. To understand how to configure and implement these facilities, review the earlier parts of this chapter.

Further, any user can use `gsec` to change his or her own password and update the optional columns in the users table.

Starting a gsec interactive session

At the command line, at Firebird's `/bin/` directory prompt, type

```
gsec -user sysdba -password masterkey [-role RDB$ADMIN]
```

The prompt changes to `GSEC>`, indicating that you are running `gsec` in interactive mode. During an interactive session you can submit a small range of simple commands to add, modify and delete user records.

To end an interactive session, type `QUIT` at the prompt.

Running gsec as a remote client

A superuser can use `gsec` on a client machine to administer user authentication on a remote server. The invocation syntax is different: it needs the `-database` switch followed by the full network path to the security database or an alias for that path that has been defined in `aliases.conf` on the host server. For example (single command):

POSIX client to Windows server v.2.5:

```
./gsec -database hotchicken:c:\Program Files\Firebird\Firebird_2_5\security2.fdb
-user sysdba -password masterkey
```

Windows client to POSIX server (standard v.2.5 Linux installation):

```
gsec -database coolduck:/opt/firebird/security2.fdb
-user sysdba -password masterkey
```



Since different Linux and other POSIX flavours stipulate idiomatic rules regarding the location of user assets, the location of Firebird's root assets may differ from the examples.

Interactive commands

The `gsec` interactive commands are `display`, `add`, `modify`, `delete`, `help` and `quit`. They are not case-sensitive. Short forms are available: some or all of the characters shown here in square brackets may be omitted by dropping in strict right-to-left order.

`add`, `modify` and `delete` are used for adding and deleting users and for changing passwords. They require the username as a parameter, along with the relevant option switches and arguments.

Commands available to all users

The following interactive commands are available to any authenticated user. However, a non-privileged user will see only its own details:

display

Displays the main columns of the `USERS` table in the security database. The 'admin' column is not present in output from versions prior to v.2.5.

- Without the username argument, lists all users
- With the optional user name argument, displays the details of that user

```
GSEC> display
user_name  uid  gid admin          full name
-----
SYSDBA
MICKEY      123  345             Sql Server Administrator
Mickey Mouse
```

D_DUCK	124	345	Donald Duck
JULIUS	125	345	J. Caesar
...			

To display the same information for a single row from the USERS table:

```
GSEC> display username
For example:
GSEC> display julius
user_name  uid  gid admin      full name
-----
JULIUS     125  345                J. Caesar
```

Passwords

Passwords are never shown and there is absolutely no way, even for privileged users, to retrieve a password from the table.

help, or its alias **?**

Either of these switches displays a summary of the gsec commands, switches and syntax.

mo[dify]

For changing (editing) a column value on an existing user record. Supply the user name for the user to change, followed by one or more switches indicating the items to change and the new value for each.

For example, to change the first name to Michael and change his password to icecream, enter the following switches:

```
GSEC> modify mickey -fname Michael -pw icecream
```

Restrictions

- You can't change a user name. Requires a privileged user to delete the old user and add a new one.
- Ordinary users can modify only their own record.

q[uit]

Ends the interactive session.

Commands available only to privileged users

The following *gsec* commands are available only to the SYSDBA and users that have superuser privileges in the security database.

a[dd]

Adds a user to the users table. Pattern:

```
a[dd] user_name -pw password [other switches]
```

where username is a unique, new user name and password is the password that is to be initially associated with that user.



If you try to use illegal characters in a password string, gsec will simply terminate with no message.

The switch for the new password when adding a user or changing a password is -pw. Don't confuse it with the abbreviated form of the SYSDBA log-on password switch, which is -pa.

For example, to add authorization for a user named Harry Potter with user name hpotter and password noMuggle, enter:

```
GSEC> add hpotter -fname Harry -lname Potter -pw noMuggle
```


To verify the new entry:

```
GSEC> display hpotter
user_name  uid  gid admin          full name
-----
HPOTTER                                Harry Potter
```

To add a new user with the user name wombat and grant it the global RDB\$ADMIN role:

```
GSEC> add wombat -pw mollycod -fname Cute -mname Little -lname Marsupial -admin yes
```

```
GSEC> display
user_name  uid  gid admin          full name
-----
SYSDBA                                Sql Server Administrator
...
HPOTTER                                Harry Potter
WOMBAT                                admin          Cute Little Marsupial
```

If the user is an ordinary user that has received the global RDB\$ADMIN role privilege, the escalated privileges will be available only if the attachment includes the option **-role RDB\$ADMIN**.

Our new user, *wombat*, for example, would use the following command options when starting gsec in order to obtain the escalated privileges; otherwise, s/he is just an ordinary user:

```
gsec -database hotchicken:secdb -user wombat -password mollycod -role RDB$ADMIN
GSEC>
```



Notice also how a database alias is used here in the argument for the -database switch that is required for connecting with gsec remotely. On the hotchicken server, security2.fdb is defined in aliases.conf as secdb.

de[lete]

Deletes the user username from the USERS table. Delete doesn't take any other arguments or switches. The pattern is:

```
de[lete] username
```

For example:

```
GSEC> delete mickey
```

You can use the display command to confirm that the entry has been deleted.

Using gsec from a command prompt

To use *gsec* from a command prompt, convert each *gsec* interactive command to a command switch by prefixing it with a hyphen (–). The characters and arguments for the option switches are the same.

For example, to add user claudio and assign the password *dbkeycop*, you would enter the following on the command line:

Windows:

```
..\BIN> gsec -add claudio -pw dbkeycop -user SYSDBA -password masterkey
```

POSIX:

```
bin]$ ./gsec -add claudio -pw dbkeycop -user SYSDBA -password masterkey
```

To display the contents of the USERS table, enter:
 > gsec -display
..and so on.

The View USERS and gsec options

Table 36-1 shows the columns in the USERS view, together with their corresponding *gsec* option switches. The only required columns are USER_NAME and PASSWD.

Table 36.1 Columns in the Users view and gsec options

Column	Description	gsec Keyword	Switch equiv.	Argument
USER_NAME	User name, as recognized by the server's user authentication.	Required first argument for the interactive add, modify and delete commands and for the corresponding command-line switches -a[dd], -mo[dify] and -d[elete].	Same	Must conform to the rules for identifiers ¹
SYS_USER_NAME	Not used			
GROUP_NAME	Not used			
UID	On some POSIX platforms, the UNIX user ID. Not required.	UID	-uid	integer
GID	On some POSIX platforms, the UNIX group ID. Not required.	GID	-gid	integer
PASSWD	The current password for this user. Required.	PW	-pw	Character. Currently, the authentication module reads and recognises only the first 8 characters.
PRIVILEGE	Not used			
COMMENT	Not used			
FIRST_NAME	User's first name. Optional.	FNAME	-fname	VARCHAR(31)
MIDDLE_NAME	User's middle name. Optional.	MNAME	-mname	VARCHAR(31)
LAST_NAME	User's last name. Optional.	LNAME	-lname	VARCHAR(31)

Column	Description	gsec Keyword	Switch equiv.	Argument
FULL_NAME	User's full name. Computed, read-only			
Also—	—required when running any gsec command or starting an interactive session from a remote workstation		-database switch	Valid host name and full (not relative) file path to security database. Alias, if configured, may be used instead of full file path.

1. The USER_NAME parameter is actually an SQL identifier and should except if the naming rules for identifiers are broken. One such rule is use of a numeral as the first character. However, there is an anomaly in the overall system, whereby *gsec* will accept a user name such as 9qwerty and store it. Do not exploit this loophole: it is a bug that will be fixed in a v.2.5 sub-release.

gsec Error Messages

Table 36.2 gsec Errors: Causes and Suggested Actions to Take

Error Message	Causes and Suggested Remedies
Add record error	Invalid syntax, or you tried to add a user that already exists, or you are not the SYSDBA. Use modify if the user already exists.
<string> already specified	You included a switch more than once in an add or modify command. Re-enter the command.
Error in switch specifications	This message accompanies other error messages and indicates that invalid syntax was used. Check other error messages for the cause.
Find/delete record error	The delete command could not find the named user, or your login does not have global superuser privileges.
Find/display record error	The display command could not find the named user.
Find/modify record error	The modify command could not find the named user, or your login does not have global superuser privileges.
Incompatible switches specified	For example, you entered multiple switches for a delete command, which requires only the mandatory username argument. Correct the syntax and try again.
Invalid parameter, no switch defined	You specified a value for an argument but omitted the switch symbol for it.
Invalid switch specified	You specified an unrecognized option switch. Use the -help switch to check which switch you wanted, fix it and try again.
No user name specified	You must specify a user name after any add, modify or delete command or switch
Record not found for user: <string>	An entry for the user specified in the username argument could not be found. List the users with display, check your spelling and try again.

Error Message	Causes and Suggested Remedies
Unable to open database	The security database does not exist or is not located where you indicated in the -database argument. Are you running <i>gsec</i> from outside of the Firebird bin directory? Are you remotely trying to access a server that is not installed?
An error occurred while attempting to add the user. No permission for insert/write access to TABLE USERS	Did you forget to include the -role RDB\$ADMIN parameter when you attached to <i>gsec</i> ?

CHAPTER 37

DATABASE-LEVEL SECURITY

Database-level security in Firebird enables two security objectives: first, to prevent authorized users of the server from accessing the data in your database and, secondly, to enable access for users who do have business with your database. The means to implement this database-level security is by SQL privileges.

The first objective—to lock out unwanted server-authenticated users—has its uses in environments where databases owned by multiple entities are being run on the same server, for example, at a site providing shared colocation services for multiple customers. In such arrangements, customers of the service provider do not normally have SYSDBA access.

The second objective has more to do with an organization's requirements to restrict the purview of confidential or sensitive data. SQL privileges can support any level of granularity for access to any item of data, down to column level.

Unlike user authentication—discussed in the previous chapter—privileges apply at database level and are stored right in the database, in the system table RDB\$USER_PRIVILEGES.

Default Security and Access

A database and all its objects (tables, views and stored procedures) are secured against unauthorized access when they are created. That is, access is “opt-in”. No user can access any object in the database unless granted permission to do so. Except for especially privileged users—Owner, SYSDBA or a system “superuser”—users must be granted SQL privileges for any operation, even a SELECT.

A privilege enables a user to have some kind of access to an object in the database. It is enabled using a GRANT statement and taken away using a REVOKE statement.

The syntax pattern for enabling access is:

```
GRANT <privilege>
ON <object>
TO <user>;
```

```
A <privilege> granted to a <user> on an <object> constitutes a permission.  
For removing a permission:  
    REVOKE <privilege>  
        ON <object>  
        FROM <user>;
```

Several variants to the “core” syntax are available. We'll explore them a little later.

Planning an Access Scheme

Although defining users and roles and applying privileges are often deferred until a system is ready to deploy, designing the privileges scheme and coordinating it with the user list needs to be a planned part of overall system design. Maintaining a tree diagram of the scheme is very useful, both for designing and testing the scheme and for documenting it.

Refer to the later topic, *Unintended Effects with Privileges*, for some examples of ways a poorly planned scheme can cause headaches as privileges amass.

Metadata Tables

The system tables, where Firebird stores all metadata, including the SQL privileges themselves, are not protected by SQL privileges at all. They will become read-only in a future release.

The ‘Catch’

The ‘catch’ with this in all current Firebird versions is that any user with access to the server may create a valid object in any database—including declarations for external functions and tables linked to external tables—that could potentially be used as a mechanism to install and run malicious code on the server.

The ability to configure limits on the places a server is allowed to access external artefacts helps to mitigate the exposure. You must take explicit steps to implement the features and the supporting operating system file access restrictions. Default access to external files is set to NONE by the installers and the external function directories are restricted to the UDF tree. It is up to you to take care of the system restrictions.



At the end of this chapter, in the topic [A Trick to Beat Idiot Users and Bad Guys](#), my esteemed colleague Pavel Cisar shares a technique he discovered, that works around the lack of in-built permissions on the metadata structures by applying SQL privileges to the metadata tables.

SQL Privileges

A privilege represents permission to perform a DML operation. As we shall see later, multiple privileges can be packaged together and assigned to a ROLE. That package of privileges becomes a privilege that can be granted and revoked as though it were an object. Table 37-1 lists the SQL privileges that can be granted and revoked:

Table 37.1 SQL privileges

Privilege	Access
SELECT	Read data
INSERT	Create new rows
UPDATE	Modify existing data
DELETE	Delete rows
REFERENCES	Refer to a primary key from a foreign key. It is always a necessary accompaniment to granting privileges on tables containing foreign keys.
ALL	Select, insert, update, delete, and refer to a primary key from a foreign key
EXECUTE	Execute a stored procedure or call it using SELECT. This privilege is never granted as part of the ALL privilege.
ROLE	Acquire all privileges assigned to the role. Once a role exists and has privileges assigned to it, it becomes a privilege that can be granted explicitly to users. A role is never granted as part of the ALL privilege.

Objects

The “other half” of a permission is the object on which the privilege is to be granted or from which it is to be revoked. An object can be a table, a view, a stored procedure or a role, although not all privileges are necessarily applicable to all types of objects. An UPDATE privilege, for example, is not applicable to a procedure and an EXECUTE privilege is not applicable to a table or a view.

There is no “packaged object” that encompasses all, or groups of objects—there will be at least one GRANT statement for each database object.

Users

Users are the recipients of permissions and the losers when permissions are revoked. A <user> can be:

- a “native Firebird user”—one that is authenticated in the security database
- a POSIX account or group
- a Windows user authenticated as a trusted user (Firebird 2.1 and higher versions)
- a user with escalated privileges
- a database object

Native Firebird users

A native Firebird user is one that is defined in the security database (`security2.fdb`, or `security.fdb` in version 1.5).

You can actually grant privileges to a user name that does not exist in the security database: the Firebird engine does not check for a pre-existing user when it processes GRANT and REVOKE requests. The reason is that Firebird can admit users that are already “trusted” by the operating system’s authentication system (q.v.)

Privileges granted to a user that does not exist at all do no harm. Users come and go. When a user departs from the security database, those privileges are not deleted, nor even revoked. They simply become unused. However, if there’s a chance of a departed user name being “recycled” for a different person, he or she might acquire some unintended privileges.

Embedded servers

It is strongly recommended that databases intended for use in the Windows embedded server versions prior to v.2.5 be tightly protected with privileges. Since the user of a pre-2.5 embedded server on Windows is, by design, not authenticated, it will be necessary to build into the application code a “mock-user” that the application can pass at connection time to match up with the privileges. Indeed, some middleware designed for the Firebird API actually requires both a user name and a password.

While a made-up user name must match the recipient of the privileges, the password can be any string that is in a valid Firebird password format.

The PUBLIC user

PUBLIC is a user that stands for “all users”. It does encompass stored procedures, triggers, views or roles that need privileges.

If multiple databases are running on the server, granting large packages of privileges to PUBLIC might save a lot of typing but, on a network that is serving multiple databases to multiple sets of client, it is fairly easy to grant privileges by mistake to users that should not have them.

POSIX users and groups

Firebird supports users and groups on POSIX platforms. If system-level authentication is implemented, you can grant privileges to groups—refer to the TO GROUP <UNIX-group> option for GRANT and REVOKE.

Windows trusted users

Since v.2.1, trusted user authentication on Windows has been supported. “Trusted users” are ordinary users that are authenticated by the operating system’s centurions instead of Firebird’s. Trusted users do not get automatic privileges to any objects: they must be granted.

Users with escalated privileges

The SYSDBA user has special rights to all databases and the objects within them, regardless of which user owns them. Furthermore, on operating systems that implement the concept of a “superuser”—a user with root or locksmith privileges—such a user also has full access and destructive rights access to all databases and their objects if it logs in under the root ID.

From Firebird 2.1 forward, Windows users with Administrator rights can acquire SYSDBA’s powers, much as the POSIX root user does. From v.2.5 it is also possible to give an ordinary user elevated privileges in a specific database by granting the predefined role RDB\$ADMIN to that user.

Initially, an object's creator, its owner, is the only user other than SYSDBA or a user with elevated privileges that has full access to an object (table, view, stored procedure or role) or can permit other users to access it. Either user can then start off “chains” of permissions by granting other users the right to grant privileges on specified objects. This right is optionally passed on by appending the WITH GRANT OPTION qualifier to the permission.

In a similar manner, SYSDBA or the owner of a role can optionally qualify a role privilege that it grants to a user as WITH ADMIN OPTION. However, privileges assumed by a user that logs in under a role does not inherit WITH GRANT OPTION authority from the role. More of this later.

Database objects as ‘users’

Views need permissions to access tables, other views and stored procedures.

Procedures and triggers: A stored procedure that accesses tables and views and executes other procedures need privileges on those objects. A trigger that executes procedures needs privileges on those and also on any tables or views it accesses.



Whilst a trigger might require privileges to access another object, access to the execution of the trigger comes to the user along with privileges on the table that owns the trigger. To condition access to a trigger by users that have privileges to the table, you can predicate conditions in the triggers on the context variables CURRENT_USER or CURRENT_ROLE, as appropriate.

Roles: When a role is being awarded privileges, it is a user. Once role has been granted the required privileges to objects, it “changes hats” and becomes a privilege that can be granted to some other types of users. When the user logs in to the database with that role in its log-in credentials, it assumes the privileges that were awarded to the role.

Privilege Restrictions

- The privileges SELECT, INSERT, UPDATE, DELETE are applied only to objects that are tables or views. REFERENCES applies only to tables.
- For views, the user of the view must have privileges for view itself, but permissions on the base tables must somehow be granted as well. The rule is that either the view's owner, the view itself or the user of view must have the appropriate privileges to the base tables. It doesn't matter how the privileges are acquired, but one of the three must have it.

Naturally updateable views also need SELECT, INSERT, UPDATE and DELETE permissions on the base tables. When read-only views are made updateable by means of triggers, the triggers need permissions on the underlying tables, according to the operations defined by the trigger events.



To be able to create a view, it is necessary to have SELECT permissions for the base tables. In the rare case that any of those SELECT privileges is revoked after the VIEW is created, it must be added to the view itself or to users of the view.

- EXECUTE can only be applied to stored procedures.
- A role is never granted “on” any object. It is first packaged and then granted—like an object—TO a user.

Granting Privileges

A GRANT statement is used to assign a specific privilege for accessing an object to a user, role, view or PSQL module.

The general syntax pattern for granting privileges on objects is:

```
GRANT <privileges>
  [ON [TABLE] <table> | <view> | <object> ]
  TO <generic-user>
  [{WITH GRANT OPTION} | {WITH ADMIN OPTION}];
```

```
<privileges> = <privilege> | <privilege-list> | <role-name> | ALL
<privilege> = INSERT | DELETE | UPDATE [(column [, column [,..]] ) ]
| REFERENCES [(column [, column [,..]] ) ] | EXECUTE
<privilege-list> = [, privilege [, <privilege-list> [...]]]
```

- Notice that the <privilege> syntax includes the provisions for restricting UPDATE or REFERENCES to certain columns, discussed in the next topic.

```
<object> = <stored-proc> | <role-with-privileges>
<generic-user> = <user> | PUBLIC | <user-list> | <UNIX-user> | GROUP <UNIX-group> |
<user-object>
<user-list> = <user>, {<user> | <user-list>}
<user-object> = <role> | <trigger> | <stored-proc>
```
- The ON clause is mandatory except that, in a statement that grants a role to a user, it is invalid and is omitted.

The following statement grants some privileges for the DEPARTMENTS table to a user, CHALKY:

```
GRANT SELECT, UPDATE, INSERT, DELETE ON DEPARTMENTS TO CHALKY;
```

UPDATE rights on columns

The UPDATE privilege, unmodified, lets the user update any column in the table or view. However, if you specify a comma-separated list of columns, the user will be restricted to updating only the specified columns.

In the following statement, all Firebird-authenticated users will have update permissions for the CUSTOMER table but they will only be able to update CONTACT_FIRST, CONTACT_LAST and PHONE_NO:

```
GRANT UPDATE (CONTACT_FIRST, CONTACT_LAST, PHONE_NO) ON CUSTOMER TO PUBLIC;
```

- When the option to grant UPDATE on a list of columns is used, multiple permissions are stored in the system table RDB\$USER_PRIVILEGES, one for each column. Rights can be granted or revoked on each column individually.
- When the column-level permission is not used, only one permission is created. There is no way to remove rights on some columns and keep others. It would be necessary to revoke the permissions that contain the rights you want to remove and add a new one with the amended rights.

Gotcha!

Thanks to the rules for SQL privileges—"the funniest thing since the Marx Brothers", according to a colleague—it is possible to grant both column-level permissions and the unqualified table-level UPDATE and REFERENCES privileges to the same user. It can cause complications if a user's column-level UPDATE or REFERENCES right is revoked, since the same user's table-level permission is unaffected.

Limiting access using Views

Views offer an elegant way to restrict access to tables, by restricting the columns and/or the rows that are visible to the user in highly-customized ways. This topic is discussed in Chapter 23, [Views and Other Run-time Set Objects](#) and below, in the topic [Privileges on Views](#).

REFERENCES rights on columns

The REFERENCES privilege is a necessary accompaniment to granting permissions on a table that has a foreign key. It is needed if a user creating a foreign key in a table does not own the table referenced by the key.

REFERENCES grants permissions on columns. If the GRANT REFERENCES statement refers to the foreign key target table without specifying columns, the privileges are granted on every column. However, columns that are not involved in the referential link are not made accessible.

If you prefer, you can specify just the key columns and, perhaps, save a little overhead if the referenced table has a lot of columns. If you do so, you must specify *all* of the linking key columns. The simplified syntax pattern is:

```
GRANT REFERENCES
  ON <primary-table> [ ( key-column [, <key-column> [, ...]] ) ]
  TO <needful-user>
  [WITH GRANT OPTION] ;
```

The next example grants REFERENCES privileges on DEPARTMENTS to CHALKY, permitting CHALKY to write a foreign key that references the primary key of the DEPARTMENTS table, even though CHALKY doesn't own that table:

```
GRANT REFERENCES ON DEPARTMENTS(DEPT_NO) TO CHALKY;
```



If the visibility of the keys is not an issue, grant the REFERENCES privileges to PUBLIC.

Privileges needed by objects

When a trigger, stored procedure or view needs to access a table or view, it is sufficient for the user or object that is executing it to have the necessary permissions.

On the other hand, privileges on tables or views can be granted to a PSQL module or view instead of to individual users, as a security measure. If the target object's owner is not the same as the owner of the object that wants access, the stored procedure, view, or trigger will need privileges to access the target object. The ultimate user needs only the appropriate privilege on the accessing view or trigger (table), or the EXECUTE privilege on the accessing procedure.

To grant privileges to a trigger or stored procedure, include the keywords TRIGGER or PROCEDURE, as appropriate, before the name of the module.

Here, the procedure COUNT_CHICKENS is granted the INSERT privilege for the PROJ_DEPT_BUDGET table:

```
GRANT INSERT ON PROJ_DEPT_BUDGET TO PROCEDURE COUNT_CHICKENS;
```

Granting the EXECUTE privilege

To use a stored procedure, users, triggers or other stored procedures need the EXECUTE privilege on it. If a view selects output fields from a selectable stored procedure, the view must have the EXECUTE privilege—not the SELECT privilege.

The simplified syntax pattern is:

```
GRANT EXECUTE
  ON PROCEDURE <procedure-name>
  TO <grantee>;
<grantee> = [ PROCEDURE <procedure-name> [, <procedure-name> [, ..]]
            [ TRIGGER <trigger-name> [, <trigger-name> [, ...]]]
            [ VIEW <view-name> [, <view-name> [, ...]]]
            | <role-name> | <user-or-list> | PUBLIC
[WITH GRANT OPTION];
```

A stored procedure or trigger needs the EXECUTE privilege on a stored procedure whose owner is not the same as its own. Note, a trigger is owned by the owner of the table that owns it.

If your GRANT EXECUTE statement is granting privileges to PUBLIC, no other types of grantees can be listed as TO arguments.

Here, the GRANT EXECUTE statement grants the privilege on the procedure CALCULATE_BEANS to two ordinary users, FLATFOOT and KILROY, and to two stored procedures whose owners are not the owner of CALCULATE_BEANS:

```
GRANT EXECUTE ON PROCEDURE CALCULATE_BEANS
  TO FLATFOOT,
  KILROY,
  PROCEDURE DO_STUFF, ABANDON_OLD;
```

Privileges on Views

Privileges on views are somewhat complicated. The owner of the view needs to grant the SELECT privilege to users, just as a table-owner would. The complications start if the view is updateable—either naturally, or through view triggers—or the view involves other views or selectable stored procedures.

Of course, a view’s data is transient. Data changes on an updateable view are actually made to the underlying (“base”) tables. Unless the owners of the base objects have already granted the user the applicable rights—INSERT, UPDATE, DELETE, EXECUTE—on the base tables and objects and any selectable stored procedures or views, the user will need to acquire them from the view’s owner.

REFERENCES privileges are not applicable to views, except under one (usually avoidable) situation. If a view uses a table that has foreign keys to other tables, the view needs REFERENCES privileges to those other tables if the tables themselves are not used in the view.

For more on the subject of views and the privileges entailed, refer to the topic *Privileges for Views* in Chapter 23, *Views and Other Run-time Set Objects*.

Bundling Multiple Privileges

It is possible to grant several privileges in one statement and to grant one or more privileges to multiple grantee users or objects.

Lists of privileges

To give a grantee several privileges on a table, name the granted privileges in a comma-separated list. The following statement assigns INSERT and UPDATE permissions on the DEPARTMENT table to user CHALKY:

```
GRANT INSERT, UPDATE ON DEPARTMENT TO CHALKY;
```

A list of privileges can be any combination, in any order, of SELECT, INSERT, UPDATE, DELETE and REFERENCES.

SELECT, INSERT, UPDATE, DELETE and REFERENCES privileges that were granted singly or in lists can be revoked singly or in lists. A REVOKE list does not have to match any originating GRANT list.

Exclusions

- EXECUTE privileges have to be granted or revoked in a separate statement, not in combination with other kinds of privileges
- Roles cannot be granted or revoked in combination with other kinds of privileges
- The REFERENCES privilege can not be granted to a view

The ALL privilege

The ALL privilege combines SELECT, INSERT, UPDATE, DELETE and REFERENCES all in one package. For example, the following statement grants CHALKY the whole package of permissions for the DEPARTMENT table:

```
GRANT ALL ON DEPARTMENT TO CHALKY;
```

You can also assign the ALL package to triggers and procedures. In this statement, the procedure COUNT_CHICKENS gets the whole package of rights to the PROJ_DEPT_BUDGET table:

```
GRANT ALL ON PROJ_DEPT_BUDGET TO PROCEDURE COUNT_CHICKENS;
```

Roles and the EXECUTE privilege are not included in the ALL package.

Roles

A role is created in a database and is available only to that database. Think of a role as a wrapper around a bundle of privileges—a package. Once the package is wrapped by having some privileges granted to it, it becomes available to be assigned—as a privilege—to some types of users.

Multiple roles can be created, the idea being to package and control discrete sets of privileges that can be granted and revoked as a whole, rather than as numerous collections of individual privileges being granted and revoked repetitively and in an ad hoc fashion.

A role can never be granted as part of the ALL package, although a role can be granted the ALL privilege and also other roles.

Roles are not groups

Roles are not like operating system user groups. A Firebird user can be assigned more than one role, but can log in under only one role in a session.

Implementing Roles

Implementing roles is a four-step process:

- 1 Create a role using the `CREATE ROLE` statement
- 2 Assign privileges to the role using `GRANT <privilege> TO <rolename>`
- 3 Grant the role to users using `GRANT <role-name> TO <user-name>`.
- 4 Specify the role, along with the user name, when attaching to a database.

Creating a role

The syntax pattern for creating a role is simple:

```
CREATE ROLE <role-name>;
```

SYSDBA or the database owner can create roles, grant privileges to them and, initially grant these “loaded” roles to users. If a role is granted WITH ADMIN OPTION, the recipient of the role can grant it on to other users, optionally including WITH ADMIN OPTION.

Assigning role privileges

To “load” a role with privileges, just grant the required privileges as if the role were a user:

```
GRANT <privileges> TO <role-name>;
```

Granting a role to users

The GRANT statement for granting a role to users omits the ON clause—it’s implicit in the permissions already “loaded” into the role.

```
GRANT <role-name> [, <role-name> [, ...]]
  TO [USER] <user-name> [, [USER] <user-name> [, ...]]
  [WITH ADMIN OPTION];
```

‘WITH ADMIN OPTION’

The optional WITH ADMIN OPTION allows grantees to grant the role to other users and to revoke it. It works in a similar way to WITH GRANT OPTION for regular permissions—see [Granting the Right to Grant Privileges](#), below.

Example

The following example creates the MAITRE_D role, grants ALL privileges on the DEPARTMENT table to this role and then grants the role to HORTENSE, using WITH ADMIN OPTION to enable Hortense to grant the MAITRE_D role to others. This gives HORTENSE the privileges SELECT, INSERT, UPDATE, DELETE and REFERENCES on DEPARTMENT.

```
CREATE ROLE MAITRE_D;
COMMIT;
GRANT ALL ON DEPARTMENT TO MAITRE_D;
GRANT MAITRE_D TO HORTENSE WITH ADMIN OPTION;
```

Attaching to the database under a role

When connecting, include ROLE in the connection parameters and specify the role whose privileges you want to acquire for that connection. It will only work if your user name has been granted the role:

```
CONNECT <database-path>
  USER <your-user-name>
  ROLE <role-name>
  PASSWORD <your-password>;
```

Dropping a role

If you drop a role, all privileges that were conferred by that role are revoked. To drop the role MAITRE_D:

```
DROP ROLE MAITRE_D;
```



If you only want to remove the privileges granted to a user through a role, or to remove privileges from a role, use a REVOKE statement (see [Revoking Privileges](#), below).

Privileges for Multiple Users

Firebird SQL implements features to enable you to grant privileges to multiple users in a single statement. You can assign privileges to

- a list of named users
- all users that have access to the database (PUBLIC)

- a POSIX group
- a role—then assign that role to a user list, to PUBLIC or to a POSIX group

To a list of users

To assign the same access privileges to a number of users in a single statement, provide a comma-separated list of users in place of the single user name.

The following statement gives INSERT and UPDATE permissions on the DEPARTMENT table to MICKEY, DONALD and HPOTTER:

```
GRANT INSERT, UPDATE ON DEPARTMENTS TO MICKEY, DONALD, HPOTTER;
```

To a POSIX group

Operating system account names on Linux, MacOSX and other POSIX platforms are accessible to Firebird security through a feature of Firebird privileges that is not standard SQL. A client running as a POSIX user adopts that user identity in the database, even if the account is not defined in the Firebird security database.

The machine accessing the the server must be listed as a trusted host on the server (files /etc/host.equiv or /etc/gds_host.equiv, or in .rhost in the user's home directory on the server). On connecting, the user gets logged in under its group identity, as long as it does not supply its Firebird user name and password as a connection parameter—Firebird credentials overrule POSIX credentials.

POSIX groups share this behavior: SYSDBA or a superuser can assign SQL privileges to UNIX groups. Any OS-level account that is a member of the group inherits the privileges granted to the group. For example:

```
GRANT ALL ON CUSTOMER TO GROUP sales;
```



At the time of this writing, no way was implemented to group Windows trusted users for acquiring privileges, other than as a comma-separated list.

To all natively authenticated users: PUBLIC

To assign the same access privileges on a table to all users, grant the privileges to PUBLIC. PUBLIC encompasses any users that can connect to the database—not triggers, procedures, views or roles.

```
GRANT SELECT, INSERT, UPDATE ON DEPARTMENT TO PUBLIC;
```

Privileges granted to users by way of PUBLIC can only be revoked by revoking them from PUBLIC. You can't, for example, revoke a privilege from CHALKY, that CHALKY acquired as a member of PUBLIC.

To a list of procedures

To assign privileges to several procedures in a single statement, provide a comma-separated list of procedures. Here, two procedures get privileges in one statement:

```
GRANT INSERT, UPDATE
ON PROJ_DEPT_BUDGET
TO PROCEDURE CALCULATE_ODDS, COUNT_BEANS;
```


Granting the Right to Grant Privileges

Initially, only the owner of a table or view, or SYSDBA, can grant permission on the object to other users. Add `WITH GRANT OPTION` to the end of the `GRANT` statement to transfer the right to grant privileges on to the user, along with the privilege itself.

The following statement assigns a `SELECT` permission to user `HPOTTER` and allows `HPOTTER` to grant the `SELECT` permission to others:

```
GRANT SELECT ON DEPARTMENT TO HPOTTER WITH GRANT OPTION;
```

`WITH GRANT OPTION` can not be assigned to a trigger or procedure.

`WITH GRANT OPTION` rights are cumulative, even if issued by different users. For example, `HPOTTER` can get the grant authority for `SELECT` on `DEPARTMENT` from one user and for `INSERT` on `DEPARTMENT` from another.

In the example, `HPOTTER` was granted `SELECT` access to the `DEPARTMENT` table with grant authority. `HPOTTER` can grant this `SELECT` permission to other users. Suppose `HPOTTER` is now granted `INSERT` permission on the table as well, but without the grant authority:

```
GRANT INSERT ON DEPARTMENT TO HPOTTER;
```

`HPOTTER` can select from and insert to `DEPARTMENT`. He can grant `SELECT` permissions on `DEPARTMENT` to other users; but he can't assign `INSERT` permissions because he doesn't have grant authority for that privilege.

The user's existing privileges can be extended to include grant authority, by issuing a second `GRANT` statement for the same privilege, that includes `WITH GRANT OPTION`.

To give `HPOTTER` authority to grant `INSERT` permission on `DEPARTMENT`, just issue a new statement:

```
GRANT INSERT ON DEPARTMENT TO HPOTTER WITH GRANT OPTION;
```

In summary, a user can grant an access privilege (`SELECT`, `INSERT`, `UPDATE`, `DELETE` and `REFERENCES`) on an object to other users or objects, if that user either

- owns the object
- has been granted that privilege on that object `WITH GRANT OPTION`
- has acquired the privilege by being granted a role containing that privilege `WITH ADMIN OPTION`

SQL does not reject `GRANT` statements that cause a user's permissions to be duplicated. An ad hoc approach to granting privileges can cause unintended adverse effects—discussed in the topic [Unintended Effects with Privileges](#) later in this chapter.

Granting Privileges on Behalf of Another

From v.2.5 forward, it is possible for the `SYSDBA` or another user who is logged in under the `RDB$ADMIN` role to grant and revoke privileges on behalf of another user. The purpose is to enable that privileged user to sustain the “chain of authority” for the actual owner (or owners) of each database object.



It is often overlooked that the owner of a database does not own any objects inside the database that were created by other users. A common (and unwise) scenario is for a regular user to be the database owner (good!) but for `SYSDBA` to own all of the objects

(bad!). Care should always be exercised during development, maintenance and restores to keep object ownership consistent.

The GRANTED BY clause

The mechanism for granting privileges on behalf of another is the GRANTED BY clause, which can be substituted by the keyword AS. The syntax pattern is:

```
GRANT <privilege> ON <object> TO <recipient-user>
[ { granted by | as } [ user ] <grantor-username> ]
```

The USER keyword is optional.

Here, SYSDBA is logged in to a database that is owned by user FBADMIN, and grants a permission on behalf of the owner:

```
GRANT ALL ON CUSTOMER TO HPOTTER GRANTED BY FBADMIN;
```

It could have been issued using the alternative keyword AS:

```
GRANT ALL ON CUSTOMER TO HPOTTER AS USER FBADMIN;
```

The syntax for revoking a privilege on behalf of the user that originally granted it is similar and uses the same keywords:

```
REVOKE <privilege> ON <object> FROM <user>
[ { granted by | as } [ user ] <grantor-username> ]
```

This time, SYSDBA revokes part of the ALL privilege on behalf of the owner:

```
REVOKE DELETE, INSERT, UPDATE ON CUSTOMER FROM HPOTTER AS FBADMIN;
```

The System Role RDB\$ADMIN

The system role RDB\$ADMIN was introduced in v.2.5 for ODS 11.2+ databases. It can be granted to an ordinary user by the SYSDBA, the owner or another user with escalated privileges, to escalate the privileges of that ordinary user to the equivalent of those of the SYSDBA, *in that database*.

You grant and revoke the RDB\$ADMIN role in just the same way as you would a regular role. However, because it is a system object, it has some usage restrictions:

- it cannot be modified by granting it more privileges or revoking any of its pre-packaged privileges
- it cannot be dropped under a v.2.5 or higher server



The WITH ADMIN OPTION (for the privilege to grant this role to others) and WITH GRANT OPTION (for the privilege to grant permissions on objects to other users without being the owner of those objects) are implicit in the RDB\$ADMIN role and are not included in the syntax.

Extending RDB\$ADMIN power for Windows Administrators

A domain Administrator on Windows could get the equivalent power of the POSIX root user power in all ODS 11.2+ databases on the server by being granted the RDB\$ADMIN role for each database, one by one. However, the SYSDBA can configure the server to map

domain Administrators to the RDB\$ADMIN role in all databases automatically. The ALTER ROLE statement, first available in Firebird 2.5, is used to achieve this (and only this) purpose.

ALTER ROLE statement

For ALTER ROLE to have any effect, the server first needs to be configured for trusted user authentication, by setting the **Authentication** parameter in firebird.conf to either *trusted* or *mixed*.



If you have to do this step for a Superserver or Superclassic installation, remember to restart the server after making the change.

The following statement enables the automatic mapping:

```
ALTER ROLE RDB$ADMIN
SET AUTO ADMIN MAPPING;
```

If automatic mapping needs to be disabled, to prevent Administrators from getting SYSDBA privileges automatically in all databases, issue this statement:

```
ALTER ROLE RDB$ADMIN
DROP AUTO ADMIN MAPPING;
```

Unintended Effects with Privileges

SQL allows the same grantee to get the same permissions from different grantors, even if it would duplicate a permission the grantee already has. Every time one user extends grant authority to another user, it opens one more source through which any user could receive permissions. The permissions structure has the potential to become the proverbial bird's nest, from which it is very difficult to extricate the actual state of permissions for an individual user or object.

Suppose two users to whom the appropriate privileges and grant authority have been extended, SERENA and HPOTTER, both issue the following statement:

```
GRANT INSERT ON DEPARTMENT TO BRUNHILDE
WITH GRANT OPTION;
```

Later, SERENA revokes the privilege and grant authority for BRUNHILDE:

```
REVOKE INSERT ON DEPARTMENT FROM BRUNHILDE;
```

SERENA thinks BRUNHILDE no longer has INSERT permission or grant authority for the DEPARTMENT table. However, the REVOKE appears to have no effect, since BRUNHILDE still has the INSERT permission and grant authority assigned by HPOTTER.

As the number of users with privileges and grant authority for a table proliferates, the likelihood that different users can grant the same privileges and grant authority to any single user also expands. Quite simply, like nuclear fission, it can go out of control. It can become a big job just to revoke a specific permission. Revoking all (or many) permissions for a particular user can become an astronomical challenge.

If it is possible that the user might have received rights from several grantors, there are two possible solutions, both messy:

- Find each and every permission granted to that user, along with the grantor in each case, and have each grantor revoke each permission it granted. This gets complicated when

ALL and PUBLIC are involved, since revoking a more targeted permission doesn't revoke rights acquired through ALL, PUBLIC, roles or groups.

- The owner of each table or object (or SYSDBA) issues REVOKE statements affecting all users of the table, then issues GRANT statements to re-establish access privileges for the users who need to keep their rights.

The server gives no feedback about any REVOKE command, regardless of whether it succeeds or fails. It's around this point in your first misadventure with SQL permissions that you roll your eyes to the heavens and ponder on the real mission of standards committees on Earth.



A well-designed graphic rights manager utility can save your sanity. Fortunately, many of the desktop administration programs do provide this support. A survey of the offerings is available at the [IBPhoenix website](#).

Revoking Privileges

A REVOKE statement is required for removing privileges assigned by GRANT statements. According to the standard, the REVOKE should cascade down through all grantees that acquired the same privilege as a result of a WITH GRANT OPTION grant from this user. However, you should not rely on this in Firebird, since conflicting rules in the standard could cause implementation logic to prevail over the standard under some conditions.

REVOKE statements can remove any privilege that GRANT can assign. However, the user that issues the REVOKE matters. A privilege must be revoked by the same user that granted it—the grantor—or by SYSDBA or an equivalent Superuser. If the same privilege was granted by more than one user, the other privileges are not affected.

Permissions acquired “in bulk” cannot be revoked individually. That means

- a privilege that a user acquired by being granted ALL or a role can only be removed by the original grantor revoking ALL or the role, respectively
- revoking a privilege for a user that got the privilege by way of a grant to PUBLIC or to a UNIX group can only be achieved by the original grantor revoking the privilege on PUBLIC or the group, respectively
- privileges granted to PUBLIC can only be revoked FROM public

Using REVOKE

The simplified syntax pattern for REVOKE is the other face of the GRANT syntax. The TO <grantee> clause is replaced by FROM <grantee>:

```
REVOKE <privileges> ON <object> FROM <grantee> ;
```

The following statement removes the SELECT privilege for the user KILROY on the DEPARTMENT table, if it was granted with GRANT SELECT:

```
REVOKE SELECT ON DEPARTMENT FROM KILROY;
```

The following statement removes the UPDATE privilege on the CUSTOMER table from the procedure COUNT_BEANS:

```
REVOKE UPDATE ON CUSTOMER FROM PROCEDURE COUNT_BEANS;
```

The next statement removes the EXECUTE privilege that was granted to the procedure, COUNT_BEANS, on the ABANDON_OLD procedure:

```
REVOKE EXECUTE ON PROCEDURE ABANDON_OLD FROM PROCEDURE COUNT_BEANS;
```



You can use the SHOW GRANTS command in isql to list all the privileges and how they were granted. For example,

```
./isql mydb -user sysdba -password masterkey
```

```
Database: mydb, User: sysdba
```

```
SQL> show grants;
```

```
/* Grant permissions for this database */
```

```
GRANT SELECT ON TRANSACTIONS TO USER TRIAL_BALANCE
```

```
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON CUSTOMER TO ROLE SALES
```

```
GRANT SALES TO ACCOUNTANT
```

```
...
```

```
SQL>
```

Revoking multiple privileges

To remove some, but not all, of the access privileges assigned to a user or procedure, list the privileges to remove, separating them with commas. For example, the following statement removes the INSERT and UPDATE privileges on DEPARTMENT from a user, SERENA:

```
REVOKE INSERT, UPDATE ON DEPARTMENT FROM SERENA;
```

The next statement removes two privileges on the CUSTOMER table from a stored procedure, COUNT_BEANS:

```
REVOKE INSERT, DELETE ON CUSTOMER FROM PROCEDURE COUNT_BEANS;
```

Any combination of previously assigned SELECT, INSERT, UPDATE, DELETE or REFERENCES privileges can be revoked from a grantee if they were granted individually or in a list, but not if they were granted in the ALL package.

As with GRANT, the REVOKE ALL privilege combines the SELECT, INSERT, UPDATE, DELETE and REFERENCES privileges into a single expression. It will revoke any of these permissions from a grantee that had the privilege granted directly to its own name, rather than via a role or PUBLIC.

For example, the following statement revokes all directly granted SELECT, INSERT, UPDATE, DELETE and REFERENCES privileges on DEPARTMENT from a user named MAGPIE:

```
REVOKE ALL ON DEPARTMENTS FROM MAGPIE;
```



REVOKE ALL can be quite useful if you don't know what privileges a grantee has. It will not cause an exception if the grantee does not have the full set of privileges that ALL encompasses. It will not necessarily solve the problem of eliminating all of the permissions available to the user, because of the limits on what REVOKE ALL is capable of revoking.

What REVOKE ALL doesn't revoke

REVOKE ALL does not revoke

- the grantee's role membership or privileges the grantee acquired with role membership
- any privileges the grantee has by way of PUBLIC
- the grantee's EXECUTE privileges

Revoking the EXECUTE privilege

The syntax for revoking a grantee's EXECUTE privilege on a stored procedure has this syntax pattern:

```
REVOKE EXECUTE ON PROCEDURE <procedure-name>
FROM <grantee> [, <grantee> [, ...]]
| [TRIGGER <trigger-name> [, <trigger-name> [,...]]]
[PROCEDURE <procedure-name> [, <procedure-name> [, ...]]]
[VIEW <view-name> [, VIEW <view-name> [, ...]]];
```

The following statement removes the EXECUTE privilege from user HPOTTER on the procedure COUNT_CHICKENS:

```
REVOKE EXECUTE ON PROCEDURE COUNT_CHICKENS FROM HPOTTER;
```

Revoking from grantees

Now, we take a look at how grantees—the objective of the FROM clause in the REVOKE statement—can be bundled for removing privileges in bulk.

From a list of grantees

Use a comma-separated list of grantees to bulk-remove privileges from a number of users in a single statement. The following statement revokes INSERT and UPDATE permissions on DEPARTMENT from three grantees in one hit:

```
REVOKE INSERT, UPDATE ON DEPARTMENTS FROM MAGPIE, BRUNHILDE, KILROY;
```

From a role

Revoking privileges granted to a role removes the encompassed privileges from any grantees having that role as a privilege:

```
REVOKE UPDATE ON DEPARTMENT FROM /* ROLE */ CARTEBLANCHE;
```

Now, users who were granted the CARTEBLANCHE role no longer have the UPDATE privilege on DEPARTMENT; but they retain other privileges—SELECT, INSERT, DELETE, REFERENCES, EXECUTE—that they might have inherited from their membership of CARTEBLANCHE.

You can use a single statement to revoke the same privileges from one or more roles:

```
REVOKE DELETE, INSERT ON DEPARTMENT
FROM /* ROLE */ CARTEBLANCHE, /* ROLE */ MAITRE_D;
```

From a role-user

Revoking membership from a role that has been assigned to a grantee denies the grantee all of the permissions that it acquired through the role. Use REVOKE to remove a role that you assigned to users. The following statement revokes the CARTEBLANCHE role from KILROY:

```
REVOKE CARTEBLANCHE FROM KILROY;
```

KILROY no longer has any of the access privileges acquired as a result of membership in the role. However, other grantees who acquired the same privileges through membership of the role are unaffected.

From objects

To revoke privileges from one or more procedures, triggers, or views, include the appropriate keyword (PROCEDURE, TRIGGER, VIEW) before the name of the grantee object.

You can revoke the same privilege from different types of grantee objects in one statement by making a separate comma-separated list for each object type. In that case, just start each list with the object type keyword.

The following statement revokes INSERT and UPDATE privileges on the CUSTOMER table from two procedures and a trigger:

```
REVOKE INSERT, UPDATE ON CUSTOMER FROM
    PROCEDURE COUNT_CHICKENS, ABANDON_OLD
    TRIGGER AI_SALES ;
```



Multiple lists in a statement are not separated by commas but by the keyword for the next list type. In the example, the keyword TRIGGER signals the end of the list of procedures.

From user PUBLIC

To revoke privileges that multiple users are granted as user PUBLIC, just treat PUBLIC as any other grantee. For example, the following statement revokes INSERT and DELETE permissions on DEPARTMENT from all users that are encompassed by PUBLIC:

```
REVOKE SELECT, INSERT, UPDATE ON DEPARTMENT FROM PUBLIC;
```

Executing this statement leaves the table's owner and SYSDBA, as well as any stored procedures, views, triggers or roles that already had them, retaining INSERT and DELETE privileges on DEPARTMENT. Also, revoking privileges from PUBLIC does not strip privileges from users that have them in their own right.

Revoking grant authority

To revoke a user's grant authority for a given privilege, but keep the associated privilege, use REVOKE GRANT OPTION:

```
REVOKE GRANT OPTION FOR <privilege> [, <privilege> [...]]
    ON <table> | <object> FROM <user> ;
```

For example, the following statement revokes grant authority for the SELECT privilege on DEPARTMENT from a user, HPOTTER, he keeps his SELECT privilege:

```
REVOKE GRANT OPTION FOR SELECT ON DEPARTMENT FROM HPOTTER;
```

The execution of the statement will cascade down to any others users that have been granted grant authority by HPOTTER.

Security Scripts

If you've stayed with this chapter so far, no doubt you have reached the conclusion that implementing SQL privileges is one helluva lot of typing. It is quite obvious that doing this job interactively is a bad idea and not just because of all the typing. If we want to do it in a clean and organised fashion, we can use a good third-party privileges management tool. For those who love a tool with a graphical interface, there are plenty of them around. Most provide a utility that automates the creation of security scripts; some carry through and install the permissions directly for you.

We also have the option to write scripts—or, rather, we write stored procedures that write the scripts for us.

Generally, if we have a good privileges scheme worked out, we can generate a couple of scripts that load up our roles and general permissions exactly as we want them. Usually, a manually-written script is wanted for EXECUTE permissions, since it is not always simple to work up a formula for those.

The EXECUTE STATEMENT statement can be our friend for this task. It gets past Firebird's inability to execute DDL statements in PSQL, enabling permissions to be bulk-loaded directly to the database through a stored procedure. An example of such a procedure is listed below.

Creating a script

The author prefers to generate a security script. It can be tested and annotated, and it provides most of the documentation needed for QA and as a basis for custom deployments.

For the script, we might use an external file into which to output the script. However, the `permscript` procedure listed here is designed to run in *isql* and to pass its output to a text file from there.

Procedure permscript

Listing 37.1 Procedure to generate permissions script

```
/* (c) Helen Borrie 2004-2011, free for use and modification
   under the Initial Developer's Public License */
SET TERM ^;
CREATE PROCEDURE PERMSCRIPT (
    CMD VARCHAR(6),                /* enter 'G' or 'R' */
    PRIV CHAR(10),                /* a privilege, or 'ALL' or 'ANY' */
    USR VARCHAR(31),              /* a username */
    ROLENAME VARCHAR(31),         /* a role, existing or not */
    GRANTOPT SMALLINT,           /* 1 for 'WITH GRANT [ADMIN] OPTION' */
    CREATE_ROLE SMALLINT)        /* 1 to create new role ROLENAME */
RETURNS (PERM VARCHAR(80)) /* a permission statement, theoretically */
AS
    DECLARE VARIABLE RELNAME VARCHAR(31); /* for a table or view name */
    DECLARE VARIABLE STRING VARCHAR(80) = ''; /* used in proc */
    DECLARE VARIABLE STUB VARCHAR(60) = ''; /* used in proc */
    DECLARE VARIABLE VUSR VARCHAR(31); /* username for 'TO' or 'FROM' */
    DECLARE VARIABLE COMMENTS CHAR(20) = '/* */';
BEGIN
    /* Necessary for some UI editors */
    IF (ROLENAME = '') THEN ROLENAME = NULL;
    IF (USR = '') THEN USR = NULL;
    IF (PRIV = '') THEN PRIV = NULL;
    /* Not enough data to do anything with */
    IF ((PRIV IS NULL AND ROLENAME IS NULL) OR USR IS NULL) THEN EXIT;
```



```

/* If there's a rolename, we'll do stuff with it */
IF (ROLENAME IS NOT NULL) THEN
BEGIN
  /* If a role name is supplied, create the role if requested */
  IF (CREATE_ROLE = 1) THEN
  BEGIN
    PERM = 'CREATE ROLE '||ROLENAME||';';
    SUSPEND;
    PERM = 'COMMIT;';
    SUSPEND;
    PERM = COMMENTS;
    SUSPEND;
  END
  VUSR = ROLENAME;
END
/* If there's a rolename, we'll apply the permissions to the role
and grant the role to the supplied user */
ELSE
  /* We are not interested in the role: permissions are just for user */
  VUSR = USR;
/* Decide whether it's a GRANT or a REVOKE script */
IF (CMD STARTING WITH 'G') THEN
  STUB = 'GRANT ';
ELSE
  STUB = 'REVOKE ';
IF (ROLENAME IS NOT NULL) THEN
BEGIN
  IF (STUB = 'GRANT') THEN
  BEGIN
    /* Grant the role to the user */
    STRING = STUB||ROLENAME||' TO '||USR;
    IF (GRANTOPT = 1) THEN
      STRING = STRING||' WITH ADMIN OPTION ';
    END
  ELSE
    STRING = STUB||ROLENAME||' FROM '||USR||';';
    PERM = STRING;
    SUSPEND;
    PERM = COMMENTS;
    SUSPEND;
  END
  /* If ANY was passed in as privilege, create all perms separately */
  IF (PRIV = 'ANY') THEN
    STUB = STUB||'SELECT,DELETE,INSERT,UPDATE,REFERENCES ON ';
  ELSE
    STUB = STUB||PRIV||' ON ';
  /* Cycle through the table and view names and create a statement for each */

```

```

FOR SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
INTO :RELNAME DO
BEGIN
    STRING = STUB||:RELNAME||' ';
    IF (CMD STARTING WITH 'G') THEN
        STRING = STRING||'TO ';
    ELSE
        STRING = STRING||'FROM ';
    STRING = STRING||VUSR;
    IF (CMD STARTING WITH 'G'
        AND GRANTOPT = 1 AND ROLENAME IS NULL) THEN
        STRING = STRING||' WITH GRANT OPTION ';
    ELSE
        STRING = STRING||' ';
    PERM = STRING;
    SUSPEND;
END
PERM = COMMENTS;
SUSPEND;
END ^
SET TERM ;^

```

Creating and running the script

Go to the Firebird `/bin` directory and start *isql* under the SYSDBA log-in, connecting to the database. We will use the procedure to make a script that adds a role 'MANDRAKE', grants the role to user USER1 and then sets up the permissions for the role. Then it will do the same thing again with an existing role 'PURPLE' for user USER2:

```

SQL> OUTPUT L:\DATA\EXAMPLES\PERMSCRIPT.SQL;
SQL> SELECT * FROM PERMSCRIPT ('G', 'ALL', 'USER1', 'MANDRAKE', 1, 1);
SQL> COMMIT;
SQL> SELECT * FROM PERMSCRIPT ('G', 'ALL', 'USER2', 'PURPLE', 1, 0);
SQL> COMMIT;
SQL> OUTPUT;
SQL> INPUT L:\DATA\EXAMPLES\PERMSCRIPT.SQL;
SQL> COMMIT;
SQL> SHOW GRANT;

```

That's all there is to it. You will get an error message when the INPUT utility encounters the non-SQL “window-dressing” printed by OUTPUT but it won't interfere with writing the permissions.

Installing perms directly from a procedure

The procedure `grant_perms` is basically the same procedure. Instead of producing a set of output lines for *isql* to run as a script, it actually installs the permissions directly, via `EXECUTE STATEMENT`.¹

1. This technique is not suitable for Firebird 1.0.x, which does not support `EXECUTE STATEMENT`.

Procedure grant_perms

Listing 37.2 Permissions procedure

```

/* (c) Helen Borrie 2004-2011, free for use and modification
   under the Initial Developer's Public License */
SET TERM ^;
CREATE PROCEDURE GRANT_PERMS
  (CMD VARCHAR(6),
   PRIV CHAR(10),
   USR VARCHAR(31),
   ROLENAM VARCHAR(31),
   GRANTOPT SMALLINT)
AS
  DECLARE VARIABLE RELNAME VARCHAR(31);
  DECLARE VARIABLE EXESTRING VARCHAR(1024) = '';
  DECLARE VARIABLE EXESTUB VARCHAR(1024) = '';
BEGIN
  IF (ROLENAM = '') THEN ROLENAM = NULL;
  IF (USR = '') THEN USR = NULL;
  IF (PRIV = '') THEN PRIV = NULL;
  IF ((PRIV IS NULL AND ROLENAM IS NULL) OR USR IS NULL) THEN EXIT;
  IF (CMD STARTING WITH 'G') THEN
    EXESTUB = 'GRANT ';
  ELSE
    EXESTUB = 'REVOKE ';
  IF (ROLENAM IS NOT NULL) THEN
    BEGIN
      IF (EXESTUB = 'GRANT') THEN
        BEGIN
          EXESTUB = EXESTUB||ROLENAM||' TO '||USR;
          IF (GRANTOPT = 1) THEN
            EXESTUB = EXESTUB||' WITH ADMIN OPTION';
        END
      ELSE
        EXESTUB = EXESTUB||ROLENAM||' FROM '||USR;
    EXECUTE STATEMENT EXESTUB;
  END
  ELSE
    BEGIN
      IF (PRIV = 'ANY') THEN
        EXESTUB = EXESTUB||'SELECT,DELETE,INSERT,UPDATE,REFERENCES ON ';
      ELSE
        EXESTUB = EXESTUB||PRIV||' ON ';
      FOR SELECT RDB$RELATION_NAME FROM RDB$RELATIONS
        WHERE RDB$RELATION_NAME NOT STARTING WITH 'RDB$'
        INTO :RELNAME DO

```

```

BEGIN
  EXESTRING = EXESTUB||':RELNAME||' ';
  IF (CMD STARTING WITH 'G') THEN
    EXESTRING = EXESTRING||'TO ';
  ELSE
    EXESTRING = EXESTRING||'FROM ';
  EXESTRING = EXESTRING||USR;
  IF (GRANTOPT = 1) THEN
    EXESTRING = EXESTRING||' WITH GRANT OPTION';
  EXECUTE STATEMENT EXESTRING;
END
END
END ^
SET TERM ^

```

A Trick to Beat Idiot Users and Bad Guys

Shared by Pavel Cisar.

Here is a technique that Pavel uses to work around the lack of in-built SQL privilege protection for Firebird metadata. It will defeat idiot users who are tempted to avoid DDL and change metadata by messing with the system tables. It also beats malicious attempts to break your metadata with a script.

Although it appears that PUBLIC has ALL access to the system tables, it is just a weird backdoor to SQL rights management that could be easily fixed. You can restrict access to the system tables, just as to any other table in the database, by granting and revoking permissions. However, rights have to be *granted* in order to be *revoked*.

All we need is to set access management for the system tables to the standard routine by executing a series of **GRANT ALL ON <system-table> TO PUBLIC** statements. After that, we can revoke rights at will.

I don't know the exact technical explanation for why it works the way it does, but here's my guess. The GRANT statement creates an access control list (ACL) for the system table, which is checked by the code. Some branch in the code assumes "grant all to public if no ACL is found for a system table, else use the ACL".

It is not a "one size fits all" solution. Keep in mind that

- this setup is a workaround. It doesn't survive a restore from backup, so write a script that you can use each time you kick a freshly-restored database into action.
- you need to be careful about removing SELECT rights from PUBLIC on certain system tables, as the API functions `isc_blob_lookup_desc()` and `isc_array_lookup_bounds()` depend on its being available. Some tools and libraries may depend on it as well.
- removing write rights to system tables doesn't restrict the ability to update them the right and proper way, through DDL commands. Only fiddling directly with the system tables is prevented.

CHAPTER

38

MONITORING AND LOGGING FEATURES

Firebird provides a range of utility features for monitoring the activity of servers and databases:

- monitoring of database activity ([page 765](#))
- running traces ([page 769](#))
- collecting database statistics ([page 780](#))
- monitoring object locks ([page 791](#))

Monitoring Database Activity

v.2.1 and higher, ODS 11.1+

Firebird 2.1 introduced the ability to monitor server-side activity happening inside a particular database. The engine delivers a set of so-called “virtual” tables on demand from a database of ODS 11.1 or higher, providing snapshots of the current activity within a database.

Virtual monitoring tables exist only in ODS 11.1 (and higher) databases. While a Firebird 2.1 or higher server can attach to a database with a lower ODS, the internal structures of legacy databases do not have the metadata for generating the virtual tables that the monitoring routines populate. A full backup/restore migration, as described in Chapter 5, **Migration Notes**, is required in order to upgrade the ODS and thus enable this feature.

How Monitoring Works

The monitoring feature delivers an activity snapshot representing the current state of the database. Outputs comprise a variety of information about the database itself, active attachments and users, transactions, prepared and running statements, and much more.

The tables are “virtual” insofar as no data are materialised in them until explicitly requested. Metadata for the virtual tables is persistent, however, from the schema of the databases. Relation names for these tables all begin with “MON\$”.

An activity snapshot is isolated inside one transaction. It is created as soon as the first SELECT.. request on any MON\$ table is received and it is preserved until that transaction ends. Even if you sent your request in a READ COMMITTED or SNAPSHOT (“concurrency”) transaction, internally the transaction's isolation is escalated to a level that behaves like a SNAPSHOT TABLE STABILITY (“consistency”) transaction. In this way, multi-table queries such as master-detail joins and those reading dependent data via subqueries, will maintain a consistent view of database state.

Refreshing this stable snapshot requires ending the transaction and requering the MON\$ tables in a new transaction context.

Security and Scope

In v.2.1, only SYSDBA (and users with escalated privileges) and the database owner have access to what is happening in all attachments to the database. Regular users are restricted to the information about their own attachments only and cannot see any data from other clients.

Monitoring Multiple Connections

Initially, in the first relase and sub-release 2.1.1, an ordinary user could monitor only the CURRENT_CONNECTION. In higher sub-releases and major releases, that ordinary user can monitor all connections logged in under the same user name.

Access to the MON\$ tables is available not just in DSQL. The metadata definitions are available to PSQL as well, which means it is very feasible to delegate your frequently-used monitoring routines to stored procedures.



*For your convenience, the structures of the MON\$ tables are described in Appendix V, **System Tables**.*

Using MON\$

A valid database connection is required in order to retrieve the monitoring data. Although creation of a snapshot is usually quite a fast operation, some delay might be expected under high load, on a Classic server particularly.

The monitoring tables materialised in the attached database return information about that database only: there is no way at this point in the implementation for one database to garner MON\$ data from other databases. If the server is serving multiple databases, each one has to be connected to and monitored separately.



“Monitoring tables” is a bit of a mouthful, especially if English isn’t your native language! “MON-dollar” will do just fine.

Excluding the “Me” Connection

Usually you will want to exclude the MON\$ connection and its own activities from the information you want to collect and analyse. The caller of the MON\$ process can, of course, uniquely identify itself through the context variables CURRENT_CONNECTION and CURRENT_TRANSACTION. Since the same IDs will show up in the corresponding MON\$ tables, the monitoring transaction can therefore use them to establish keys for excluding information about itself and its own connection.

Character Set of “Platform” Columns

From v.2.5, the handling of file specification strings and some other character parameter items in the DPB are reflected in a change to the character set of the related columns in the MON\$ tables, which are defined by the system domain RDB\$FILE_NAME2. The character set of that domain definition changes at v.2.5, from NONE to UNICODE_FSS.

The columns affected by the character set change are MON\$DATABASE_NAME, MON\$ATTACHMENT_NAME and MON\$REMOTE_PROCESS.

Usage Examples

To give a feeling for the information you can retrieve, the following are some simple examples to get started with.

An Attachment Query

Suppose you want to retrieve the process IDs of all of the Classic server instances currently consuming CPU resources. We have a field in MON\$ATTACHMENTS that can tell us which processes are busy and which are idle:

```
SELECT MON$SERVER_PID
      FROM MON$ATTACHMENTS
     WHERE MON$ATTACHMENT_ID <> CURRENT_CONNECTION -- exclude Me
     AND MON$STATE = 1
```

Who is Doing What from Where?

You might want to know what clients are connected to the database and, if they are connected through a v.2.1 or higher client version, their network address and what applications they are using:

```
SELECT
      MON$USER,
      MON$REMOTE_ADDRESS,
      MON$REMOTE_PID,
      MON$TIMESTAMP
     FROM MON$ATTACHMENTS
     WHERE MON$ATTACHMENT_ID <> CURRENT_CONNECTION -- exclude Me
```

Clients that are v.2.0.x or lower cannot provide information about the application or their network address. The relevant fields will be NULL. In a different query you might like to use that situation to find out how many of the currently connected clients are using old versions of the client library! The time the connection started (MON\$TIMESTAMP) and the user name and/or role name might help with this kind of forensic search.

What is My Isolation Level?

This time, instead of excluding the Me transaction, you explicitly want to know something about it:

```
SELECT MON$ISOLATION_MODE
      FROM MON$TRANSACTIONS
     WHERE MON$TRANSACTION_ID = CURRENT_TRANSACTION
```

What Statements are Active?

This query is a little more adventurous. Two MON\$ tables are joined to extract quite detailed information about what statements are running and whose workstations requested them:

```
SELECT ATT.MON$USER,
       ATT.MON$REMOTE_ADDRESS,
       STMT.MON$SQL_TEXT,
       STMT.MON$TIMESTAMP
FROM MON$ATTACHMENTS ATT
     JOIN MON$STATEMENTS STMT
       ON ATT.MON$ATTACHMENT_ID = STMT.MON$ATTACHMENT_ID
WHERE ATT.MON$ATTACHMENT_ID <> CURRENT_CONNECTION -- exclude Me
      AND STMT.MON$STATE = 1
```

What's Happening with Executable Code on the Server?

You can use a common table expression (CTE) to do a comprehensive query on all the PSQL modules that are executing:

```
WITH RECURSIVE HEAD AS
(
  SELECT CALL.MON$STATEMENT_ID,
         CALL.MON$CALL_ID,
         CALL.MON$OBJECT_NAME,
         CALL.MON$OBJECT_TYPE
  FROM MON$CALL_STACK CALL
       WHERE CALL.MON$CALLER_ID IS NULL
  UNION ALL
  SELECT CALL.MON$STATEMENT_ID,
         CALL.MON$CALL_ID,
         CALL.MON$OBJECT_NAME,
         CALL.MON$OBJECT_TYPE
  FROM MON$CALL_STACK CALL
       JOIN HEAD
         ON CALL.MON$CALLER_ID = HEAD.MON$CALL_ID
)
SELECT MON$ATTACHMENT_ID,
       MON$OBJECT_NAME,
       MON$OBJECT_TYPE
FROM HEAD
     JOIN MON$STATEMENTS STMT
       ON STMT.MON$STATEMENT_ID = HEAD.MON$STATEMENT_ID
WHERE STMT.MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```

Cancelling a Running Query

The MON\$ system in ODS 11.1 and higher databases enables problem queries to be terminated from a separate, dedicated connection, i.e., one that is not engaged in any regular client activities. The SYSDBA or the database owner can make use of data available in the monitoring tables to isolate the offending query and devise an appropriate mechanism for reining it in.

Example As a very rough example, the following statement will kill all statements currently running in the database, other than any that belong to the Me connection:

```
delete from mon$statements
where mon$attachment_id <> current_connection
```

Naturally, this won't normally be what you want to do! Study the metadata carefully, along with the data you are seeing through MON\$. You will be able to figure out the most effective ways to isolate the problem queries in your system and target them for cancellation.



The Firebird 2.5 API introduced the `fb_cancel_operation` function to enable intervention through a client function call.

Cancelling an Entire Session

In versions 2.5 and higher, it is also possible to cancel all of the transactions and statements for an entire session, including the client attachment itself. Having obtained the attachment ID you can “zap” that session from another privileged session, e.g.,

```
delete from mon$attachments
where mon$attachment_id = <the attachment ID>
```

The MON\$ Tables

The full metadata for the MON\$ tables can be found in Appendix V, **System Tables**, in the section **Monitoring Tables** on page 942.

Trace and Audit Services

v.2.5 and higher

Trace and audit facilities, available from v.2.5 onward, enable tracing and logging of various events performed inside the engine, such as statement execution, connections, disconnections, and so on. Logs can be collated and analysed against performance, virtually in real time.

Modes of Use

There are three general ways that traces can be employed. In summary, they are

- Constant auditing of engine activity through *system audit tracing*
- *Interactive tracing* on an as-required basis
- Capturing engine activity figures into files, over time, for retrospective analysis

System Audit Tracing

System audit tracing is performed at the behest of an Administrator, who kicks off the trace by configuring a “recipe” for it in a file and pointing the **AuditTraceConfigFile** parameter in `firebird.conf` to that file.



To make audit configuration changes known to the engine, Firebird must be restarted.

The Administrator can subsequently suspend, resume or stop this session without needing to restart Firebird.

Interactive trace on demand

On-demand, interactive tracing can be effected in one or more databases, for all activities or for specified activities. An application (which could be the *fbtracemgr* utility) starts a user trace session, reads its output and shows traced events to the user in real time on the screen. The user can suspend and resume the trace and, finally, stop it.

Collecting engine activity over time

Engine activity over a meaningful period of time (a few hours or perhaps even a whole day) can be collected for later analysis.

An application starts a user trace session, reading the trace output regularly and saving it to one or more files. Multiple user trace sessions can be running concurrently. A user trace session must be stopped manually, although not necessarily by the same application that started it.



If multiple trace sessions are running, a listing can be requested in order to identify the session of interest.

Trace Sessions

A trace takes place in the context of a trace session. Each trace session has its own configuration, state and output. The Firebird engine has a fixed list of events it can trace. How the engine forms the list of events for a session depends on which events are included in the configuration file. Some events depend on which sort of trace is requested—system or user.

Every trace session is assigned a unique session ID. When any trace session begins, the Services Manager outputs this ID as the message:

Trace session ID nnnn started

where nnnn is the ID, of course.



The same configuration file can be used for simultaneous user trace sessions. Only one system audit trace session can run.

Trace Scope on Windows

On Windows, the scope of tracing is restricted to just processes that are accessible from the current Windows session. The reason for this restriction is that, on this platform, the trace routines exhibit shared memory conflicts if multiple engine instances in different Windows sessions are allowed to run traces in global namespace.

Configuring a Trace or Audit

System audit traces and user traces are configured by way of a configuration file, a text file that you create for a specific trace that you want to do. You might have just one file that you always use or you could have several, each configured to do trace sessions over a different set of parameters. Both system audits and user trace sessions use the same format and are processed the same way, although they differ in aspects of set-up and control.

Table 38.1 System audit vs User trace

System audit	User trace
Started by the engine automatically if enabled in <code>firebird.conf</code>	Started by an explicit user request
Output is sent to a permanent log file	Output goes to a temporary file that is deleted automatically when read by a client

fbtrace.conf—your template

The template file `fbtrace.conf`, found in Firebird's root directory, contains the full list of available events, with format, rules and syntax for composing a system audit or user trace configuration file. The template file contains a large amount of commented text explaining the purpose and syntax of each entry. You can just make a copy of this template and edit it to suit your needs.

A trace configuration can have two sections: one specifying a *database* for that trace and the other specifying *services* you want to include or exclude from tracing. The file will be processed from the top down, so be sure to configure a database for tracing services that are database-specific.

Separate sections are used to configure database[s] and services—not more than one of each kind.

Database section

The database section is bounded by the markers

```
<database expression>
...
</database>
```

The *expression* argument can be the database file name without the path, in which case the trace configuration file needs to be in the same location as the database, or a regular expression that will find the named database file in any directory, e.g.,

```
<database %[\]\]mydatabase.fdb>
    enabled true
</database>
```

A regular expression can specify multiple databases, e.g.,

```
<database %[\]\](mydatabase|yourdatabase|ourdatabase).fdb>
    enabled true
    log_filename \friday.log
</database>
```

The specified parameters within the database section will be applied to all of the listed databases.

Parameters

```
enabled true | false
```

Indicates whether database events are to be traced. Default is false (no tracing)

```
log_filename file-path
```

Used for the *system audit trace only*, provides the file path for the operations log. Default is blank. If it is left blank and the system audit is enabled, the log will be written to a file named `default_trace.log`, usually in the current directory of Firebird process, viz.,

- On POSIX, *cwd* been set explicitly by Firebird.
- On Windows, it will be the path pointed to by `lpCurrentDirectory`, normally the location of the Firebird executable.

The *sed* syntax for substitutions is supported, viz.,

```
\0 - whole matched string
\1 ... \9 - parenthesis subexpressions
\\ is used for a literal backslash
```

max_log_size *n*

Maximum size of log file in MB. Used by the system audit trace for rotating the log. When the current log file reaches this limit it is renamed using current date and time and a new log file is created. A value of zero (the default) means that the size of the log file is unlimited and rotation will not occur.

include_filter [*expression*]

and

exclude_filter [*expression*]

These optional SQL query filters take regular expressions to specify statements to be matched for specific inclusion in or exclusion from the output.

log_connections false | true

Include attach and detach requests in the output. The default (false) excludes them.

connection_id *n*

Trace only the connection ID provided by *n*. If *n* is zero (the default), all connections are traced.

log_transactions false | true

Record *transaction start* and *transaction end* events. Default is *false*.

log_statement_prepare false | true

Record statement *prepare* events. Default is *false*.

log_statement_free false | true

Record SQL statement *free* events. Default is *false*.

log_statement_start false | true

Record statement *execution start* events. Default is *false*.

log_statement_finish false | true

Record statement *execution finish* or *fetch to EOF* events. Default is *false*.

`log_procedure_start false | true`

Record when stored procedures start executing. Default is *false*.

`log_procedure_finish false | true`

Record when stored procedures finish executing. Default is *false*.

`log_trigger_start false | true`

Record when triggers start executing. Default is *false*.

`log_trigger_finish false | true`

Record when triggers finish executing. Default is *false*.

`log_context false | true`

Record when RDB\$SET_CONTEXT changes a context variable. Default is *false*.

`print_plan false | true`

Record the plan (access path) with SQL statements. Default is *false*.

`print_perf false | true`

Print detailed performance information when applicable. Default is *false*.

`log_blr_requests false | true`

Record BLR requests for compile and execute events. Default is *false*.

`print_blr false | true`

Print BLR requests. Default is *false*—don't print them.

`log_dyn_requests false | true`

Record execution of *dyn* requests. Default is *false*.

`print_dyn false | true`

Print *dyn* requests. Default is *false*—don't print them.

`time_threshold n`

Log an *xxx_finish* record only if its timing exceeds *n* milliseconds. Default is 100 milliseconds.

`max_sql_length n`

Maximum length of SQL string logged. The default is 300 bytes. Take care never to let *n* exceed 64 KB for a single event.

max_blr_length *n*

Maximum length of BLR request, if logged. Default is 500 bytes.

max_dyn_length *n*Maximum length of *dyn* request, if logged. Default is 500 bytes.**max_arg_length** *n*

Maximum length of any individual string argument logged. Default is 80 bytes.

max_arg_count *n*

Maximum number of query arguments to record in the log. Default is 30.

Services section

The Services filters can be applied to any of the following services, whose names are listed here. As Firebird development proceeds, check the commented text in the newest `fbtrace.conf` template for any additions to this list.

- Backup Database
- Restore Database
- Repair Database
- Add User
- Delete User
- Modify User
- Display User
- Database Properties
- Database Stats
- Get Log File
- Incremental Backup Database
- Incremental Restore Database
- Start Trace Session
- Stop Trace Session
- Suspend Trace Session
- Resume Trace Session
- List Trace Sessions
- Set Domain Admins Mapping to RDB\$ADMIN
- Drop Domain Admins Mapping to RDB\$ADMIN

The default Services section is bounded by the markers

```
<services>
...
</services>
```

Parameters

`enabled` `false` | `true`

Enables or disables Services tracing. Default is *false*.

`log_filename` *file-path*

Described above in the [Database section](#).

`max_log_size` *n*

Described above in the [Database section](#).

`include_filter` [*expression*]

Use a regular expression with this parameter to explicitly record only the services whose names match the expression.

`exclude_filter` [*expression*]

Use a regular expression with this parameter to exclude the services whose names match the expression.

`log_services` `false` | `true`

Record service attach, detach and start events. Default is *false*.

`log_service_query` `false` | `true`

Record service query events. Default is *false*.



Tracing was new in v.2.5. As sub-releases and releases progress, watch for new events and facilities that will be added from time to time to extend the tracing capabilities.

The System Audit Session

A system audit session is started by the engine when it starts up, as long as the configuration parameter ***AuditTraceConfigFile*** in `firebird.conf` is not commented out and is not empty. There can be at most one system audit trace in progress. In the *trace configuration file* whose location and name are pointed to by ***AuditTraceConfigFile***, it looks for a recipe to determine which events the session is interested in.



*By default, the value of the ***AuditTraceConfigFile*** parameter is empty, indicating that no system audit tracing is configured.*

What's in a trace configuration file?

A configuration file contains list of traced events and points to the placement of the trace log(s) for each event. It is sufficiently flexible to allow different sets of events for different databases to be logged to separate log files.

User Trace Sessions

A user trace session is managed by a user, using calls to the Services API. There are five new service functions for this purpose:

- `start: isc_action_svc_trace_start`

- stop: `isc_action_svc_trace_stop`
- suspend: `isc_action_svc_trace_suspend`
- resume: `isc_action_svc_trace_resume`
- list all known trace sessions: `isc_action_svc_trace_list`

User Interfaces

Implementation of the service function calls can be expected in your favourite host language interface, so that you can “roll your own” utilities for running user trace sessions. We do not discuss the API function calls, *per se*, in this volume. However, later in this topic, you can read about the *fbtracemgr* command-line tool that is distributed with Firebird 2.5 and higher kits, for working interactively with the trace services. It has its own syntax of switches and parameters.

Another way to manage user trace sessions is by using the general Services Manager command-line utility, *fbsvcmgr*, which lets you pass services function calls, including the `isc_action_svc_trace_*` functions, directly across the API. Some examples are provided at the end of this topic.

You might also like to search for third-party tools that already surface the trace features in self-contained client tools. The best-known example at the time of this writing is **Fb Trace Manager**, by [Upscene Productions](#).

Workings of a User Trace Session

When a user application starts a trace session, it sets a session name (optional) and the path to the session configuration (mandatory). The session configuration is a text file. Apart from the lines relating to placement of the session output, it uses the same rules and syntax as the `fbtrace.conf` template that is in Firebird's root directory.

There is no default location or file name that the tracing process falls back to if the application fails to supply the path. The application must supply the mechanism for finding the configuration file.



*An example of such a mechanism is the saved-file parameter, `trc_cfg`, that the command-line Services Manager utility **fbsvcmgr** provides as a symbol for the path to the configuration file. An example of its usage appears at the end of this topic.*

Once the user trace session has been started by the application, the application has to read its output, using calls to `isc_service_query()`. The output of a session is stored in a set of temporary files, each of 1MB, up to a (default) maximum of 10 files.

It is the responsibility of the user application to provide a mechanism to capture the output and, if needed, to save it to a file system structure to which it has unconditional access.

Once a file has been completely read by the application, it is automatically deleted. The maximum number of files, as an aggregate total, can be configured to a smaller or larger limit using the ***MaxUserTraceLogSize*** in `firebird.conf`.

The service could be generating output faster than the application can read it. If the total size of the output reaches the ***MaxUserTraceLogSize*** limit, the engine automatically suspends the trace session and waits for the application to finish reading a file. Once the application has finished reading that file, it is deleted, capacity is returned and the engine resumes the trace session automatically.

At the point where the application decides to stop its trace session, it simply requests a detach from the service. Alternatively, the application can use the `isc_action_svc_trace_*` functions to suspend, resume or stop the trace session at will.

Who Can Manage Trace Sessions?

Any user can initiate and manage a trace session:

- An ordinary user can request a trace only on its own connections and cannot manage trace sessions started by any other users.
- Administrators can manage any trace session.

Abnormal Endings

Be aware that user trace sessions do not survive in the event of Firebird “super” processes being stopped. If a Superserver or Superclassic process is shut down, any user trace sessions that were in progress, including any that were awaiting a *resume* condition, are destroyed and a *resume* cannot restart them.



This situation doesn't apply to the Classic server, of course, since each connection involves its own dedicated server instance. No service instance can outlive the connection that instigated it. Thus, there is no such thing as “shutting down” a Classic server instance.

User Trace Sample Configuration Texts

The following samples provide a reference for composing configuration texts for user trace sessions.

Example a Trace the prepares, “frees” and executions of all statements within connection 12345:

```
<database mydatabase.fdb>
  enabled           true
  connection_id     12345
  log_statement_prepare true
  log_statement_free  true
  log_statement_start true
  log_statement_finish true
  time_threshold    0
</database>
```

Example b Trace all connections of a given user to database `mydatabase.fdb`, logging executed INSERT, UPDATE and DELETE statements and nested calls to procedures and triggers, and show corresponding PLANS and performance statistics.

```
<database mydatabase.fdb>
  enabled           true
  include_filter     %(INSERT|UPDATE|DELETE)%
  log_statement_finish true
  log_procedure_finish true
  log_trigger_finish true
  print_plan         true
  print_perf         true
  time_threshold    0
</database>
```



Services events can be traced by name and targeted using include and exclude filters.

Command-line Utility *fbtracemgr*

The command-line utility named *fbtracemgr*, for working interactively with trace services, can be found in the `/bin/` directory of the Firebird 2.5 and higher installations. It has its own syntax of switches and parameters that are largely consistent with the other command-line utilities.

Usage is as follows.-

```
fbtracemgr <action> [<parameters>]
```

The available actions, with their parameters, are listed below in Table 38.2. Connection and action parameters are case-insensitive on all platforms and can be combined in any order.

Table 38.2 Action Switches for *fbtracemgr*

Action	Parameters	Description
-STA[RT]		Start trace session
-STO[P]		Stop trace session
-SU[SPEND]		Suspend trace session
-R[ESUME]		Resume trace session
-L[IST]		List existing trace sessions
Parameters for actions:	-I[D] <number>	Session ID, mandatory for all actions except LIST)
	-N[AME] <string>	Session name (optional if the Session ID is provided, but recommended for ease of use with LIST)
Parameters for connecting:	-C[ONFIG] <string>	Trace configuration file name. Argument <string> must be a valid file path specification for the platform.
	-SE[RVICE] <string>	Service name = service_mgr if local hostname:service_mgr if remote or localhost
	-U[SER] <string>	User name
	-P[ASSWORD] <string>	Password
	-FE[TCH] <string>	Fetch password from file. Argument <string> must be a valid file path specification for the platform.
	-T[RUSTED] <string>	Force trusted authentication.

Note *Strings in the configuration file are interpreted according to the platform character set and, basing the strings on UTF-8, subjected to platform rules such as treating file names as case-insensitive on Windows and case-sensitive on POSIX.*

Examples List all currently running user trace sessions on server `remote_host` and list to the remote console:

```
fbtracemgr -SE remote_host:service_mgr -USER SYSDBA -PASS masterkey -LIST
```

Start a user trace session on the local server, naming the session “my_trace”, using the recipe found in the file `my_cfg.txt`:

```
fbtracemgr -SE service_mgr -START -NAME my_trace -CONFIG my_cfg.txt
```

Suspend the user trace session on the local server that has the number 2:

```
fbtracemgr -SE service_mgr -SUSPEND -ID 2
```

Resume the user trace session on the local server that has the number 2:

```
fbtracemgr -SE service_mgr -RESUME -ID 2
```

Stop (terminate) the user trace session on the local server that has the number 4, deleting any output awaiting dispatch to the application:

```
fbtracemgr -SE service_mgr -STOP -ID 4
```



On any platform, press Ctrl-C to terminate the trace session you are currently interacting with.

Using *fbsvcmgr* to Submit Trace Service Requests

The general Services utility, *fbsvcmgr*, discussed in Chapter 40, is for submitting service function calls from the command line. The `isc_action_svc_trace_*` functions can be called through this utility.

Special symbols

The *fbsvcmgr* utility provides a set of symbolic names for use with the `isc_action_svc_trace_*` functions for introducing the arguments that it might need to pass into the service function call structure. They are:

<i>trc_id</i>	Trace Session ID (optional if the <code>trc_name</code> is provided; otherwise mandatory for all calls except <code>action_trace_list</code>)
<i>trc_name</i>	Trace Session Name (optional if the <code>trc_id</code> is provided; otherwise mandatory for all calls except <code>action_trace_list</code>)
<i>trc_cfg</i>	Trace configuration file name, used only with <code>action_trace_start</code> . The following argument must be a valid file path specification for the platform

Example 1 Start a user trace named “My trace” using a configuration file named `fbtrace.conf` and read its output on the screen:

```
fbsvcmgr service_mgr action_trace_start trc_name "My trace" trc_cfg fbtrace.conf
```

To stop this trace session, press Ctrl+C at the *fbsvcmgr* console prompt. (See also Example 5, below).

Example 2 List trace sessions:

```
fbsvcmgr service_mgr action_trace_list
```

Example 3 Suspend trace session with ID 1:

```
fbsvcmgr service_mgr action_trace_suspend trc_id 1
```

Example 4 Resume trace session with ID 1:

```
fbsvcmgr service_mgr action_trace_resume trc_id 1
```

Example 5 Stop trace session with ID 1:
`fbsvcmgr service_mgr action_trace_stop trc_id 1`



List sessions in another console, look for the ID of a session of interest and use it in the current console to stop the session.

Trace Plug-in Facilities

The Trace API provides a set of hooks which can be implemented as an external plug-in module to be called by the engine when any traced event happens. A plug-in assumes responsibility for logging such events in some custom way.

The Trace API that is in use in Firebird 2.5 will be changed in succeeding sub-releases. As at v.2.5.1, it was not officially published and should be regarded as unstable.

The implemented “standard” trace plug-in, `fbtrace.dll` (`.so`), resides in the `\plugins` folder of your Firebird 2.5 installation.

Collecting Database Statistics—*gstat* All versions

Firebird provides a command-line tool that displays real-time statistical reports about the state of objects in a database. The tool produces a number of reports about what is going on in a database, especially with indexes.

gstat command-line tool

You need to run *gstat* on the server machine since it is a completely local program that does not access databases as a client. Its default location is the `/bin` directory of your Firebird installation. It reports on a specified database and can be used by the SYSDBA or the owner of the database.

gstat on POSIX

Because *gstat* accesses database files at the filesystem level, it is necessary on Linux and UNIX platforms to have system-level read access to the file. You can achieve this in one of two ways, either:

- log in under the account which is running the server (by default, user *firebird* on v.1.5 and higher, root or *interbas* on v.1.0.x.
- or
- set the system-level permissions on the database file to include read permission for your group.

The *gstat* interface

Unlike some of the other command-line tools, *gstat* does not have its own shell interface. Each request involves calling *gstat* with switches.

Syntax
`gstat [switches] db_name`

or

```
gstat db_name [switches]
```

`db_name` is the fully qualified local path to the database you want to query.

Graphical tools

gstat is not user-friendly and some GUI tools do a neater job of reproducing *gstat*'s output, generating the equivalent of the *gstat* switches though the Services API (“Services Manager”). Snapshots below were taken using the open source IBOConsole utility.

gstat Switches

Table 38.3 Switches for *gstat*

Switch	Description
<code>-u[ser] <i>username</i></code>	Checks for user <i>username</i> before accessing database
<code>-p[assword] <i>password</i></code>	Checks for password <i>password</i> before accessing database
<code>-fetch</code>	Fetches password from file (POSIX only)
<code>-tr</code>	Use trusted user authentication (Windows only)
<code>-h[earer]</code>	Report the information on the header page, then stop reporting.
<code>-log</code>	Report the information on the header and log pages, then stop reporting
<code>-i[index]</code>	Retrieve and display statistics on indexes in the database. Analyses leaf pages of the indexes.
<code>-d[ata]</code>	Retrieve and display statistics on user data tables in the database
<code>-a[ll]</code>	This is the default report if you do not request <code>-index</code> , <code>-data</code> or <code>-all</code> . It retrieves and displays statistics from both <code>-index</code> and <code>-data</code>
<code>-s[ystem]</code>	As <code>-all</code> (above) but additionally includes statistics on the system tables
<code>-r</code>	Retrieves and displays record size and version statistics (including back versions)
<code>-t <i>table_list</i></code>	Used in conjunction with <code>-data</code> , restricts output to the tables listed in <i>table_list</i> . Table names are case-sensitive and must be typed in upper case unless they were defined in double-quoted identifiers. However, in versions below Firebird 2.0, <i>gstat</i> does not support this switch on databases that use quoted, lower case or mixed case table identifiers.
<code>-z</code>	Print product version of <i>gstat</i>

It is recommended that you pipe the output to a text file and view it with a scrolling text editor.



Because *gstat* performs its analysis at file level, it does not run in a transaction context. Consequently, index statistics include information about indexes involved in non-committed transactions. Do not be alarmed, for example, if a report shows some duplicate entries in a primary key index.

The **-index** switch

Syntax **gstat -i[index] db_path_and_name**

Retrieves and displays statistics about indexes in the database: average key length (bytes), total duplicates, and maximum duplicates for a single key. Include the **-s[ystem]** switch if you would like the system indexes in the report.

Unfortunately, there is no way to get the stats for a single index—but you can restrict the output to a single table using the **-t** switch followed by the name of the table. You can supply a space-separated list of table names to get output for more than one table.



*If your table names are case-sensitive—having been declared with double-quoted identifiers—then the t-switch argument[s] must be correct for case but not double-quoted. For tables with spaces in their names, **gstat -t** does not work at all.*

To run **gstat -i** over the **employee.fdb** database and pipe it to a text file named **gstat.index.txt** do as follows:

On POSIX, type (all in one line)

```
./gstat -index /opt/firebird/examples/employee.fdb -t CUSTOMER
-user SYSDBA -password masterkey > /opt/firebird/examples/gstat.index.txt
```

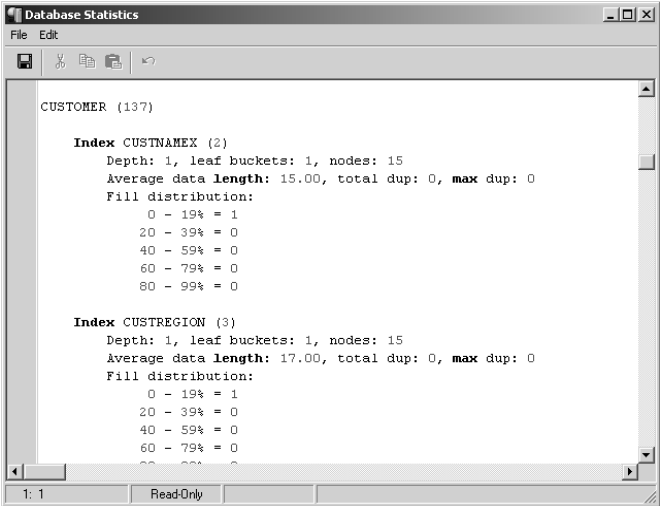
On Win32, type (all in one line)

```
gstat -index "c:\Program Files\Firebird\Firebird_1_5\examples\employee.fdb" -t CUSTOMER
-user SYSDBA -password masterkey > "c:\Program
Files\Firebird\Firebird_1_5\examples\gstat.index.txt"
```



The double quotes around the database path are required on Windows if you have spaces in the path name.

Figure 38.1 Example of index page summary output



What it all means

To begin with, a summary of the index information, line by line:

Table 38.4 gstat -i[index] output

Item	Description
Index	The name of the index
Depth	The number of levels of indirection in the index page tree. If the depth of the index page tree is greater than three, then record access through the index will not be as efficient as possible. To reduce the depth of the index page tree, increase the page size. If increasing the page size does not reduce the depth, then increase the page size again.
Leaf buckets	The number of lowest-level (leaf node) pages in the index tree. These are the pages that contain the actual record pointers. Higher level pages store indirection linkages.
Nodes	The total number of records indexed by in the tree. This should be equivalent to the number of indexed rows in the table, although gstat output may include nodes that have been deleted but are not yet garbage-collected. It may also include multiple entries for records that have had their index key modified.
Average data length	The average length of each key, in bytes. This is typically much shorter than the length of the declared key because of suffix and prefix compression.
Total dup	The total number of rows that share duplicate indexes.
Max dup	The number of duplicated nodes in the "chain" having the most duplicates. This will always be zero for unique indexes and a sign of poor selectivity if the number is high in proportion to the Total Dup figure.
Average fill	This is a histogram with five 20% "bands", each showing the number of index pages whose average fill percentage falls within that range. Fill percentage is the proportion of the space on each page that actually contains data. The sum of these numbers gives the total number of pages containing index data.

Because *gstat* performs its analysis at file level, it has no concept of transactions. Consequently, index statistics include information about indexes involved in non-committed transactions.

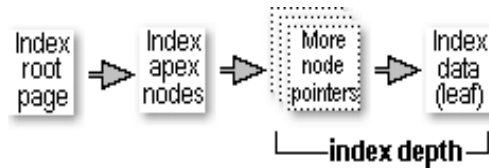
Index depth

The index is a tree structure, with pages at one level pointing to pages at the next level, and so on, down to the pages that point to actual rows. The more depth, the more levels of indirection. Leaf buckets are the bottom level pages of the index, which point to individual rows.

In the diagram below, the index root page (created when the database is created) stores a pointer for each index and an offset pointing to another page of pointers that contains

pointers for that index. In turn, that page points to the pages containing the leaf data—the actual data of the nodes—either directly (depth=1) or indirectly (adding one level for each level of indirection).

Figure 38.2 Index depth



Two factors affect depth: page size and key size. If the depth is larger than three and the page size is less than 8192, increasing the page size to 8192 or 16384 should reduce the levels of indirection and help the speed.



You can calculate the approximate size (in pages) of the Max dup chain from the stats data. Divide Nodes by Leaf buckets to get the number of nodes per page. Multiplying that result by Max dup gives the approximate number of pages.

Analysing some statistics

The following excerpts are *gstat -index* output from a badly-performing database. The first is the supporting index that was automatically created for a foreign key:

Example 1 Index RDB\$FOREIGN14 (3)
 Depth: 2, leaf buckets: 109, nodes: 73373
 Average data length: 0.00, total dup: 73372, max dup: 32351
 Fill distribution:
 80 - 99% = 109

Analysis **Depth: 2, leaf buckets: 109, nodes: 73373:** tells us that the bottom level of the index has 109 buckets (pages), for an index of 73,373 nodes. That may not be the total row count of the table. For one thing, *gstat* is not aware of transactions, so it cannot tell whether it is looking at committed or uncommitted pages. For another, the column may have nulls and they are not considered in the statistics.

The bottom level of the index—where leaf nodes are stored—has only 109 pages in it. That seems suspiciously few pages for so many rows. The next statistic tells us why.

Average data length: 0.00, total dup: 73372, max dup: 32351: “max dup” is the length of the longest duplicate chain, accounting for almost half of the nodes. The “total dup” figure says that every node but one is duplicated.

This is a fairly classic case where the designer applies a foreign key without considering its distribution. It is probably a Boolean-style column, or perhaps a lookup table with either very few possible values or a practical bias toward a single value.

An example of this was an electoral register application which stored a column for country of residence. The electorate was approximately three million voters and registration was compulsory. The database supplied a COUNTRY table of more than 300 countries, keyed by the CCCIT country code. It was present on nearly every table in the database as a foreign key. The trouble was, nearly every elector lived in the same country.

The average data length is the average length of the key as stored. This has very little to do with the length as defined. An average length of zero only means that, by a process of suppression and compression, there is not enough “meat” left on which to calculate an average.

Fill distribution: shows that all 109 pages are in the 80 – 99% range: well-filled. The fill distribution is the proportion of space in each page of the index that is being used for data and pointers. 80-99% is good.



Lower fill distributions are the strongest clue that you should rebuild the index.

Example 2 The next example shows the statistics for the system generated index for the primary key of the same table:

```
Index RDB$PRIMARY10 (0)
Depth: 3, leaf buckets: 298, nodes: 73373
Average data length: 10.00, total dup: 0, max dup: 0
Fill distribution:
0 - 19% = 1
80 - 99% = 297
```

Analysis The **key length** of 10 indicates that some compression is done. This is normal and good. That one page is under-filled is quite normal—the number of nodes did not fit evenly into the pages.

Example 3 This database has a smaller table that holds data temporarily for validation. It is flushed and reloaded periodically. The following statistics were generated for a foreign key on that table:

```
Index RDB$FOREIGN263 (1)
Depth: 1, leaf buckets: 10, nodes: 481
Average data length: 0.00, total dup: 480, max dup: 480
Fill distribution:
0 - 19% = 10
```

Analysis **Total dup** and **Max dup** are identical—every row has the same value in the index key. Selectivity does not get worse than that. The fill level is very low in all pages, suggesting spotty deletions. If this were not such a small table, it would be an index from Hell.

The table—a work-in-progress queue—is very dynamic, storing up to 1000 new rows per day. After validation, the rows are moved to production tables and the rows in the WIP table are being deleted, causing the system to slow down. Frequent backup and restore of the database is necessary to get things moving.

Solutions

The problem foreign keys should have been avoided. In such cases, if referential integrity constraints are deemed essential, they can be user-defined in triggers.

However, if database design absolutely requires foreign key constraints for volatile tables on columns with low selectivity, there are recommended ways to mitigate the overhead and index page deterioration caused by constantly deleting and re-populating the table. Monitor the fill level on the problem indexes and take action whenever it drops below 40 per cent. The choice of action depends on requirements:

- First choice, if possible, is to delete all the rows at once rather than deleting them one by one, at random. Drop the foreign key constraint, delete the rows and commit that transaction. Recreate the constraint. As long as there are no long-running transactions inhibiting garbage collection, the new index will be completely empty.

- If the deletions must be incremental, to choose a time to get exclusive access and use ALTER INDEX to rebuild the index. It will be faster and more predictable than incremental garbage collection on huge duplicate chains.

Other gstat switches

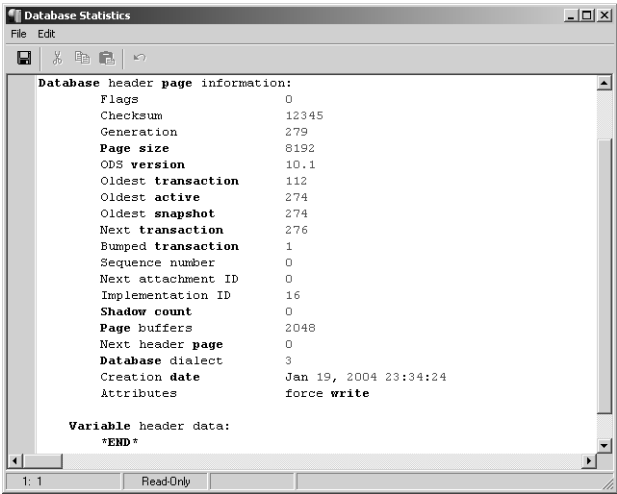
The gstat statistics can provide useful information about other activities in databases.

The *-header switch*

```
gstat -header db_path_and_name
```

Displays a database summary showing the header page information.

Figure 38.3 Example of gstat header page summary output



The first line displays the name and location of the primary database file. The following lines contain information from the database header page:

Table 38.5 gstat -h[header] output

Item	Description
Flags	—
Checksum	The header page checksum. In ancestor versions of InterBase, it was a unique value computed from all the data in the header page. It was turned off and, in Firebird, it is always 12345. When the header page is stored to disk and later read, the checksum of the retrieved page is compared to 12345 and, if it does not match, a Checksum error is raised. This catches some kinds of physical corruption.
Generation	Counter incremented each time the header page is written to.
Page size	The current database page size, in bytes.
ODS version	The version of the database’s on-disk structure. It will be 11.2 for v.2.5, 11.1 for v.2.0, 10.1 for v.1.5 or 10.0 for v.1.0.

Item	Description
Oldest transaction	The transaction ID of the oldest “interesting” transaction, often referred to as “OIT”. For info about these, refer to Chapter 25, Overview of Firebird Transactions .
Oldest active	The transaction ID of the oldest active transaction, often referred to as “OAT”.
Oldest snapshot	Often referred to as “OST”, the ID of the transaction that was the oldest one not eligible for garbage collection when the last sweep or GC process began. The engine subtracts the OST from the oldest (interesting), OIT, and compares the result with the sweep interval, to determine whether it is time to flag an auto-sweep. The default sweep interval is 20,000 but it can be configured to a smaller or larger “gap”.
Next transaction	The ID that Firebird will assign to the next transaction.
Bumped transaction	Obsolete in Firebird.
Sequence number	The sequence number of the header page, always zero.
Next connection ID	ID number of the next database connection.
Implementation ID	Intended to identify the architecture of the hardware on which the database was created.
Shadow count	The number of shadow sets defined for the database.
Number of cache buffers	The size of the database cache, in pages. Zero means the database uses the server's default (DefaultDBCACHEPages in firebird.conf, default_cache_pages in ibconfig/isc_config (v.1.0.x)).
Next header page	The page number of the next header page—although it appears not to be maintained and is not used in current Firebird versions.
Database dialect	The SQL dialect of the database. For more about database dialect and its implications, refer to the topic in Chapter 5, Migration Topics. It is not recommended to take a dialect 1 database beyond Firebird 1.5 without “doing the hard yards” to bring its capabilities up to date.
Creation date	The date when the database was created or last restored from a <i>gbak</i> backup.

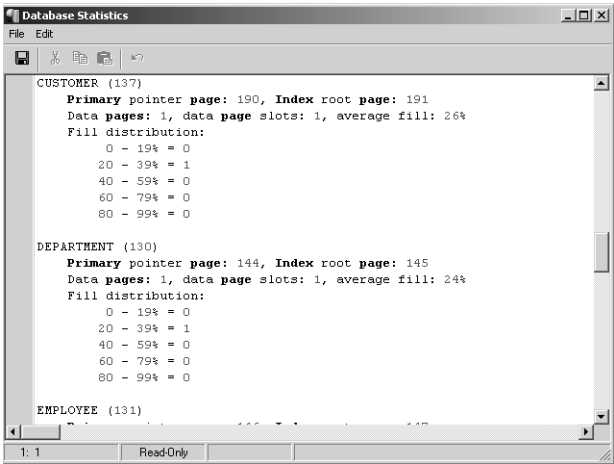
Item	Description
Attributes	<p>force write—indicates that forced database writes (synchronous writes) are enabled.</p> <p>no_reserve—indicates that space is not reserved on each page for old generations of data. This enables data to be packed more closely on each page and therefore makes the database occupy less disk space—ideal for read-only databases.</p> <p>read_only—indicates that the database is in read-only state.</p> <p>shutdown (full, single or multi)—indicates database is shut down, i.e. no ordinary users can connect. A shutdown state has been effected by use of the command line gfix -shut <type> or an equivalent Services API call.</p> <p>locked—nBackup has the database locked for an operation and updates are being temporarily saved to a delta file</p>
Variable header data	<p>Enumerates any variable values that may have been assigned to the database, viz.,</p> <ul style="list-style-type: none">—sweep interval—information about secondary files (if any)

The -data switch

```
gstat -data db_path_and_name
```

Retrieves a table-by-table database summary displaying information about data pages. To include the system tables (RDB\$XXX) in the report, add the -system switch.

Figure 38.4 Example of gstat data page summary output



The command-line output is similar:

```
COUNTRY (31)
Primary pointer page: 190, Index root page: 19
Data pages: 1, data page slots: 1, average fill: 26%
```

Fill distribution:

0 - 19% = 0
 20 - 39% = 1
 40 - 59% = 0
 60 - 79% = 0
 80 - 99% = 0

For each table in the database the following figures are displayed:

Table 38.6 gstat -d[ata] output

Item	Description
Primary pointer page	The page number of the first page of indirection pointers to pages storing data from the table.
Index root page	The page number that is the first pointer page for the table's indexes.
Data pages	The total number of data pages on which the table's data are stored. The count includes pages storing uncommitted record versions and garbage, since this program can not distinguish the difference.
Data page slots	The number of pointers to database pages kept on pointer pages. Should be the same as the number of data pages.
Average fill	This is a histogram with five 20% "bands", each showing the number of data pages whose average fill percentage falls within that range. Fill percentage is the proportion of the space on each page that actually contains data. In the sample here, the average fill is low because the employee.fdb database has small record structures and there are not many of them. The sum of these numbers gives the total number of pages containing data.
Fill distribution	A histogram summarizing the distribution of usage capacity across all of the pages allocated for the table. In this example, only one page happens to have been used so far and it is less than 40 percent filled.

Restricting the output from gstat -data

If you don't want a data report for every table, the -t switch allows you to specify a list of tables in which you are interested.

Syntax is

```
gstat -data db_path_and_name -t TABLENAME1 [TABLENAME2 [TABLENAME3 ...]]
```



In versions below Firebird 2.0, table names must be typed in upper case. In the old versions, gstat does not support the -t[table-list] switch on databases that use quoted, case-sensitive table identifiers.

The -r[ecords] switch

```
gstat -r db_path_and_name
```

Retrieves and displays record size and version statistics:

- For rows: average row length (bytes) and total rows in the table.

- For back versions: average version length (bytes), total versions in the table, and maximum version chain for a record.

The total number of rows reported in a table could include both active and dead transactions. Record and version length are applicable to the actual user data—they do not take into account the header that prefixes every record version.

Example The three tables CAULDRON, CAULDRON1 and CAULDRON2 have identical metadata and cardinality of 100,000 records. The nominal, uncompressed record length is ~900 bytes.

CAULDRON is a clean table of rows with no back versions. The average stored row length is ~121 bytes: about an 87% compression efficiency.

CAULDRON1 has an active transaction that just executed:

```
DELETE FROM CAULDRON1;
```

Every row has a zero (0.00) length because the primary record is a delete stub that contains only a row header. The committed records were all restored as back versions and compressed to the same size they had when they were the primary records. The table still has the same number of pages (4,000) as before the DELETE operation. The average fill factor rose from 85% to 95% to house all the delete stubs.

CAULDRON2 has an active transaction that just executed:

```
UPDATE CAULDRON2 SET F2OFFSET = 5.0
```

The updated records have each grown by two bytes (121 to 123), attributable to lower compression.

The value 5.0 replaced many missing and zero values, which made the field value different from its neighboring fields. There are now 100,000 back versions averaging 10 bytes each. The average fill factor has risen to 99% and the table grew 138 pages to 4,138.

```
> gstat proj.gdb -r -t CAULDRON CAULDRON1 CAULDRON2
...
Analyzing database pages ...
CAULDRON (147)
Primary pointer page: 259, Index root page: 260
Average record length: 120.97, total records: 100000
Average version length: 0.00, total versions: 0, max versions: 0
Data pages: 4000, data page slots: 4000, average fill: 85%
Fill distribution:
0 -19%=0
20 -39%=0
40 -59%=0
60 -79%=0
80 -99%=4000
CAULDRON1 (148)
Primary pointer page: 265, Index root page: 266
Average record length: 0.00, total records: 100000
Average version length: 120.97, total versions: 100000,
max versions: 1
Data pages: 4000, data page slots: 4000, average fill: 95%
Fill distribution:
0 -19%=0
```

```

20 -39%=0
40 -59%=0
60 -79%=0
80 -99%=4000
CAULDRON2 (149)
Primary pointer page: 271, Index root page: 272
Average record length: 122.97, total records: 100000
Average version length: 10.00, total versions: 100000, max
versions: 1
Data pages: 4138, data page slots: 4138, average fill: 99%
Fill distribution:
0 -19%=0
20 -39%=0
40 -59%=0
60 -79%=0
80 -99%=4138

```



For API programmers, the Firebird API function `isc_database_info()` and the Services API provide items which enable you to retrieve performance timings and database operation statistics into your application programs. For Delphi®, C++Builder® and Kylix® programmers, several suites of components are available with support for capturing database statistics.

Monitoring Locks

All versions

Locks are used in multi-user environments to synchronize work and prevent processes from destroying the integrity of one another's work. Firebird uses both operating system locking and a proprietary lock manager to coordinate database access. The Lock Print utility can be a revealing tool for tracking down deep problems in client code.

Locking in Firebird

Why, you might well ask, should we be interested in locks in a system that doesn't lock things? The concurrency and consistency that are visible to clients are based on transactions and controlled through record versions. However, the answer is that Firebird does use locks internally. It maintains the consistency of the on-disk structure through locks and careful write. Coincidentally, it also uses the operating system locking services to control access to database files prevent the SuperServer from opening the same file twice under different names.

Transaction-based locking allows locks to be acquired at any point during the transaction. However, once acquired, they can only be released at the end. Even the statement-level explicit locking introduced in Firebird 1.5 does not allow an "unlock" statement. Statements execute in a normal transaction and commit or rollback releases the lock, as usual.

While Firebird eschews the standard two-phase locking for its primary concurrency control for its lack of adequate levels of concurrency and consistency, it does use locks for the duration of changes, to keep two transactions from writing to the same page at the same time. The internal locking is controlled by the system itself. Objects in the system—known as *lock owners* or, simply *owners*—contend for locks on a number of resources. These, not surprisingly, comprise the pages containing rows that have been earmarked for update.

When a transaction gets a lock on a page, it holds the lock until the lock manager asks it to release the page, so a transaction can make many changes to a single page without releasing and re-reading it.

The Lock Manager Module

In Superserver, the Lock Manager can be thought of as a separate “control centre” that transactions negotiate with to acquire the right to proceed with requests. The Lock Manager comprises a chunk of memory and some request-handling routines. Its memory is tabulated into various blocks:

- lock blocks which refer to resources
- request blocks which represent requests for a lock on a resource
- owner blocks which represent transactions and other objects that request locks
- history blocks

Its routines are responsible for accepting and managing requests by owners for locks on resources, for allocating the locks and for releasing them. SuperServer also manages “latches” for coordinating changes by concurrent transactions.

Classic is simpler: owners queue to get control of the lock table so that each process's lock management code can queue, grant, and release locks for its owners.

Lock states

Every operating system offers some kind of free/busy mechanism for synchronizing resource events. Because Firebird needs a multi-state control mechanism, it implements its own seven-state lock management system. The following figure illustrates how the locking levels are decided:

Figure 38.5 Firebird internal lock states

	0 no lock	1 null lock	2 shared read	3 protected read	4 shared write	5 protected write	6 exclusive
0 no lock	Y	Y	Y	Y	Y	Y	Y
1 null lock	Y	Y	Y	Y	Y	Y	Y
2 shared read	Y	Y	Y	Y	Y	Y	n
3 protected read	Y	Y	Y	Y	n	n	n
4 shared write	Y	Y	Y	n	Y	n	n
5 protected write	Y	Y	Y	n	n	n	n
6 exclusive	Y	Y	n	n	n	n	n

- 0 is free.
- 1 a null lock, indicates an interest in the object with no restrictions on others' use of it. Acquiring a null lock allows a transaction to read the lock data.
- 2 a shared read, which allows writers. Shared read is the normal mode for table locking until the transaction changes some part of the table.
- 3 a protected read, allows other readers, but not writers. Protected read is the normal mode for locking a database page that is in cache and has not been modified.
- 4 is shared write, the other normal table locking mode. Shared write is compatible with shared read and other shared writers, but not with either protected mode.
- 5 is protected write which allows shared readers and null locks and nothing else. Protected write is used for consistency mode (table stability) and for locking a database so normal users can not access it.
- 6 is exclusive and is used for internal structures when concurrent access could interfere with an update or cause the second transaction to read incomplete data.

The lock table

The Lock Manager maintains a lock table to coordinate resource sharing among the client threads. The information stored here can be useful when trying to correct deadlock situations, for example:

- all the locks currently in the system, with their states
- global header statistics such as the size of the lock table, the number of free locks, the number of deadlocks, and so on.
- process flags, indicating such things as whether the lock has been granted or is waiting

Locks are stored in series, each series identified by a number, according to the type of resource locked. The series numbers are explained in Table 38.11, “Resource types (Series)”.

Use of null locks

Transactions starting up use the lock table as a bulletin board. To avoid garbage collecting record versions that another transaction needs, each transaction needs to know the identity of the oldest action any other transaction has seen. Here is how it is done:

When it starts, a transaction stores the ID of the oldest transaction still running into the data portion of its transaction lock.

It then gets null locks on all concurrent transactions. When each lock is acquired, the contents of its data portion are returned.

The new transaction checks the lock of each existing transaction to discover the identifier of the oldest transaction that any active transaction knows about.

“Free ...” lists

Lists of owners, resources and requests are chained forward and backward. An offset in each block contains the pointer to the next block. The lists of offsets are used when the Lock Manager needs to allocate a new block and tries to find blocks that are free to be reused. It will allocate a new block only if there are no free blocks of the right type and size.

The “free ...” items indicate the forward and backward pointers to the offsets of the first and last free request blocks, respectively.

Deadlocks

A deadlock occurs when Owner A wants a lock on Resource 1 which is held in an incompatible mode by Owner B and Owner B wants a lock on Resource 2 which is held in an incompatible mode by Process A. A deadlock can also occur with a single resource if both owners start with read locks and request conversions to write locks.

It is a stand-off that the owners can not resolve without intervention. Resolution comes once the deadlock is detected by the next scan and the lock manager returns a fatal error to one owner or the other. The default deadlock scan interval—DeadlockTimeout in the configuration file—is 10 seconds. They are not done under every condition where there is a WAIT. Waiting is normal in a system that manages parallel updates and unnecessary scanning is costly.

“Deadlocks” that are not deadlocks

The deadlocks reported may not help to pinpoint "deadlock problems" observed in your applications. Deadlocks always involve two owners (or two separate transactions) each being stymied by the other. Firebird tends to deliver "deadlock" messages to clients for most types of locking conflict, even though a true deadlock—such as would be reported in a lock print—is relatively rare.

- Errors that are returned as "lock conflict" from NO WAIT lock requests are not recorded in the lock table as deadlocks because only one owner is waiting.
- Errors returned as "deadlock" with the secondary message Update conflicts with concurrent update are not actual deadlocks either. What has happened in those cases is that one owner has modified (or deleted) a row and moved on. Another concurrent owner has attempted to modify (or delete) the same record, has waited for the first owner to release it and now fails because the latest committed version has changed.

The lock table data can be reported in a more or less human-readable format, by way of the *Lock Print* utility.

The Lock Print Utility

The program that extracts the lock table statistics is an executable named *fb_lock_print*, which you will find in the /bin directory of your Firebird installation. (For v.1.0.x, look for *iblockpr.exe* on Windows, *gds_lock_pr* on POSIX.). Two syntaxes are available: one for static reports, the other for capturing a sampling interactively at specified intervals.

From V.2.5 onward, each database has its own lock file and either the database path or the path to the lock file must be specified, although not both. For the details, see the notes for the *-d* and *-f* switches in Table 38.7, “Switches for Lock Print reports”, below.

```
Syntax for Firebird 2.5 and higher:
fb_lock_print -d path/to/db_file <other switches>
or
fb_lock_print -d path/to/lock_file <other switches>
```

```
Versions 2.1.x, 2.0.x and 1.5.x:
fb_lock_print <switches>
```

v.1.0.x, POSIX:

`gds_lock_pr <switches>`

v.1.0.x, Windows:

`iblockpr <switches>`

The *fb_lock_print* program accepts a number of switches, described below in Table 38.7.

When no switches are provided, *fb_lock_print* prints summary information describing the lock header and the owners communicating with the lock manager.

Table 38.7 Switches for Lock Print reports

Switch	Description
<code>-d filepath</code>	V.2.5 and higher only (mandatory if <code>-f</code> switch is not supplied, invalid if <code>-f</code> switch is supplied): file path to the database file. ¹
<code>-f filepath</code>	V.2.5 and higher only (mandatory if <code>-d</code> switch is not supplied, invalid if <code>-d</code> switch is supplied). Specifies that analysis should be done on the named file rather than the live lock file. Since V.2.5, lock files are database specific and are located not in Firebird's root directory, but as follows: POSIX: <code>\tmp\firebird</code> Windows: <code>%commonappdata%\firebird</code> ²
(no other switches)	Prints summary information describing the lock header and the owners communicating with the lock manager.
<code>-a</code>	Prints the contents of the lock table including the lock header, lock blocks, owner blocks, and request blocks. A lock block represents a resource that can be locked (database, transaction, relation, database page, etc.) and identifies owners that have or have requested a lock on the object. A request block describes a request by a process for a lock on a resource. A request block may represent either a granted lock or a pending request for a lock.
<code>-c</code>	Indicates that the lock table should be copied rather than used live. The copy is quick and produces a static snapshot of the lock table. It will, however, stop all database access on the computer while it runs.
<code>-h</code>	Prints only the history.
<code>-i <switches> table-names</code>	Begins interactive mode (see below). If <code>-i</code> is used alone, it prints everything.
<code>-l</code>	Prints only lock blocks.
<code>-n</code>	Indicates that there is “no bridge”. A bridge is a transitional mechanism recognizing multiple servers with different major versions of Firebird on the same machine. It is not applicable to Firebird v.1.0.x or v.1.5 but is likely to be implemented when future major versions of Firebird are released.
<code>-o</code>	Prints owner blocks.

Switch	Description
-p	Same as -o. (Owner blocks are sometimes called process blocks)
-r	Prints request blocks
-s <n>	Prints out the lock table header block, the owner blocks, and the locks in a particular series. The argument <n> is a number identifying the lock resource type that you want to report on. Refer to Table 38.11, “Resource types (Series)” for the numbers.
-t	Prints statistics for all series (interactive reporting only)
-w	Prints the <i>waiting on</i> graph—owner blocks with waiting requests, what they are waiting for, what those owners are waiting for and so on. With this report you can work out which owner’s request is blocking others’ in the lock table. This is the easiest way to find a blockage, though a full lock print will tell you more about the interrelationships of the enqueued requests

- 1. Versions 2.5 and higher maintain separate lock files for each database.
- 2. Location of %commonappdata% is Windows version dependent and by default should be %allusersprofile%\Application Data. Use the *set* command to find %allusersprofile%

Static reports

Static reports print a snapshot of the lock table. Any switches are valid except -i and you can “stack” multiple switches into one. For example, to print the “waiting on” graph plus the history block:

```
fb_lock_print -wh
```

Interactive reports

The second form collects a specified number of samples at fixed intervals and produces an interactive report monitoring current lock table activity. The syntax is:

```
fb_lock_print [-i{a,o,w}] [t n]
```

- t specifies the time in seconds between samplings
- n specifies the number of samples to be taken.

If you do not supply values for n and t, the default is n = 1.

Sampling occurs *n* times at intervals of *t* seconds. One line is printed for each sample. The average of the sample values is printed at the end of each column.

The following statement prints “acquire” statistics (access to lock table: acquire/s, acqwait/s, %acqwait, acqrtry/s, and rtrysuc/s) every 3 seconds until 10 samples have been taken:

```
fb_lock_print -ia 3 10
```

At the end of the chapter is a snapshot of an interactive report with explanations of the meanings of the columns.



Buffer limits in the command shell may cause all but the last part of the output to “vanish”. You can pipe the output to the more utility (or less on POSIX, if you prefer it).

```
fb_lock_print -wh | more
```

Once the shell display is full, press or hold down the Return (Enter) key to display more output, a row at a time. Ctrl-C terminates either more and less.

Reporting to a file

The results are usually quite long for viewing on the console. You can supply a pipe to an output file, for example in a directory named /data/server_reports/ (your choice!), like this: `fb_lock_print -a > /data/server_reports/lock.txt`. If you find a lock print running for more than a minute or two, or find that it is filling your disk, kill it with Ctrl-C or your platform equivalent.

The blocks are listed in the order of the internal lists. New blocks are put at the head of the list, so a newly minted lock table will be shown with blocks in reverse numeric order. As lock, request, and owner blocks are released and reused, the order becomes thoroughly scrambled. A text editor is very useful for chasing through the relationships.

A simple lock print

We'll take a look at an example from a very simple static lock print with no switches.

- The lock header is always first.
- Next come the owner blocks—owner block followed by all the requests for that owner. Each owner in the chain is printed with its requests.
- After all owners and requests come the locks.
- The last items are the history records.

The Lock_Header block

First, we'll consider only the lock header block that describes the general shape and condition of the lock table. In Figure 38.6, numbers have been added for reference to each item of explanation in Table 38.8, “Lock Header block entries”, below.

Our lock print represents a database that has just been created and is being accessed by a single copy of *isql* on a Windows Superserver version.

Figure 38.6 The Lock Header block

```

1) LOCK_HEADER BLOCK
  (2) Version: 114, (3) Active owner: 0, (4) Length: 32768, (5) Used: 12976
  (6) Semmask: 0x0, (7) Flags: 0x0001
  (8) Enqs: 10, (9) Converts: 0, (10) Rejects: 0, (11) Blocks: 0
  (12) Deadlock scans: 0, (13) Deadlocks: 0, (14) Scan interval: 10
  (15) Acquires: 36, (16) Acquire blocks: 0, (17) Spin count: 0
  (18) Mutex wait: 0.0%
  (19) Hash slots: 101, (20) Hash lengths (min/avg/max): 0/ 0/ 1
  (21) Remove node: 0, (22) Insert queue: 0, (23) Insert prior: 0
  (24) Owners (5): forward: 12056, backward: 11628
  (25) Free owners (4): forward: 11804, backward: 12232
  (26) Free locks (1): forward: 11560, backward: 11560
  (27) Free requests (1): forward: 12116, backward: 12116
  (28) Lock Ordering: Enabled

```

Table 38.8 Lock Header block entries

Tag#	Item	Explanation
1	LOCK_HEADER_BLOCK	First block on any lock print report. Each report outputs exactly one lock header block.
2	Version	The lock manager version number. For Firebird 1.5, version is 115 for SuperServer and 5 for Classic. For Firebird 1.0.x (like our sample) the versions are 114 and 4, respectively.
3	Active owner	The offset of the owner block representing the owner which currently has control of the lock table, if any. In this case, no process is writing to the lock table, so the Active Owner is 0.
4	Length	Total space allocated to the lock table in bytes.
5	Used	The highest offset in the lock table which is currently in use. There may be free blocks in the table between the beginning and the used point if owners have come and gone. Before new blocks are allocated between this point and the end of the lock table, any free blocks will be reused.
6	Semmask	On systems that use static semaphores (e.g. Posix), this is the pointer to an SMB block containing the number of semaphores in use. When a semaphore is needed and none are available, the lock manager will loop through the owner blocks, looking for one that has a semaphore that it's not using. Failing that, the system returns the error “semaphores are exhausted” – meaning that all the semaphores compiled into the operating system are in use.
7	Flags	Two flag bits are defined: LHB_shut_manager which, if set, indicates that the database is shutting down and the lock manager ought not to grant more requests; and LHB_lock_ordering. The Firebird default for LHB_lock_ordering is that locks are granted in the order requested (FIFO order). The alternative setting relates to an obsolete locking strategy and is not used.
8	Enqs	Enqueue requests—requests that have been received for locks. This number comprises requests that cannot yet be satisfied and requests that can be satisfied immediately, but not requests that have come and gone.

Tag#	Item	Explanation
9	Converts	Requests to increase the level of a lock. A process which holds a lock on a resource will request a mode change if its access to the resource changes. Conversions move from a lower level lock (e.g. shared read) to a more restrictive level (e.g. exclusive). For example, a transaction in concurrency mode, which has been reading a table and decides to change data in the table will convert its lock from shared read to shared write. Conversions are very common on page locks because a page is usually read before being altered.
10	Rejects	Requests that cannot be satisfied. These may be locks requested in "no wait" mode, or they may be requests which were rejected because they caused deadlocks. Since the access method occasionally requests "no wait" locks for internal structures, you will sometimes see rejects even when all transactions run in "wait" mode and there is no conflict between their operations.
11	Blocks	Requests which could not be satisfied immediately because some other owner has an incompatible lock on the resource.
12	Deadlock scans	The number of times that the lock manager walked a chain of locks and owners looking for deadlocks. The lock manager initiates a deadlock scan when a process has been waiting 10 seconds for a lock.
13	Deadlocks	The number of actual deadlocks found. See the topic Deadlocks, below.
14	Scan interval	The number of seconds the Lock Manager waits, after a request starts waiting, before starting a deadlock scan. Ten seconds is the default.
15	Acquires	The number of times an owner—or the server on behalf of a specific owner—acquired exclusive control of the lock table to make changes.
16	Acquire blocks	The number of times an owner—or the server on behalf of a specific owner—had to wait to acquire exclusive control of the lock table.
17	Spin count	There is an option to wait on a spin lock and retry acquiring the Firebird lock table. By default, it is set to zero (disabled), but can be enabled with the configuration file.

Tag#	Item	Explanation
18	Mutex wait	The percentage of attempts that were blocked when owner tried to acquire the lock table; i.e. $((\text{acquire blocks}) / (\text{acquires})) * 100$
19	Hash slots	Resources are located through a hash table. They are stored according to value (q.v.). By default, the hash table is 1009 slots wide (101 in Firebird 1.5). That value (which should be a prime number) can be increased using the configuration file. It should never be reduced to less than the default.
20	Hash lengths	Below each hash slot hang the resources (lock blocks) that hash to that slot. This item reports the minimum, average, and maximum length of the chain of lock blocks hanging from the hash slots. An average hash length > 15 indicates that there are not enough slots.
21	Remove node	To avoid the awkward problems caused when the active owner dies with the lock table acquired and potentially half-updated, the owner records the intention to remove a node from the table. When the operation succeeds, the owner removes the remove notation. If any owner finds a remove notation that it did not create, it cleans up.
22	Insert queue	The equivalent of the remove node entry above, except that this is the node being inserted.
23	Insert prior	To clean up a failed insert, it is necessary to know not just what was being inserted, but also where it was being put. This is where.
24	Owners	The number of owners that have connections to the lock table. Only one of those owners can update the table at any one time (the "active owner"). Other owners hold and wait for locks. In our example there are four owners, none active. Two owners are attachments from ISQL, one may be an attachment from DSQL and one is the database itself.
25	Free owners	The number of owner blocks that have been allocated for owners that have terminated their connections leaving the blocks unused. In this case, there are two, probably transactions involved in creating the database that have since committed.

Tag#	Item	Explanation
26	Free locks	Lock blocks identify a resource (database, relation, transaction, etc.) that has been locked, not a lock on the resource. This item is the number of lock blocks that have been released and not yet reused. In this case there is one free lock. When an owner requests a lock on a resource that is not currently locked, the lock manager looks first to the free lock list in the lock header. If there is a block there with the right key length, that lock block is reused. If not, a new lock block will be allocated out of free space.
27	Free requests	Request blocks identify a request for a lock on a resource, whether satisfied or not. This item is the number of request blocks that have been released and not reused.
28	Lock ordering	Lock ordering means taking lock requests in the order received, even if that blocks subsequent requests that could be served immediately. It is enabled when the LHB_lock_ordering flag (see item 7) is set. It is the default in Firebird, since it provides optimal throughput overall. The alternative (deprecated now) is to grant locks to all owners willing to share and "starve" non-sharing owners that have existing requests. This deprecated strategy would ensure that the sharers would be handled quickly, but at the risk of detrimmenting the others.

Owner blocks

An owner block, depicted in Figure 38.7, describes a transaction or other user of the Lock Manager. Owners fall into one of several types, identified by one of five numbers:

- 1 Process
- 2 Database
- 3 Client attachment. In Classic, client attachments are always processes.
- 4 Transaction
- 255 Dummy process



By some bizarre coding convention, transactions are never referred to by the number 4, but by 255 instead.

Figure 38.7 An Owner block

```
(1) OWNER BLOCK 11872
      (2)Owner id: 9909104,(3) type: 2,(4) flags: 0x202,(5)pending: 0,
      (6)semid: 3
      (7)Process id: 1868, (8)UID: 0x0 (9)Alive
      (10)Flags: 0x02 scan
      (11)Requests (3): 12300, backward: 12124
      (12)Blocks: *empty*
```

The offset of the particular owner block in the lock table (here it is 11872) is also the ID used in the lock header for the “active owner”, if that owner is actively modifying the lock table. The first block printed will usually be the number printed in the Lock Header Block as the beginning of the Owner list.. The value of the list pointers is actually a field in the block which contains the block's own forward and backward pointers.



If you inspect your lock print in an editor, you can search for requests belonging to this owner ID.

Table 38.9 Owner block entries

Tag#	Item	Explanation
1	OWNER BLOCK	Identifies the particular owner. The number following the header (11872) is the offset of the owner block in the lock table and is used as this owner's ID throughout the table.
2	Owner ID	In Classic, the owner is always a process and the Owner ID is always a process ID. In Superserver, the owner is either the database and the ID is that of the database block, or the owner is the attachment and the ID is that of the attachment block
3	Owner type	The owner type is a number between 1 and 4, or 255 (dummy process).
4	Flags	Bits that indicate a specific state. A process can be in more than one state at once—refer to Table 38.10, “Owner flag states”, below.
5	Pending	The offset of a lock request block describing a lock that this owner has requested but has not yet been granted. An owner can have at most one pending lock request at a time.
6	Semid	The ID of the semaphore assigned to this owner. If it could be used, the word “Available” would follow the id.

Tag#	Item	Explanation
7	Process	On Classic, the process ID of the owning process. On Superserver, if the owner is an attachment, database, or transaction, it will be the process ID is that of the Superserver process.
8	UID	On POSIX, the user ID of the owner process. On Windows, it is always zero.
9	Alive Dead	The lock printer invokes the routine <code>ISC_check_process_existence</code> and reports the results.
10	Flags	The flag mnemonics—correctly printing 2 as 0x02 (not 0x202, as in (4)).
11	Requests	Lock requests, either acquired or pending, that are associated with this process. The forward and backward numbers are the beginnings of the forward and backward self-relative queues of requests that belong to the process. The numbers are offsets.
12	Blocks	Transitory list of locks (request blocks) owned by this process that are blocking other lock requests. It is cleared when the process has been notified that it should release or downgrade its lock, assuming it is able to do so.

Table 38.10 Owner flag states

Symbol	Value	State
OWN_blocking	1	Owner is blocking. If set, the process in question has at least one lock that another owner wants non-shared.
OWN_scanned	2	Owner has been checked in the current deadlock scan.
OWN_manager	4	Systems that disallow signals to cross groups have a privileged lock manager to transmit signals. This owner is that manager.
OWN_signal	8	Owner needs to signal and has failed to do so directly, so it is being called on by the manager.
OWN_wakeup	32	Owner has been poked to release a lock.
OWN_starved	128	This thread may be starved. Starvation happens in Solaris multi-threading and indicates that the Owner (process) has made more than 500 unsuccessful attempts to get the lock table to release a lock.
OWN_signaled	16	Signal is thought to be delivered. It applies to <code>OWN_ast_flags</code> , but is OR'ed into the reported flags. NB The value printed here, hex 202, seems to be a print-parsing error.

Lock blocks (resource blocks)

Lock blocks follow request blocks in the printout, but it makes request blocks easy if you understand lock blocks first. A lock block represents *a resource that has been locked*.

Lock types—“Series”

Resource locks come in different types, or *series*, according to the type of resource that owners request to lock. Table 38.11 identifies and describes the various resource lock types and summarizes their purposes.

Table 38.11 Resource types (Series)

Symbol	Series	Type
LCK_database	1	Root of lock tree. In Classic, a database lock is taken by each process that attaches a database. The first process takes an exclusive lock. The next process notices a conflict and signals the first to downgrade his lock from exclusive to shared. Thereafter, all locks on the database itself are for shared read. In Superserver, the database takes out an exclusive lock on itself.
LCK_relation	2	Individual table lock. A table lock indicates that the process has read or written to the specified table in its current transaction, or that it has used the RESERVING clause on the START TRANSACTION statement to declare its intention to read or write to the table. In this case, both owners are reading the table. The key field is the RDB\$RELATION_ID value for the table. Note that both requests report their state as 2(2), indicating that they requested and received a shared read lock on the table.
LCK_bdb	3	Individual buffer block. A BDB lock is a lock on a database page. These locks are held when two or more owners have attached a database on Classic. They are taken when the process wants to read or write a page and released when the process runs out of buffers in cache and needs to free up space or when another owner needs the page.
LCK_tra	4	Individual transaction lock. Each action takes an exclusive lock on its own transaction ID when it starts. Other owners can acquire null locks on it to read its state.
LCK_rel_exist	5	Relation existence lock. Prevents tables from being dropped while any owner has a prepared request that uses that table.
LCK_idx_exist	6	Index existence lock. Prevents indexes from being dropped or inactivated while any owner has a prepared request that uses the resource.

Symbol	Series	Type
LCK_attachment	7	Not used. Attachment lock to support dBase record locks which can exist across transaction boundaries.
LCK_shadow	8	Lock to synchronize addition of shadows, mainly on Classic.
LCK_sweep	9	Sweep lock for single sweeper. Sweep is a moderately expensive operation and works best if only one thread or attachment does it. The actual sweeper keeps an exclusive lock in this series to avoid conflicts. This series is used for interprocess communication in Classic.
LCK_file_extend	10	Lock to synchronize file extension. Extending the database file is another operation that doesn't go as well if two transactions try to do it at once. This series is used for interprocess communication in Classic.
LCK_retaining	11	Youngest commit retaining transaction. This is used only on VMS. It probably marks a place where Firebird has extended the locking semantics to fit its needs and so requires a special hack to work with the VMS lock manager.
LCK_expression	12	Expression index caching mechanism. Originally this series was intended to describe expression indexes—how to evaluate them, what the result of the evaluation was likely to look like, etc. For some reason, they're now used when deleting any index.
LCK_record_locking	13	Lock on existence of record locking. This series indicates that record locking has been requested for a particular table. The first process to request record locking for a table also gets a protected lock on the table. Until that lock is challenged, record locks are kept in the attachment. When a second transaction arrives, the table lock is downgraded to shared and locks are expressed. This series is used only in the deprecated PC emulation code.
LCK_record	14	Record Lock. This series is also used only in the deprecated PC emulation code and uses the record's RDB\$DB_KEY as the lock name.
LCK_prc_exist	15	Procedure existence lock. Prevents procedures and triggers from being dropped while any owner has a prepared request that uses (or depends on) that resource.

Symbol	Series	Type
LCK_range_relation	16	Relation refresh range lock. Again, this series is used only in the PC emulation code which has a concept of update ranges.
LCK_update_shadow	17	Shadow update sync lock, used to limit to one the number of processes that cause processing to roll over to a shadow or disable shadowing.

Syntax variation for lock block reports

To print the resource lock blocks for a specific series, you need to include the series number as an argument:

```
fb_lock_print -s 2
```

Series 1—Database

The lock block in Figure 38.8 represents the database itself. It is being held exclusively by one owner—itsself. In Classic, you would see several owners for the database.

Figure 38.8 A lock block—Series 1 (database)

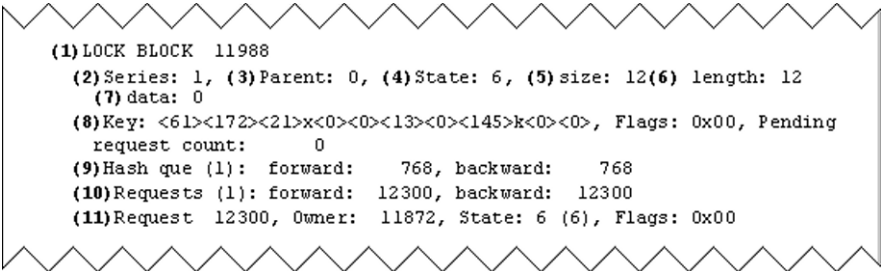


Table 38.12 Lock (resource) block entries

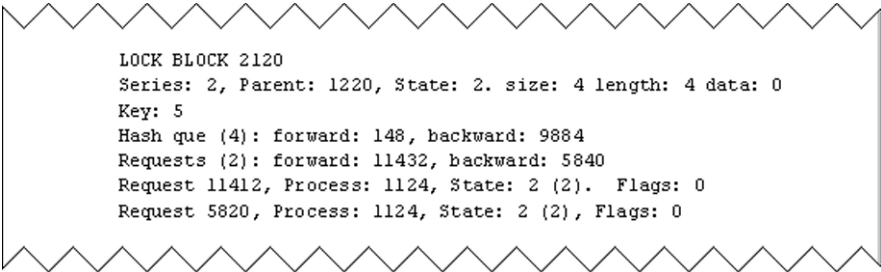
Tag	Value	Explanation
1	LOCK BLOCK	Identifies the block as the description of a resource that has been locked. The number is the offset of that block in the lock table. It identifies the block in the other blocks that refer to it.
2	Series	The type of resource this lock represents. This one is a Series 1 type—a database resource.
3	Parent	The parent of all locks associated with a database is the database lock itself. The only resource locks that should 0 for the parent are the database locks and journal. The keys that identify locks within a series are meaningful only in the context of a database. NB Discrepancies (a.k.a “bugs”) will be noticed by the careful reader.

Tag	Value	Explanation
4	State	The highest current state of the lock. Locks have seven possible states—refer to Figure 38.1 <i>Firebird internal lock states</i> , right at the beginning of this topic. A null lock allows a process to acquire a lock on a resource regardless of whether (and how) someone else has locked it. Acquiring that lock allows the locker to read the data from the lock itself. Firebird keeps some important but volatile information in locks—see the topic below, Use of null locks.
5	Size	The length, in bytes, of the portion of the lock block that holds the key. The size is rounded to the natural boundary (word, longword, quadword) for the machine.
6	Length	The actual length of the key, which, because of rounding, may be less than the size.
7	Data	Only journal locks and transaction locks carry data. The data portion is a 32-bit integer.
8	Key	<p>The identifier of the resource being locked. The combination of the key, the series, and the parent uniquely identify the resource being locked:</p> <p>—For the database the key is the name of the database (or something equivalent). May not print on systems that use an integer identifier.</p> <p>—For relation and relation existence locks, it is the relation id.</p> <p>—For index existence locks, it is the relation id * 1000 plus the index id.</p> <p>—For a shadow lock, the key is null because there is only one state of shadowing for a database.</p> <p>—For a transaction, it is the transaction id.</p> <p>—For an attachment, it is the attachment id..</p>

Tag	Value	Explanation
9	Hash queue	The beginning and end of the hash queue for the resource key. The lock manager keeps a hash table to facilitate lookup of resources by name. When a process requests a lock on a resource, it identifies the resource by series, parent, and key. The lock manager mashes the values together to create a hash key, then searches the list associated with that hash key value for the desired lock block.
10	Requests	<p>First the number of lock requests for this resource, then forward and backward pointers to the request blocks. Note that the backward pointer points to the end of the last block.</p> <p>—Request—The list of requests including the identifier of the request block, the process that made the request, the actual state of the lock with the requested state in parenthesis. The state—6(6) in this case—indicates an actual state of 6 and a requested state 6.</p> <p>—Flags—The request flag contains bits which can be combined. They are:</p> <ol style="list-style-type: none">1. Blocking— A request is marked as blocking if someone else wants the resource and can not share it because of its current lock level. The blocking bit is cleared when a blocking notice has been sent to the blocking owner.2. Pending— This bit is the one most often seen. It indicates that the request is waiting for a blocking process to release its lock. You should not see the pending bit set for bdb locks.4. Converting— A request is converting if the process already has a lock on the resource and wants a higher level lock and the conversion can not be done immediately.8. Rejected— A lock request is rejected if the request is in “no wait” mode and cannot be satisfied immediately or if granting the request would cause a deadlock.

Series 2—Relation

Figure 38.9 A Series 2 (Relation resource) block



In this case, both owners are reading the relation. The key field is the RDB\$RELATION_ID value for the relation. Note that both requests report their state as 2(2), indicating that they requested and received a shared read lock on the table.

Series 3—BDB Descriptor (database page)**Figure 38.10** A Series 3 (BDB Descriptor) block

```

LOCK BLOCK 5384
Series: 3, Parent: 1220.  State: 3. size: 4 length: 4 data: 0
Key: 14
Hash que (2): forward: 11468, backward: 220
Requests (2): forward: 7644. backward: 1712
Request 7624, Process: 2556, State: 3 (3), Flags: 0
Request 1692, Process: 1124, State: 3 (3), Flags: 0

```

A BDB lock is a lock on a database page. These locks are held when two or more owners have attached a database on Classic. They are taken when the process wants to read or write a page and released when the process runs out of buffers in cache and needs to free up space or when another owner needs the page. In this example, both owners are reading page 14 (key value).

On Classic there are a lot of Series 3 type locks—one for each page buffer in the cache of each independent attachment. On SuperServer, most page locks are held within the server and not expressed in the lock table.

Series 4—Transaction**Figure 38.11** A Series 4 (Transaction resource) block

```

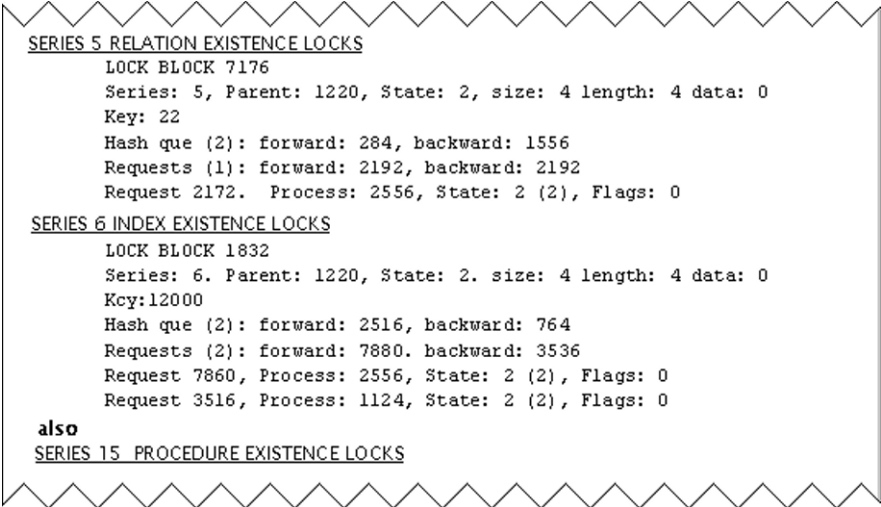
LOCK BLOCK 10492
Series: 4, Parent: 1220, State: 6, size: 4 length: 4 data: 585
Key: 585
Hash que (1): forward: 748, backward: 748
Requests (2): forward: 11528, backward: 6416
Request 11508, Process: 1124, State: 6 (6), Flags: 0
Request 6396, Process: 2556. State: 0 (3), Flags: 2

```

Each action takes an exclusive lock on its own transaction ID when it starts. This block describes the state of the locks on transaction 595. One transaction is waiting for the other to finish so it can decide whether the update it wants to do is legitimate. When the owner that holds the lock departs, its locks will be released and the waiting transaction can read the transaction inventory page to determine the fate of the vanished transaction.

Series 5, 6 & 16—Existence

Figure 38.12 Series 5, 6 and 15 Existence lock blocks



The relation existence locks (Series 5) prevent tables from being deleted while any process has a prepared request that uses that table. This lock is the source of Object in use errors that often occur when attempting to drop tables.

When a statement is prepared into a database “request”, the compiling process takes out a shared read lock on the existence of the relations and indexes included in the statement. Those locks are held until the request is released or the database is detached.

When a process wants to drop a relation from the database, rather than just delete its content, it must get an exclusive lock on the existence of the relation. Because no one can get an exclusive lock on a resource that is locked for shared read by another process, the shared read locks prevent the relation from being destroyed, and so prevent online metadata operations from breaking prepared requests.

This particular relation existence lock is on a relation which has an RDB\$RELATION_ID of 22.

The index existence locks prevent indexes from being dropped or deactivated while any process has saved a request that uses the index.

When a statement is prepared into a database “request”, the compiling process also takes out a shared read lock on the existence of the indexes included in the statement. Those locks are held until the request is released or the database is detached.

When a process wants to delete or deactivate an index, it must get an exclusive lock on the existence of the index.

Because no one can get an exclusive lock on a resource that is locked for shared read by another process, the shared read locks prevent the relation or index from being destroyed, and so prevent online metadata operations from corrupting compiled requests.

The index existence lock id is 12000 which is the relation id times 1000 plus the index id. This lock records an interest in the existence of index 0 for relation 12.

The procedure existence locks are exactly like relation and index existence locks, and serve a similar purpose. The key is the procedure ID from the system table RDB\$PROCEDURES.

Series 8—Shadow

Figure 38.13 A Series 8 (Shadow resource) block

```

LOCK BLOCK 1352
Series: 8, Parent: 1220, State: 2, size: 4 length: 4 data: 0
Key: 0
Hash que (3): forward: 1460. backward: 108
Requests (2): forward: 6844, backward: 7152
Request 6824, Process: 2556, State: 2 (2), Flags: 0
Request 7132, Process: 1124. State: 2 (2), Flags: 0

```

Every process that attaches to a database takes out a shared read lock on the state of shadowing for the database. If a process wants to add a new shadow file, it converts its lock to exclusive which notifies all other processes that a shadow file is about to appear, and they should write changes to that file. This series is used for interprocess communication in Classic. It is also used in Superserver, although not to any great effect, since IPC is not required.

Request blocks

Figure 38.14 Some request blocks

```

(1) REQUEST BLOCK 7624
(2) Process: 2556, (3) Lock: 5384, (4) State: 3, (5) Mode: 3. (6) Flags: 0
(7) AST: 3afc4c30, (8) argument: 3af776e0

REQUEST BLOCK 11508
Process: 1124, Lock: 10492, State: 6, mode: 6, Flags: 0
AS'I': 0, argument: 3af76a2c

REQUEST BLOCK 6396
Process: 2556, Lock: 10492, State: 0. Mode: 3, Flags: 2 AST: 0, argument: 0

```

Table 38.13 Request block entries

Tag#	Item	Explanation
1	REQUEST BLOCK	The identifier of this particular request
2	Process	The offset of the process block that describes the process that made this request.
3	Lock	The offset of the lock block that describes the resource being locked.

Tag#	Item	Explanation
4	State	The state of the lock which has been granted on the resource.
5	Mode	The state in which the lock was requested. In the first two examples, the state is the same as the mode. These are locks which have been granted. The first was granted in protected read mode, the second in exclusive. In the third example, the lock is pending, so the state is 0 (no lock) but the mode is 3 (protected read).
6	Flags	The request flag contains bits, which can be combined. They are: 1 - blocking 2 - pending 4 - converting 8 - rejected
7	AST	<p>The address of a routine to call if someone wants a conflicting lock on the resource held by this request. Routines to downgrade or release locks are always supplied for locks on the database, the state of shadowing and the buffer descriptor blocks which identify a database page in cache.</p> <p>—The database lock will be downgraded from exclusive (for the first user) to shared read when the second user shows up in the classic architecture. In SuperServer, the database holds an exclusive lock on itself.</p> <p>—The shadow shared read lock is released when another process requests the lock in exclusive mode so it can create new shadow file(s). As soon as the files are created, everyone re-seizes shared read locks on the state of shadowing.</p> <p>—When there is a conflict for a database page, the process that holds the page immediately releases it and downgrades its lock unless the page is actually in the process of being modified. If so, the page is marked as needing to be released as soon as the modification is done.</p>
8	Argument	<p>The address of something that the AST routine will want.</p> <p>—In the case of a BDB, it is the address of the structure in the process that describes the buffer.</p> <p>—In the case of the database and shadow locks, it is the address of the master block (IDB) that describes the database.</p>

The History block

The Lock Manager keeps track of the I/O actions it took on behalf of any owner. The most recent actions are output as the last two items in the printout, History and Events. Figure 38.15 is a snapshot of a sequence of History records:

Figure 38.15 Snapshot of History printout

```

GRANT:  owner = 11628, lock = 11744, request = 12516
ENQ:    owner = 12056, lock = 0, request = 11512
DENY:   owner = 12056, lock = 11744, request = 11512
ENQ:    owner = 12056, lock = 0, request = 12408
POST:   owner = 11628, lock = 11744, request = 12516
WAIT:   owner = 12056, lock = 11744, request = 12408
SCAN:   owner = 12056, lock = 11744, request = 12408
POST:   owner = 11628, lock = 11744, request = 12516
POST:   owner = 11628, lock = 11744, request = 12516
POST:   owner = 11628, lock = 11744, request = 12516
DEQ:    owner = 11628, lock = 11744, request = 12516
GRANT:  owner = 12056, lock = 11744, request = 12408

```

Owner 11628 is GRANTED a lock on resource 11744. Owner 12056 ENQueues a request for the same resource, asking to get it with no wait. The lock held by owner 11628 is in an incompatible mode, so that request is denied (DENY). Owner 12056 comes back with a different ENQueue request, asking for the lock again, but willing to wait. The Lock Manager POSTs pokes a notification to owner 11628 on the subject of resource 11744. Owner 12056 is told to WAIT. After 10 seconds, owner 12056 is still waiting, so the lock manager starts a deadlock SCAN. That yields nothing, so the lock manager goes back to poking owner 11628 (POST, POST, POST). Eventually the pokes get through, 11628 gives up (DEQueues) the lock and it is GRANTED to 12056.

The Events printouts deliver the same history information in a different format. Figure 38.16 is a snapshot of a sequence of history records output in the Events part of the printout:

Figure 38.16 Snapshot of Events printout

```

DEL_OWNER:  owner = 12232, lock = 12232, request = 0
DEL_OWNER:  owner = 12056, lock = 12056, request = 0
DEL_OWNER:  owner = 11872, lock = 11872, request = 0
DEL_OWNER:  owner = 11376, lock = 11376, request = 0
DEL_OWNER:  owner = 11804, lock = 11804, request = 0

ACTIVE:     owner= xxxxxx, lock= 0, request= 0

```

On Classic, an Event record like the “Active” one shown in the figure might be cause for concern. It indicates that one Classic process got a mutex for access to the resource, wrote its owner ID into the lock header block and then got killed while it still held the lock table acquisition. However, the secondary lock header block should have enough information to enable a new process to undo any actions left partly-completed by the killed process.

Interactive sampling

The interactive time-series lock activity reports that *fb_lock_print* generates with the *-i* switch measure Lock Manager performance. The layout is modeled after the UNIX “sar” utility (System Activity Reporter). The output illustrated in Figure 41-13 sampled every 4

seconds for 10 intervals The sampling depicted in Figure 38.13 was generated on a Classic version of Firebird running four local processes with a significant number of conflicts:
`fb_lock_print -ia 4 10`

Figure 38.17 Interactive sampling

	(1)	(2)	(3)	(4)	(5)
	acquire/s	acqwait/s	%acqwait	acqrtry/s	rtrysuc/s
14:32:55					
14:32:59	15654	110	0	0	0
14:33:03	16328	91	0	0	0
14:33:07	16639	106	0	0	0
14:33:11	15467	115	0	0	0
14:33:15	14864	87	0	0	0
14:33:19	15292	117	0	0	0
14:33:23	14939	85	0	0	0
14:33:27	14992	103	0	0	0
14:33:31	15660	124	0	0	0
14:33:35	14800	114	0	0	0
Average:	15463	105	0	0	0

“Acquire” statistics from `fb_lock_print -ia` :

1	acquire / s	Average number of attempts to acquire the lock table per second.
2	acqwait / s	Average number of acquire attempts that were forced to wait each second.
3	%acqwait	Percentage of attempts that were forced to wait.
4	acqrtry / s	Average number of retries following a spin wait for the lock table mutex per second (SMP machines only, in theory).
5	rtrysuc / s	Average number of successful retries per second.



For clues as to what the columns in the many varieties of interactive reports mean, read the white paper **Reading a Lock Print**, by Ann Harrison, at www.ibphoenix.com. Ann’s contributions to this topic were significant.

Lock configuration settings

The Lock Manager's default settings should suit most environments initially. Under load, especially with the Classic server, there may be benefits in tuning the settings to improve throughput or to resolve deficiencies in the lock resources.

The following parameters will be of interest with regard to configuring locking on your Firebird server.

Table 38.14 Lock-related configuration parameters

firebird.conf parameter	isc_config/ibconfig (v.1.0)	Comments
<u>LockAcquireSpins</u>	lock_acquire_spins	
<u>LockHashSlots</u>	lock_hash_slots	
<u>LockMemSize</u>	lock_mem_size	
<u>LockSemCount</u>	any_lock_sem_count	Pre-2.5 versions only
<u>LockSignal</u>	any_lock_signal and v4_lock_signal	Pre-2.5 versions only
<u>LockGrantOrder</u>	lock_grant_order	Deprecated, retained for testing purposes only

Details of these parameters are in Chapter 34, [*Configuration Parameters in Detail*](#).

BACKING UP DATABASES

Regular backup is essential for good housekeeping, data integrity, security and disaster fallback. Firebird comes with two backup utilities: *gbak* and *nBackup*. The two utilities work quite differently and do not interact with each other in any way.

A full Firebird database backup with *gbak* saves the entire metadata and data of a database compactly to a file on a hard disk or other storage medium. In the process, it performs some important housekeeping on the database itself.

Backup with *nBackup* is incremental, a mechanism that scans the pages of the database and makes snapshots of changed pages at configured intervals. Because it never touches “data as data”, *nBackup* backs up the contents of database pages warts and all. Its primary purpose is backup.

Also discussed in this chapter are shadowing—which is not safe to rely on as a sole backup solution—and some broad comments about using the “warm backup” capabilities of a replication module for rapid, reliable disaster fallback.

gbak or *nBackup*?

Although operating systems usually include facilities to archive database files, it is strongly recommended that one or both of the Firebird utilities be used for backups and that your database files be excluded from any other backup regime on the system. However, *nBackup* comes with a useful ancillary tool that lets you “freeze” a database for writes while you use a third-party backup utility. For users, life goes on: the tool redirects any writes temporarily to a “delta” file, which it merges back into the main database when the freeze lock is released.

While *nBackup*’s incremental backup is preferred for large databases, it does not have the means to dispose of the obsolete record versions that may have been left behind by the engine’s own garbage collection mechanisms. For that reason, it may be necessary to do a full *gbak* backup periodically for housekeeping purposes and perhaps also to restore a fresh database from it.

Use of one does not exclude the other. When a *gbak* or other (third-party) backup is needed, the *nBackup* process can be diverted temporarily to a ‘delta’ file and brought back into synch when *gbak* is done.

Inherent issues with client applications that manage transactions poorly or sites that suffer from network instability or end-user carelessness will usually require *gbak* to be the utility of choice until those problems are expunged. For small to medium sized databases, especially on less well-resourced hardware, *nBackup* may be ‘overkill’, anyway.



Use of nBackup is not recommended on heavily-loaded Classic installations on servers prior to v.2.5.

Alternative backup utilities

Never rely on backups of your Firebird databases made by operating system utilities, specialised backup tools, file-copying methods or compression utilities like gzip or WinZip, unless the database is completely shut down or “frozen” by the *nBackup* tools. Such backups will, like *nBackup*, store uncleared garbage that would be restored along with the rest of the file. However, because these utilities usually place low-level locks on blocks of disk, with no awareness of the internal layout of a Firebird database file, their use is a recognized cause of corruption if they are allowed to run while there are live connections writing to the database.

The *gbak* Utility

The command-line backup and restore utility, *gbak*, creates backups of your databases that are (by default) platform-independent. The same utility, running on the same or another server, also restores a fresh database from this stable snapshot archive. Backup files made by *gbak*—often named, by convention only, with the suffix “.fbk”—may be safely backed up by the host’s system utilities or a third-party backup utility.

Firebird’s backup is a “hot” backup: normal database activity can continue whilst *gbak* is analyzing the metadata and writing the backup file. The operation runs in a snapshot transaction and captures database state as it stands when the transaction begins. Data changes that clients commit to the database after the backup begins are not recorded in the backup file.



Backup files should be transferred to a portable storage medium and stored at a secure physical location remote from the server.

About *gbak* Backup Files

The *gbak* utility analyzes and decomposes Firebird database files, storing cleaned metadata and data separately, in a compact format. A backup made with *gbak* is not a database file and will not be recognized by the server. In order to become useable, it must be restored to a Firebird format that is readable by the server, using the version of *gbak* that corresponds to the version of Firebird server that is running.

When restoring its files back to database format, *gbak* performs validation of both metadata and data before using query-language commands internally to reconstruct the database and repopulate it with its data.

If corrupt structures or data are detected, *gbak* stops restoring and reports the condition. Its ability to analyze problems makes it an invaluable helper when recovery of broken databases is being attempted. If you are in this unfortunate situation, refer to Appendix IX, *Database Repair How-To*.



Making backups is but one part of a data protection scheme. If you have blind faith in the integrity of backups then, regardless of the backup system you use, you will be tempting Fate. The fact that a backup completes is no guarantee that it will restore. Always make test restores as essential and regular a part of your backup and data assurance regime as the backups themselves.

***gbak*'s other talents**

gbak also performs an important sequence of other utility tasks in the course of analyzing, storing and restoring your database. Some are automatic; others can be requested by means of switches in the command-line call when you invoke the program.

- Backup performs garbage collection on outdated records during backup—an optional switch, enabled by default. If enabled, this housekeeping occurs even if you don't restore the backup and begin working with a fresh file.

Restore tasks can include:

- balancing indexes, to refresh the performance capability of your database
- reclaiming space occupied by deleted records and packing the remaining data. This often reduces database size and improves performance by "packing" data.
- optionally changing the database page size on restore
- changing the database owner—optional—but watch out! it can happen by accident if you are not careful
- upgrading an InterBase® database to Firebird or a lower version Firebird database to a higher version, i.e. upgrading the on-disk structure (ODS)
- splitting the database into multiple files or resizing existing multiple files—optional
- distributing a multi-file database across multiple disks—optional

Upgrading the on-disk structure

New major releases of the Firebird server are likely to feature changes that alter the on-disk structure (ODS). If the ODS has changed and you want to take advantage of any new Firebird features, upgrade your databases to the new ODS.

A new server version can access databases created with some previous versions, but won't be able to share any of its new features with an older database with a lower ODS. All the same, it is recommended that you do follow the upgrade procedure on your databases when you do a transition to a newer server version. Otherwise, what is point of upgrading the server?

To upgrade existing databases to a new ODS, perform the following steps:

- 1 Before installing the new ODS version of Firebird, back up databases using the old version of *gbak*
- 2 Install the new version of the Firebird server as described in Chapter 2.
- 3 Once the new version is installed, restore the databases using the *gbak* that is installed along with it, in the \bin directory beneath the Firebird root.

For a much more detailed discussion of migrating systems to a newer server, refer to Chapter 5, *Migration Notes*.

Dialect 1 databases

Legacy databases that started life last century under InterBase v.4 or v.5 had several non-standard data types. When InterBase 6, the ancestor of Firebird, introduced 64-bit integers and a number of changes to SQL language constructs, the language of the old databases became known as “SQL dialect 1”.

A dialect 1 database remains dialect 1 when it is backed up and restored on any Firebird server, regardless of the new ODS it restores to.

For a full discussion of the issues and recommendations for changing the dialect, refer to the topic *Dialect 1 Databases* in Chapter 5, *Migration Notes*.

Database backup & restore rights

Use of *gbak -backup* is restricted to SYSDBA (or equivalent) and the owner of the database. The login credentials of one of these users must be supplied, either in the backup request command or through the ISC_USER and ISC_PASSWORD environment variables.

Any user can restore a database using the [C]reate switch (see below), as long as the database being created will not overwrite an existing database. If the database is being overwritten then the restrictions are the same as for backup.

Changing ownership of a database

A restored database file, or one created from a *gbak* file, is owned by the user that performed the restore. Backing up and restoring is thus a mechanism for changing the ownership of a database.



Anyone can steal a Firebird database by restoring a backup file to a Firebird server on machine where he knows the SYSDBA password. It is important to ensure that your backup files are secured from unauthorized access.

User name and password

When Firebird checks authority to run *gbak*, it determines the user according to the following hierarchy:

- 1 The user name and password specified as switches in the *gbak* command
- 2 For a local *gbak* run only , the user name and password specified in the ISC_USER and ISC_PASSWORD environment variables, provided they also exist in the security database (security2.fdb, or security.fdb in v.1.5). Keeping these environment variables set permanently is strongly not recommended, since it is extremely insecure.
- 3 If no user credentials are supplied at either of the previous levels, Firebird allows the root user (on POSIX) or a trusted Administrator (on Windows, with Authentication configured in firebird.conf) to run *gbak* locally.

v.1.X Windows Administrator authority is not implemented in Firebird 1.5 or 1.0 versions.

Suppressing database triggers

Database triggers, introduced in Firebird 2.1, might cause problems for *gbak* when it attaches to the database. If you need to suppress them for *gbak*'s connection, use the `-nod[btriggers]` switch.



Database trigger suppression is available only to SYSDBA or equivalent and the database owner.

Running a Backup

To invoke *gbak*, either change to the Firebird `/bin` directory where *gbak* is located, or use an absolute root path. The wrapped portions of lines are shown as indented but the entire command must be on a single line.

In the following syntax patterns and examples, the assumption is that *gbak* is being called locally, on a server version that supports a “serverless” connection. For Superserver on POSIX, the `-se hostname:service_mgr` switch is mandatory.

POSIX:

```
$) ./gbak -b[ackup] <options> source target [n]
```

or

```
$) /opt/firebird/bin/gbak -b[ackup] <options> source target [n]
```

Windows:

```
C:\Program Files\Firebird\Firebird_2_5\bin> gbak -b[ackup] <options> source target [n]
```

or

```
C:\> C:\Program Files\Firebird_2_5\bin\gbak -b[ackup] <options> source target [n]
```



*The `-b[ackup]` switch is the default switch for *gbak* and is optional.*

Arguments for gbak -b[ackup]

source is the full path and file name of a database to back up. It can be an alias—and this is recommended. If backing up a multi-file database, use only the name of the first (primary) database file or the alias for that file.

target is the full path and file name of a storage device or backup file to which the backed up database is to be sent. A backup file does not have to obey the location restrictions that apply to database files. The location can be anywhere in the network.

In the case that the backup is being output to multiple files, there will be multiple targets. The token *n* is an integer parameter, included with each output file except the last, to indicate the length of the file in (by default) bytes. A lower-case character can be appended to the number to specify that size is in Kilobytes (k), Megabytes (m) or Gigabytes (g). Refer to the examples below.

On POSIX, target can also be `stdout`. In this case *gbak* writes its output to the standard output (usually a pipe)

<options> can be a valid combination of switches from Table 39.1, below. The switches are case-insensitive.

Table 39.1
 gbak Backup switches

Switch	Effect
<code>-b[ackup_database]</code>	Causes <i>gbak</i> to run a backup the named database to file or device
<code>-co[nvert]</code>	Converts external files into internal tables. On restore, any external tables are converted to intra-database tables and the association with the external file will be gone.
<code>-e[xpand]</code>	Creates the backup without compression
<code>-fa[ctor] n</code>	Uses a blocking factor <i>n</i> for a tape device.
<code>-fe[tch_password] file-name</code>	For use instead of <code>-password</code> if <i>gbak</i> is to fetch the password from a file. On POSIX, if <i>file-name</i> is given as <code>stdin</code> , the user will be prompted to enter the password.
	Not available prior to Firebird 2.5.
<code>-[no_]g[arbage_collect]</code>	Suppresses garbage collection during backup. Use this switch if you are planning to restore the database from the backup into immediate use afterwards. <i>Gbak</i> doesn't store garbage so it doesn't make sense to add the extra overhead if you are not going to use the old database afterwards.
<code>-i[gnore]</code>	Causes checksums to be ignored during backup. You can use this switch on a re-run of a backup that failed from checksum errors
<code>-l[imbo]</code>	(Lower-case letter “L”) causes limbo transactions to be ignored. Don't use this switch for regular backups. It is available for cleanup after a two-phase transaction has failed due to losing a server before commits have taken place.
<code>-m[etadata]</code>	Backs up just the metadata—no data are saved. It can be a quick way to get an “empty” database in preparation for deploying into production.
<code>-nod[bttriggers]</code>	Suppress database triggers when reading data. Not valid prior to v.2.5.
<code>-nt</code>	Creates the backup in non-transportable format. By default, the data stored in <i>gbak</i> files are in XDR format, a standard protocol for porting data across platforms.
<code>-ol[d_descriptions]</code>	A deprecated switch—it backs up metadata in an old InterBase® format
<code>-pass[word] password</code>	Checks for password <i>password</i> before accessing the database. This is required (along with a user name) for remote backups and also locally, where <code>ISC_USER</code> and <code>ISC_PASSWORD</code> environment variables are not available. NOTE that the minimum abbreviated form of the <code>-password</code> switch (<code>-pass</code>) for <i>gbak</i> is different to that for <i>isql</i> (<code>-pas</code>).

Switch	Effect
<code>-ro[le]</code> <i>role-name</i>	SQL role log-in, if required. Currently, used for logging in as RDB\$ADMIN. It will except if the <code>-user</code> argument contains the name of a user that has not been granted RDB\$ADMIN in the database being (re)created and also does not have global rights via AUTO ADMIN MAPPING (see <i>ALTER ROLE statement</i> in Chapter 37, Database Security). Present in all versions but not active prior to 2.5.
<code>-se[rvic]</code> <i>servicename</i>	Creates the backup files on the host where the original database files are located. The <i>servicename</i> argument invokes the Service Manager on the server host. For details of syntax, refer to the topic <i>Using gbak with the Firebird Services Manager</i>
<code>-t[ransportable]</code>	Stores the gbak data in XDR transportable format. It is the default and may be omitted. To store data in a compressed native format, use the <code>-nt</code> switch.
<code>-tru[sted]</code>	Use “trusted user” authentication on Windows, instead of user name and password. Works only if the Windows server is configured for this style of authentication, in <i>firebird.conf</i> . Not available in versions 1.0.x, 1.5.x or 2.0.x.
<code>-user</code> <i>name</i>	Checks for user name before accessing the database. This is required (along with a password) for remote backups and also locally, where ISC_USER and ISC_PASSWORD environment variables are not available.
<code>-v[erify]</code>	Provides a detailed trail of what gbak does. You can optionally output the text to a file, using the <code>-y</code> switch (see below).
<code>-y {filespec suppress_output}</code>	Directs status messages to <i>filespec</i> , a fully-qualified path to the file you want to have created. It will fail if the named file already exists. If the backup finishes normally and the <code>-v(erbose)</code> switch isn't used, the file will be empty. <code>suppress_output</code> can be used instead for a “silent” backup with no output messages at all.
<code>-z</code>	Shows the version numbers of both <i>gbak</i> and the Firebird engine.

Transportable backups

Accept the default `-transportable` switch if you operate in a multi-platform environment. It writes data in the cross-platform standard XDR format, allowing the file to be read by the *gbak* program on a platform different to the one on which it was backed up.

Cross-version backups

The *gbak* program on servers with lower on-disk structure (ODS) than the Firebird server that created a database will generally NOT be able to restore from a backup made with a newer major version of *gbak*. Minor version mismatches should be fine.

Trying to back up with a version of *gbak* that does not match the server version on which the database was created or last restored risks failure if the program was written for an ODS lower than that of the database. However, it has a reasonable chance of succeeding if the database does not use structures of features unknown to the older program.

For the “2” series, efforts were made to make *gbak* more version-friendly. If you need to “downgrade” a database that has been upgraded from an earlier version and ODS, it can be at least worth trying to run the older *gbak* version under the newer server in an attempt to make a backup that the same *gbak* version can restore under the older server.

Backing up to a single file

For a simple local backup of a single-file or multiple-file database:

```
gbak -b d:\data\ourdata.fdb d:\data\backups\ourdata.fbk
```

The source name is the same whether the database you are backing up is a single or a multiple-file database. When you are backing up a multi-file database, specify only the **first** file in the backup command. The paths to the second and subsequent files will be found by *gbak* in the database and file headers during the course of the backup.



If you supply the name[s] of the secondary database file[s], they will be interpreted as backup file names.

The target file can have any name you wish and be located anywhere in the network, as long as it is valid on the filesystem to which it is being written.

Backing up a multi-file database to multiple files

When you back up a multi-file database to multiple *gbak* files, there is no requirement to match the database file-for-file with the backup. If there is to be more than one target file, the names and sizes of the target files need to be specified for all except the last file in the set. By default, the file size (an integer) is taken to be in bytes. To modify that, append a lower-case character to tell *gbak* you intend the size to be in Kilobytes (k), Megabytes (m) or Gigabytes (g).

The following command backs up a database to three backup files on different filesystem partitions and writes a verbose log. It is all one command: the indented portions are wrapped here for easier reading.

POSIX:

```
./gbak -b /data/accounts.fdb /backups/accounts.fb1 2g  
      /backups2/accounts.fb2 750m /backups3/accounts.fb3  
      -user SYSDBA -password m1llp0nd  
      -v -y /logs/backups/accounts.20040703.log
```

Windows:

```
gbak -b d:\data\accounts.fdb e:\backups\accounts.fb1 2g  
      f:\backups2\accounts.fb2 750m g:\backups3\accounts.fb3  
      -user SYSDBA -password m1llp0nd  
      -v -y d:\data\backuplogs\accounts.20040703.log
```

Single-file database to multiple targets

If you back up a single-file database to multiple targets, the syntax is identical. In fact, *gbak* is not interested in whether your source database is single or multiple-file.

Points to note

- The backup will fail if any designated target file is smaller than 2048 bytes. If you are logging, the reason will appear in the log.

- *gbak* fills the named target files in left-to-right order. It won't begin the next file until the previous one has reached its designated capacity. In the example above, `accounts.fb3` will not be created if `accounts.fb2` is not filled.
- file paths for targets need not be physically controlled by the host but, if you are using the `—service` switch (refer below) on systems where file permissions are in force, your user profile must and have the appropriate write permissions.

Metadata-only backup

A metadata-only backup is typically needed for creating an “empty” database when you are preparing to deploy a system into production, load QA data or reconstitute a database by pumping existing data. The following command does a metadata-only backup of our `accounts` database:

```
gbak -b -m d:\data\accounts.fdb e:\QA\accounts.fbk
```

Remote and localhost backups

With the `—se[rvice]` switch, you can invoke the Service Manager on the remote server from a client node. If you want the backup file to be created locally at the client, you will need to map a location that can be accessed from Firebird's host node.

Superserver on POSIX platforms does not support “serverless” connections for any client. That means using the `—se[rvice]` switch always for any applications—including *gbak*—that attach to a local database.

See the topic [*Using gbak with the Firebird Services Manager*](#), below, for more information.

Security considerations

It is a good precaution to set the read-only property on your backup files at the filesystem level after creating them, to prevent them from being accidentally or maliciously overwritten. You should move backup files to a storage device that is physically separate from the disk where the database lives.

You can protect your databases from being “kidnapped” on UNIX and on Windows NT systems by placing the backup files in directories with restricted access.



Backup files that are kept on Windows 95/98/ME systems, or in unrestricted areas of other systems, are completely vulnerable.

Return codes and feedback

A backup run on Windows returns an errorlevel of 0 on success, 1 on failure. If an error occurs, check the `firebird.log` file (`interbase.log` in v.1.0.x). For a full backup log file, use the `—y` and `—v` switches.

Running a Restore

The syntax pattern for the restore options is:

POSIX:

```
$] ./gbak {-c[reate] | -r[ecreate_database [ 0[VERWRITE] ] | -rep[lace_database] }  
      <options> source target
```

or

```
$] /opt/firebird/bin/gbak {-c[reate]
```

```
| -r[ecreate_database [ 0[OVERWRITE] ] | -rep[lace_database ]
<options> source target
```

Windows:

```
C:\Program Files\Firebird\Firebird_2_5\bin> gbak {-c[reate]
| -r[ecreate_database [ 0[OVERWRITE] ] | -rep[lace_database ]
<options> source target
```

or

```
C:\> C:\Program Files\Firebird\Firebird_2_5\bin\gbak {-c[reate]
| -r[ecreate_database [ 0[OVERWRITE] ] | -rep[lace_database ]
<options> source target
```

Arguments for *gbak* restores

source is the fully qualified path to a *gbak* backup file.

target must be a fully qualified database path or a valid alias pre-declared in *firebird.conf*.

options must include user name (*-u[ser]*) and password (*-pas[sword]*), unless you have the environment variables *ISC_USER* and *ISC_PASSWORD* active and set to the log-in of the user you want as Owner of the restored database. If you are exercising your rights as a user of the *RDB\$ADMIN* role, the *-ro[le]* option must be used as well.

Intended behaviour

The intended behaviour of the restore switches is as follows:

- c[reate_database]* creates a fresh database if the target database does not exist already.
- r[ecreate_database]* creates the target database if it does not exist already, unless the *-OVERWRITE* option is present. If so, it overwrites the database that is there.
- rep[lace_database]* creates a fresh database if the target database does not exist already; otherwise it overwrites the target database.



*The *-rep[lace_database]* switch is currently supported only to provide a degree of backward compatibility. It will become unavailable in some future Firebird release.*

WARNING! **V.1.X versions**

The *-recreate_database* switch is not supported in the old v.1.0.x and v.1.5.x versions. In these old versions, if the *-r* (or *-R*) switch is used, or any string up to the full “*-replace_database*”, the restored database will overwrite the target database if it exists—the same as the current behaviour described above for *-rep[lace_database]* and *-r[ecreate_database] -0[OVERWRITE]*.

If the restore should fail or otherwise not complete for any reason, your database will be gone.

Tip for migrators

Any legacy batch or cron scripts that rely on “*gbak -r*” or “*gbak -R*” will except if the target database exists. If you are migrating your system to the “2” series or higher and want your script to retain the ability to overwrite your database unconditionally, you will need to modify the command to use *-recreate_database -OVERWRITE* or correct the short form for *-replace_database*. from *-r* to *-rep*.

Restore or Create?

The notion of “restoring a database” by overwriting it was born in another age, when disk space was worth more than hiring an expert to reconstruct a broken database or taking on a team of data-entry staff to reconstruct the company’s system from paper records.

In short, overwriting a database whose survival you care about is not recommended under any circumstances. Ponder these painful facts:

- If a restore fails to complete, the overwritten database is gone forever—and things can go wrong with any restore
- Restoring over a database that is in use will cause corruption
- Allowing users to log in to a partly-restored database will also cause corruption

If you decide to use one of the potential database-killers (**-r[ecreate_database]** or **-r[eplace_database] -o[verwrite]**) anyway, despite the risks, then you can only do so if you supply the login credentials of the database owner, or of SYSDBA or a user with escalated privileges.

Any user authenticated on the server can restore using the **-c[reate]** option. Consider the implications of that fact and take the appropriate precautions to keep your backups out of the wrong hands.

Restore switches

Table 39.2 *gbak* switches for Restores

Switch	Effect
-c[reate_database]	Restores database to a new file.
-b[uffers]	Sets default cache size (in database pages) for the restored database.
-fix_fss_D[ata]	For use during a migration restore of a pre-v.2.1 database: fixes malformed UNICODE_FSS user data. Not valid in versions prior to 2.5
-fix_fss_M[etadata]	For use during a migration restore of a pre-v.2.1 database: fixes malformed UNICODE_FSS metadata. Not valid in versions prior to 2.5
-f[etch_password] file-name	For use instead of -password if <i>gbak</i> is to fetch the password from a file. On POSIX, if <i>file-name</i> is given as <i>stdin</i> , the user will be prompted to enter the password.
-i[nactive]	Not available prior to Firebird 2.5. Makes indexes inactive in the restored database. Useful when retrying a restore that failed because of an index error
-k[ill]	Suppresses recreation of any shadows that were previously defined
-m[eta_data]	Restores only the metadata.
-mo[de] {[read_write read_only]}	Specifies whether the restored database is to be read-only or read-write. Possible values are <i>read_write</i> (default) or <i>read_only</i>

Switch	Effect
–nod[bttriggers]	Suppress database triggers when writing data. Not valid prior to v.2.5.
–n[o_validity]	Deletes validity constraints from restored metadata. Use if you need to retry a restore that failed because of CHECK constraint violations.
–o[ne_at_a_time]	Restores one table at a time. Can be used for partial recovery if a database contains corrupt data.
–pa[ge_size] n	Resets page size to n bytes (1024, 2048, 4096, 8192 or 16384). Default is 4096. A page size of 16384 is not possible if the filesystem does not support 64-bit file I/O.
pass[word] <i>password</i>	Checks for password <i>password</i> , along with –u[ser], before attempting to (re)create the database
–r[ecreate_database] [-O[VERWRITE]]	Restores database. If the -overwrite option is present, an existing database file with the same name as the target will be overwritten; if not, creates a new file with the target name. Not valid in versions 1.X.
–rep[lace_database]	Restores database, replacing the existing file matching the named target, if it exists; if not, creates a new file with the target name. This switch is deprecated from the “2” series onward. Abbreviated form not valid in versions 1.X.
–r[eplace_database]	Firebird 1.0.x and 1.5.x only—same as –rep[lace_database] in “2” series and onward.
–ro[le] <i>role-name</i>	SQL role log-in, if required. Currently, used for logging in as RDB\$ADMIN. It will except if the -user argument contains the name of a user that has not been granted RDB\$ADMIN in the database being (re)created and also does not have global rights via AUTO ADMIN MAPPING (see ALTER ROLE statement in Chapter 37, Database Security). Present in all versions but not active prior to 2.5.
–se[rvice] servicename	Creates the restored database on the host where the backup files are located. Use this switch if you are running gbak from a remote node and want to restore from a backup that resides on the same server as the database. It invokes the Firebird Service Manager on the server host and saves both time and network traffic. More details in Using gbak with the Firebird Services Manager , below.
–tru[sted]	Use “trusted user” authentication on Windows, instead of user name and password. Works only if the Windows server is configured for this style of authentication, in firebird.conf. Not available in versions 1.0.x, 1.5.x or 2.0.x.
–user name	Checks for user name, along with –pa[ssword], before attempting to recreate the database

Switch	Effect
<code>-use[_all_space]</code>	Restores database with 100% fill ratio on every data page, instead of the default 80% fill ratio. This is ideal for read-only databases, since they don't need to keep reserve space on database pages to allow for variations in data size as rows are inserted, updated and deleted. To revert a restored database to normal fill ratio, use the <code>gfix -use reserve</code> switch, i.e. <code>gfix -use reserve</code>
<code>-v(erify)</code>	Provides a detailed trail of what gbak does. You can optionally output the text to a file, using the <code>-y</code> switch (see below).
<code>-y {filespec suppress_output}</code>	Directs status messages to <code>filespec</code> , a fully-qualified path to the file you want to have created. It will fail if the named file already exists. If the backup finishes normally and the <code>-v(erify)</code> switch isn't used, the file will be empty. <code>suppress_output</code> can be used instead for a “silent” backup with no output messages at all.
<code>-z</code>	Shows the version numbers of both gbak and the Firebird engine.

User-defined objects

When restoring a backup file to a server other than the one on which it was backed up, you must ensure that any character sets and collations referenced in the backup file exist on the destination server. The backup will not be able to be restored if the language objects are missing.

External function and BLOB filter libraries referred to by declarations in the database, similarly, must be present for a restored database to work without errors.

Restoring to a single file

The following command performs a single restore from one backup file to one database file:

```
gbak -c d:\data\backups\ourdata.fbk d:\data\ourdata_trial.fdb
```

Multiple-file restores

Single or multiple backup files can be restored to single- or multiple-volume database files. There is no requirement to have a one-to-one correspondence between a backup file volume and a database file volume.

When restoring from a multi-file backup, you must name all of the backup files, in the order in which they were backed up—*gbak* complains noisily if it gets the list in the wrong order or if a volume is missing.

For the target (database) files, you need to supply a size parameter, in database pages, for all files except the last. The minimum value is 200 database pages. The last file always expands as needed to fill all available space.

Single-volume backup restored to multi-volume database

POSIX:

```
./gbak -c /backups/stocks.fbk /data/stocks_trial.fd1 262144
/data/stocks_trial.fd2 262144 /data/stocks_trial.fd3
-user SYSDBA -password m1llp0nd
-v -y /logs/backups/stocks_r.20040703.log
```

Windows:

```
gbak -c e:\backups\stocks.fbk d:\data\stocks_trial.fd1 262144
d:\data\stocks_trial.fd1 262144 d:\data\stocks_trial.fd1
-user SYSDBA -password m1llp0nd
-v -y d:\data\backuplogs\stocks_r.20040703.log
```

If you specify several target database files but have only a small amount of data, the target files are initially quite small—around 800KB for the first one and 4KB for subsequent files. They grow in sequence, to the specified sizes, as you populate the database.

Multi-volume backup restored to single-volume database

POSIX:

```
./gbak -c /backups/accounts.fb1 /backups2/accounts.fb2 /backups3/accounts.fb3
/data/accounts_trial.fdb -user SYSDBA -password m1llp0nd
-v -y /logs/backups/accounts.20040703.log
```

Windows:

```
gbak -c e:\backups\accounts.fb1 f:\backups2\accounts.fb2
g:\backups3\accounts.fb3 d:\data\accounts_trial.fdb
-user SYSDBA -password m1llp0nd
-v -y d:\data\backuplogs\accounts.20040703.log
```

Multi-volume backup restored to multi-volume database

POSIX:

```
./gbak -c /backups/accounts.fb1 /backups2/accounts.fb2 /backups3/accounts.fb3
/data/accounts_trial.fdb 500000 /data/accounts_trial.fdb
-user SYSDBA -password m1llp0nd
-v -y /logs/backups/accounts.20040703.log
```

Windows:

```
gbak -c e:\backups\accounts.fb1 f:\backups2\accounts.fb2
g:\backups3\accounts.fb3 d:\data\accounts_trial.fdb 500000
d:\data\account_trial.fdb
-user SYSDBA -password m1llp0nd
-v -y d:\data\backuplogs\accounts.20040703.log
```

Return codes and feedback

A restore on Windows returns an errorlevel of 0 on success, 1 on failure. If an error occurs, check the `firebird.log` file (`interbase.log` on v.1.0.x).

Page size and default cache size

You can optionally change the page size when restoring, by including a `-p[age_size]` parameter switch in the command, followed by an integer representing the size in bytes. Refer to Table 39.2, *gbak switches for Restores*, for valid page sizes.

This example tells gbak to restore the database with a page size of 8192 bytes:

```
gbak -c -p 8192 d:\data\backups\ourdata.fbk d:\data\ourdata_trial.fdb
```

Similarly, you can use a restore to change the default database cache size (in pages, or “buffers”):

```
gbak -c -b 10000 d:\data\backups\ourdata.fbk d:\data\ourdata_trial.fdb
```

Page size and performance

The size of a restored database is specified in database pages. The default size for database files is 200 pages. The default database page size is 4Kb, so if the page size has not been changed, the default database size is 800KB. This is sufficient for only a very small database.

Changing the page size can improve performance in the following conditions:

- Firebird performs better if rows do not span pages. Consider increasing the page size if a database contains any frequently-accessed tables with long rows of data.
- If a database contains any large indexes, a larger database page size reduces the number of levels in the index tree. The smaller the depth of indexes, the faster they can be traversed. Consider increasing the page size if index depth is greater than three on any frequently used index.
- Storing and retrieving BLOB data is most efficient when the entire blob fits on a single database page. If an application stores many BLOBs exceeding 4Kb, a larger page size reduces the time for accessing blob data.
- Reducing the page size may be appropriate if most transactions involve only a few rows of data, because the volume of data needing to be passed back and forth is lower and less memory is used by the disk cache.

Using gbak with the Firebird Services Manager

The `[-se]rvice` switch invokes the Service Manager on a remote network server, which can be the local loopback server, localhost.

- Superserver on POSIX platforms must use the Service Manager, even if the backup is to be “local”.
- For any remote backups, the Service Manager must be used.



Although you can run gbak backup remotely and have the backup written to a networked location other than the host, the savings in resources and traffic can be significant if the backup file is written to a disk on Firebird's host server, compressed with a utility and passed back to the remote requester by file transfer.

When you run *gbak* with the `-service` switch, it causes *gbak* to invoke the backup and restore functions of Firebird's Service Manager on the server where the database resides.

You can back up to multiple files and restore from multiple files using Service Manager.

The `-se[rvice]` switch takes an argument that consists of the host name of the server concatenated to the constant string `service_mgr` by a special network symbol. The syntax of this argument varies according to the network protocol you are using:

- TCP/IP: `hostname:service_mgr`
- Named pipes: `\\hostname\\service_mgr`

Backup

In this example, we back up a database residing on a remote Windows server's D: drive, to a backup file on the F: drive of the same remote machine. We use the `-verify` option to pass the verbose output of the operation to a log file in another directory. As usual, the example is a single string, parts indented for easier reading:

```
gbak -b -se hotchicken:service_mgr
      hotchicken:d:\data\stocks.fdb
      f:\backups\stocks.20040715.fbk
      -user sysdba -pas m1llp0nd -v -y f:\backups\logs\stocks.20040715.log
```



Switch order matters when you use the `-se` switch. If you want to specify a log file, make sure that you place it after the host service has been identified, to avoid the command failing because the location for the log file could not be found.

Backing up for restoring on POSIX

The user that is current on the server when the Service Manager is invoked to perform the backup—either root or firebird—is the owner the backup file at the filesystem level, causing it to be readable to only that user.

When you need to restore a database on a POSIX server that has been backed up using the Service Manager, you must either use the Service Manager for the restore or be logged in to the system as that file owner.

When *gbak* is used locally with the embedded client, `libfbembed.so`, the `-service` option is not used and the filesystem ownership of the backup file is attributed to the login of the user that ran *gbak*.

Keep these constraints in mind when planning to back up a database on POSIX. They do not apply on Windows platforms.

Restore

The next example restores a multi-volume database from the `/january` directory of the hotchicken server to the `/currentdb` directory. It uses the `-r[ecreate_database]` switch with `-o[verwrite]` and will overwrite the database `magic.fdb` if it is found in `/currentdb`. The first two files of the restored database are 500 pages long and the last file grows as needed.

```
gbak -r -o -user frodo -pas pipeweed -se hotchicken:service_mgr
      /january/magic1.fbk /january/magic2.fbk /january/magicLast.fbk
      /currentdb/magic.fdb 500 /currentdb/magic.fd2 500
      /currentdb/magic.fd3
```

The next example executes on server hotchicken and restores a backup that is on hotchicken to another server called icarus:

```
gbak -c -user frodo -pas pipeweed -se hotchicken:service_mgr
      /january/magic.fbk icarus:/currentdb/magic.fdb
```


gbak Error Messages

Table 39.3 Error messages from *gbak* backup and restore

Error Message	Causes and Suggested Actions to Take
Array dimension for column <string> is invalid	Fix the array definition before backing up
Bad attribute for RDB\$CHARACTER_SETS	An incompatible character set is in use
Bad attribute for RDB\$COLLATIONS	Fix the attribute in the named system table
Bad attribute for table constraint	Check integrity constraints; if restoring, consider using the -no_validity option to delete validity constraints
Blocking factor parameter missing	Supply a numeric argument for “factor” option, e.g on a tape backup device
Cannot commit files	Database contains corruption or metadata violates integrity constraints. Try restoring tables using -one_at_a_time option, or delete validity constraints using -no_validity option
Cannot commit index <string>	Data might conflict with defined indexes. Try restoring using “inactive” option to prevent rebuilding indexes
Cannot find column for blob ..	Try using -one_at_a_time to find the problem table.
Cannot find table <string> ..	ditto
Cannot open backup file <string>	Correct the file name you supplied and try again
Cannot open status and error output file <string>	Messages are being redirected to invalid file name, or a file that already exists. Check format of file or access permissions on the directory of output file; or delete the existing file; or choose another name for the log file.
Commit failed on table <string>	Data corruption or violation of integrity constraint in the specified table. Check metadata or restore “one table at a time”
Conflicting switches for backup/restore	A backup-only option and restore-only option were used in the same operation; fix the command and execute again
Could not open file name <string>	Fix the file name and re-execute command
Could not read from file <string>	Fix the file name and re-execute command
Could not write to file <string>	Fix the file name and re-execute command
Datatype n not understood	An illegal data type is specified somewhere. Check metadata and, if necessary, retry using -one_at_a_time.
Database format n is too old to restore to	The gbak version used is incompatible with the version of Firebird that created the database or the backup. Try backing up the database using the -expand or -old options and then restoring it.
Database <string> already exists.	<p>You used -create or -recreate in restoring a backup file, but the target database already exists. If you actually want to replace the existing database, use the -OVERWRITE switch with -recreate; otherwise, use another database file name.</p> <p>If this error occurs on a v.1.5.X server, use the -replace_database switch if you want to overwrite the existing database.</p>

Error Message	Causes and Suggested Actions to Take
Could not drop database <string> (database might be in use)	In trying to restore a backup file to an existing database target, you tried to use a switch to replace the target, but the database is in use. Either supply a different name for the target database or wait until the existing database is not in use.
Do not recognize record type n ..	Check metadata and, if necessary, restore using <code>–one_at_a_time</code> .
Do not recognize <string> attribute n --continuing ..	A non-fatal data error, but treat it as an alert that warrants attention.
Do not understand BLOB INFO item n ..	?
Error accessing BLOB column <string> --continuing ..	A non-fatal data error, but treat it as an alert that warrants attention.
ERROR: Backup incomplete The backup cannot be written to the target device or file system	Cause could be insufficient space, a hardware write problem, or data corruption
Error committing metadata for table <string>	The table could be corrupt. If restoring a database, try using <code>–one_at_a_time</code> to isolate the table
Exiting before completion due to errors	This message accompanies other error messages and indicates that back up or restore could not execute. Check other error messages for the cause. If you were running a restore that would replace the target database, stop all writing to the host machine and start a file-level retrieval of the deleted database file.
Expected array dimension n but instead found m	There is a problem with an array column somewhere. If the error message or the verbose output log can't help to find the problem table, attempt a “one table at a time” restore to identify it.
Expected backup database <string>, found <string>	Check the names of the backup files being restored against possible spelling errors in your command. On POSIX, consider case-sensitivity of file names.
Expected array version number n but instead found m	There is a problem with an array.
Expected backup description record ...	?
Expected backup start time <string>, found <string> ..	?
Expected backup version 1, 2, or 3. Found n ..	?
Expected blocking factor, encountered <string>	The <code>–factor</code> option requires a numeric argument
Expected data attribute ..	?
Expected database description record ..	?
Expected number of bytes to be skipped, encountered <string> ...	?
Expected page size, encountered <string>	The <code>–page_size</code> option requires a numeric argument. Tip: check that you are using the correct abbreviations for <code>–page_size</code> and <code>–pagesword</code> .
Expected record length ..	You might be trying to restore a non-transportable backup to a different platform than the original.
Expected volume number n, found volume n	(Antiquated): When backing up or restoring with multiple tapes, be sure to specify the correct volume number.

Error Message	Causes and Suggested Actions to Take
Expected XDR record length ..	You might be trying to restore a non-transportable backup to a different platform than the original.
Failed in put_blr_gen_id ..	Possible corruption in the system blob that stores generators (sequences).
Failed in store_blr_gen_id ...	ditto
Failed to create database <string>	The target database specified is invalid; it might already exist.
Column <string> used in index <string> seems to have vanished	An index references a non-existent column. Check either the index definition or column definition
Found unknown switch	An unrecognized <i>gbak</i> option was specified. Tip: it's nothing to worry about if you have used "gbak -help" or "gbak -?" to view the compact online help. Otherwise, it could indicate you are trying to use a newer switch on an old version of gbak.
Index <string> omitted because n of the expected m keys were found ..	?
Input and output have the same name Disallowed.	A backup file and database must have distinct names; correct one of the names and try again.
Length given for initial file (n) is less than minimum (m)	Insufficient space was allocated for restoring a database into multiple files. No action needed, as Firebird automatically increases the page length to the minimum value.
Missing parameter for the number of bytes to be skipped ..	?
Multiple sources or destinations specified	Only one device name can be specified as a source or target.
No table name for data	The database contains data that are not assigned to any table. Use gfix to validate or mend the database. If need be, follow the procedure in Appendix IX, <i>Database Repair How-To</i>
Page size is allowed only on restore or create	The -page_size option was used during a back up instead of a restore. Tip: check that you specified a restore switch. If you omit one, <i>gbak</i> assumes you are requesting a backup.
Page size parameter missing	The -page_size option requires a numeric argument. Tip: check that you are using the correct abbreviations for -pa[ge_size] and -pas[word].
Page size specified (n bytes) rounded up to m bytes	Non-fatal error. Invalid page sizes are rounded up to valid ones: 1024, 2048, 4096, 8192 or 16384, whichever is closest. A page_size of less than 4096 is invalid on Firebird 2.5+ servers.
Page size specified (n) greater than limit (16384 bytes).	Specify a page size of 2048, 4096, 8192 or 16384. A page_size of less than 4096 is invalid on Firebird 2.5+ servers.
Password parameter missing	The backup or restore is accessing a remote host machine. Use the -password switch and specify a password . Check also that you are using the correct abbreviation for -pas[word].
Protection is not there yet	Check your switches. This message indicates that you might have tried to use an undocumented switch for an option that has not yet been implemented.
Redirect location for output is not specified	Check your switches. This message indicates that you might have tried to use an undocumented switch for an option that has not yet been implemented.

Error Message	Causes and Suggested Actions to Take
REPLACE specified, but the first file <string> is a database	If you are attempting a restore with -recreate_database -o or -replace_database, check that the file name following the -replace option is a backup file rather than a database. Tip: remember that the order of the backup file and database path arguments are reversed according to whether you are backing up or restoring.
Requires both input and output file names	Specify both a source and target when backing up or restoring.
RESTORE: decompression length error	Possible incompatibility in the gbak version used for backing up and the gbak version used for restoring. Check whether -expand should have been specified for the backup; or whether you might be trying to restore a non-transportable backup that is incompatible with the platform you are restoring to.
Restore failed for record in table <string>	Possible data corruption in the named table.
Skipped n bytes after reading a bad attribute n ..	Abstruse but non-fatal error.
Skipped n bytes looking for next valid attribute, encountered attribute m ..	ditto
Trigger <string> is invalid ..	?
Unexpected end of file on backup file	Restoration of the backup file failed; the backup procedure that created the backup file might have terminated abnormally. If possible, create a new backup file and use it to restore the database
Unexpected I/O error while accessing <string> backup file	A disk error or other hardware error might have occurred during a backup or restore.
Unknown switch <string>	An unrecognized gbak option was specified. Tip: it's nothing to worry about if you have used "gbak -help" or "gbak -?" to view the compact online help. Otherwise, it could indicate you are trying to use a newer switch on an old version of gbak.
User name parameter missing	Supply a user name with the -user switch
Validation error on column in table <string>	The database cannot be restored because it contains data that violates integrity constraints. Try deleting constraints from the metadata by specifying -no_validity during restore
Warning -- record could not be restored	Possible corruption of the named data
Wrong length record, expected n encountered n ..	Possible incompatibility in the gbak version used for backing up and the gbak version used for restoring. Check whether -expand should have been specified for the backup; or whether you might be trying to restore a non-transportable backup that is incompatible with the platform you are restoring to.

Incremental Backup Tool (nBackup)

nBackup is an on-line backup utility that makes both full and incremental backups. A full backup creates a complete image from the disk of all the database pages at a point in time. It can be used to restore a working image of the database to the same disk or another.

The incremental backups save images of database pages that have changed during a specified interval of time. If a restore is required, the restore tool chains the full backup and the subsequent incremental files together, up to the specified restore point, and reconstitutes the result as a whole, working database.

Because the backup process is not concerned with reading and writing data, it can provide a speedier way to back up large databases, particularly on well-resourced hardware.

nBackup is a tool with two blades. An ancillary task that *nBackup* can perform is freezing the main database file against new writes, to enable the use of some preferred third-party backup software to do a lengthy backup without affecting database performance for active users. While the freeze is active, *nBackup* writes the data changes to a “delta file” that it picks up and merges into the main database in response to a request to “unfreeze”, i.e., to release the write lock.

About nBackup

The primary job of *nBackup* is to maintain incremental backups that, if properly managed, can allow point-in-time recovery after a catastrophic failure of the hard disk where the database resides. It does what it does by monitoring for pages that change, capturing the on-disk image of a page and storing it in a file.

nBackup was originally commissioned by Broadview Software Ltd, a long-time sponsor of Firebird development. The “n” in its name came from the first name of the developer who was hired to write it: Nikolay Samofatov. Broadview contributed the source code to the Firebird Project during the early development of the “2” series and it was first distributed in Firebird 2.0.



One user remarked, “nBackup has a hundred times more documentation in the Bug Fix lists than it has in any manual!” In its wry way, it is fair comment. The nBackup code has been progressively refined as time has passed, to address problems that escaped the QA processes in earlier days. If you intend to use nBackup, be sure to use the newest sub-release of whatever major “2” series version you want to run.



Use of nBackup is not recommended on heavily-loaded Classic servers prior to v.2.5.

What nBackup cannot do

Unlike *gbak*, *nBackup* has no capability to “housekeep” the logical structures *within* a database. Indeed, it faithfully preserves any broken structures.

Hence, its backup function does not clear garbage or release the space on pages that is occupied by obsolete record versions. A restore from an *nBackup* backup file does not “compact” the database by laying it out anew, filling its structures with data and building fresh indexes. It cannot change database header attributes such as page size, Owner, cache size and so on.

nBackup cannot open databases with an on-disk structure lower than 11. If you are unsure of the ODS of your database, use `gstat -h` from the command line to display the header information.

The backup files maintained by *nBackup* are not transportable between incompatible operating systems, hardware architectures or Firebird versions. Unlike *gbak*'s backup files, the *nBackup* files cannot be written to network storage but only to locally-attached devices. Remote backups are not possible.



To the time of this writing, *nBackup* does not support multi-file databases. Do not even attempt to use these tools if your database is multi-file. You risk data loss and corruption from causes and effects that *nBackup* is not capable of detecting.

In short, *nBackup* is a backup utility with benefits for sites with large databases and well-featured hardware. It is not a tool for database housekeeping.

Running nBackup

nBackup is a command-line application that is included in the standard installation. You run its commands directly from within a command shell or you can automate them in a script file.



It is a very good idea to make a realistic assessment of what you need for your backup scheme and capture it in a well-tested schedule of scripts. Don't overlook a means of taking copies of your backup files off-line—even off-site—in a regular and organised fashion. An insurance policy is only good if you keep the premiums up to date!

Along with the other Firebird tools, *nBackup* executable is located in the `/bin/` subdirectory of your installation folder. If you have a default installation, that will be `/opt/firebird/bin/nbackup` on POSIX or

`C:\Program Files\Firebird\Firebird_2_X\bin\nbackup.exe` on Windows (where '2_X' represents the Firebird major version).

Making Backups

Backups can be made at multiple levels, from a “base” level of 0, advancing up levels as needed to achieve the level of granularity that meets requirements.

- Level 0 is a full backup. In a typical scenario, you begin by making this full backup and plan to do incremental backups at one or more higher levels until next time you decide to do a full backup—or until the day when you need to restore your database!
- A level 1 backup is the first level of *incremental* backup for your backup scheme. It backs up all the pages that have changed since the last level 0 (full) backup. Each successive backup file “reaches back” to that last level 0 backup, independently of any level 1 backup files that already exist. If you have a scheme that is faithfully followed, you thus provide the means to enable a restore to a point in time before some known corruption occurred.
- Your backup scheme might involve more levels. For example, you might schedule level 1 backups once a week and level 2 backups daily. Level 2 backs up all the pages that have changed since the last level 1 backup. Each successive backup file “reaches back” to that last level 1 backup, independently of any level 2 backup files that already exist. Thus, depending on the timing of your backups, under this sort of scheme, you could have up to six daily level 2 backups and up to four or five weekly level 1 backups in a month.

- ...and so on, up the levels. In practice, it makes little sense to have more than three or four levels unless you have identified a genuine need for a high degree of granularity in your “fall-back” solution.

Syntax pattern for a backup

The syntax pattern for a backup is:

```
nbackup -U <user> -P <password> -B <level> <database> [<backup-file-path> | stdout]
```

For example, to do a full (level 1) backup of the employee database, assuming you have it aliased as ‘employee’:

```
nbackup -B 0 employee /data/employee_20120516.nbk
```

On Windows, it might be something like:

```
nbackup -B 0 employee d:\data\employee_20120516.nbk
```

The example will perform a full database backup of the database and write the image file to a dedicated storage space on the host machine.



You can use a fully-qualified path for the database, instead of an alias.

The “delta” file

To enable users to keep working in the database whilst a backup is underway, *nBackup* makes the database file read-only and diverts pages that change afterwards to a temporary disk file. The file it creates and writes to is named, by default, with the database name suffixed with “.delta”. This temporary file is thus often referred to as a “delta file”.

When the backup completes, the contents of the delta file are merged back into the main database file and then the delta is deleted.



By default, the delta file is named using a hard algorithm as XXXXXXXX.delta (where XXXXXXXX would be substituted by the main part of the database file name). The file is written to the current working directory.

From v.2.5, it is possible to configure a custom location and name for the delta file. See the notes below about the ADD/DROP DIFFERENCE FILE parameter for the ALTER DATABASE statement.

Suppressing database triggers

Database triggers, introduced in Firebird 2.1, might cause problems for *nBackup* when it attaches to the database. If you need to suppress them, use the -T switch, which is equivalent to the -nodbtriggers switch in *gbak*.



Database trigger suppression is available only to SYSDBA or equivalent and the database owner.

About file names and locations

You should create a location on the host server for your backups. It is not sufficient to create a partition for your backups on a dedicated partition of the SAME disk. It should be on a disk that is physically separate from the one that the database resides on. Make sure enough space is available to accommodate both the full backup and the accumulating “difference” files from the incremental backups.

If you take the option not to provide a backup file path, *nBackup* will use a hard algorithm to name the file for you, e.g., employee-0-20120516-1419.nbk for this full backup example. The “employee” part used the database alias, followed by the level, the date and a numeric suffix representing the time of day. Higher-level (“difference”) files will have their

level infix where this example has “-0-” so that, for housekeeping and restoring, you will be able to track the levels and ages of the files.

By default, the files are written to the current working directory if the destination path is omitted. If you are running *nBackup* from your Firebird `/bin/` directory, that is where the file will land. This is not usually what you need or want. If you wish to avail yourself of this auto-naming option, run *nBackup* from your backup directory, using the absolute path to the application.

Spaces in file paths

Avoid using file paths with names that contain spaces.

The **nBackup -R** syntax takes a space-separated list of file names as the arguments that map the restore sequence. The complications from using file paths containing spaces are obvious.

stdout

If you use `stdout` in the optional target argument, instead of a file path, the backup will go to standard output, from where you can redirect it “somewhere”, to a device file, perhaps, or an application that compresses input.



Don't forget that path names are case-sensitive on many POSIX platforms.

User name and password

The **-U** (user) and **-P** (password) switches are required unless either

- the environment variables `ISC_USER` and `ISC_PASSWORD` are set; OR
- you are logged in as a “superuser” (root or equivalent on POSIX; or an Administrator group member on Windows with suitable privileges enabled in Firebird and the database).

Switch order

The switches (**-B**, **-U** and **-P**), although case-insensitive, are somewhat sensitive to the order of presentation. Arguments for each must be presented right after the switch they apply to and in the correct order. In our syntax pattern

```
nbackup -U <user> -P <password> -B <level> <database> [<backup-file-path>]
```

the items `<level>`, `<database>` and the optional `<backup-file-path>` are all arguments for the **-B** switch. The optional `<backup-file-path>` cannot be omitted if the **-B** switch is not the last switch presented.

“Difference” backups

Once you have the full backup in place, you can begin making the level 1 and higher backups. Although, like the level 0 backup, the higher level backups scan the entire database, they actually write only the changed pages to disk, along with a few pieces of essential file-level information about the structure of the backup “chain”.

The files written are known as “incremental files” or “difference files”—not to be confused with the file referred to in the `DIFFERENCE FILE` optional parameter available in `CREATE DATABASE` and `ALTER DATABASE` statements. It refers to *The “delta” file*.



Make sure you do not skip levels in your scheme. As described earlier, level *n* files reach back to a level *n-1* file.

Getting back to our example scheme, you want to use level 1 for weekly backups and level 2 for our daily ones. For the last Sunday of each month, you could schedule a level 0 (full) backup.

The syntax for the “difference” backups is the same as for the full backup, except for the `<level>` argument and, if you are using a custom file-naming scheme, the names you use for the files.

On the day after the full backup (level 0), you make the first *weekly* one with level 1, which will chain back to the level 0 backup. It will also provide a link for your daily backups to chain back to in the coming week. Assume we are running this from a directory called `/usr/var/backup` and the `/usr` directory is on a different disk to the one where the databases live. The database is aliased as ‘ourdb’:

```
/opt/firebird/bin/nbackup -U sysdba -P m1llp0nd -B 1 ourdb
```

The difference file, containing pages changed since Sunday’s full backup, will be named in the pattern `ourdb-1-yyyyymmdd-hhmm.nbk`.

If you want to start the daily backups now, do this:

```
/opt/firebird/bin/nbackup -U sysdba -P m1llp0nd -B 2 ourdb
```

and difference file `ourdb-2-yyyyymmdd-hhmm.nbk` will be stored, containing all of the pages changed since the still-current level 1 (weekly) backup.

Next day, you make another level 2 backup. The stored pages in that difference file will also point back to that still-current weekly backup (not to yesterday’s daily one).

And so it will go, for the rest of the week, until you do the next weekly backup. When you do, the new level 1 difference file will have all the changed pages since the still-current level 0 (full monthly) backup. All the subsequent level 2 (daily) files will have the changes since the newer level 1 backup.

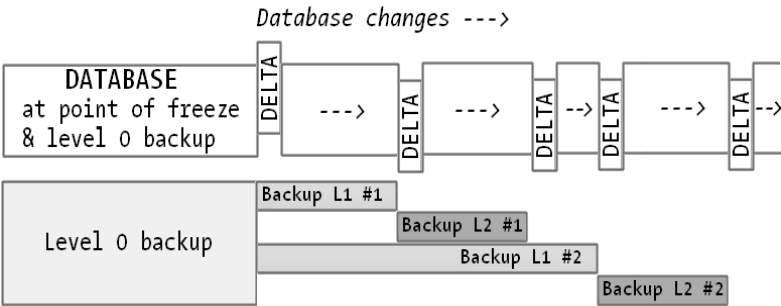
Backup “sets”

The concept of a “backup set” is yours to control. Nothing in the *nBackup* constellation formally identifies any concept such as a set, group or cycle. It simply makes the backup files it is instructed to make and connects the newest level *n* backup back to the newest level (*n-1*) file. For restoring, it uses unique identifiers in each file to establish the linkages—this is discussed in more detail in the upcoming topic [*Restoring from nBackup Files*](#).

Sample backup map

Figure 39.1 illustrates a sample “map” of a backup sequence. This example is just one way you might arrange things. You might want a level 3, for example, to capture differences hourly, or two-hourly. You might want the frequency of your level 0 backups to be more or less often than monthly. How you do it is over to you.

Figure 39.1 Backup sequence for *nBackup*



Restoring from *nBackup* Files

When *nBackup* reconstructs a database from its backup files, it starts with the latest level 0 database and works its way, in order, up the levels of the difference files that you specify in the restore command.

The switch for restoring incremental backups is `nbackup -R`. The syntax pattern is as follows:

```
nbackup [-U <user> -P <password>] -R <database> <file0> [<file1> [...] ]
```

The list of file name arguments could be quite lengthy, since the full chain of files that you want restored into a database must be enumerated individually, from the bottom up, in the exact order that will reconstitute the database as it stood at the point of the newest file in the list.



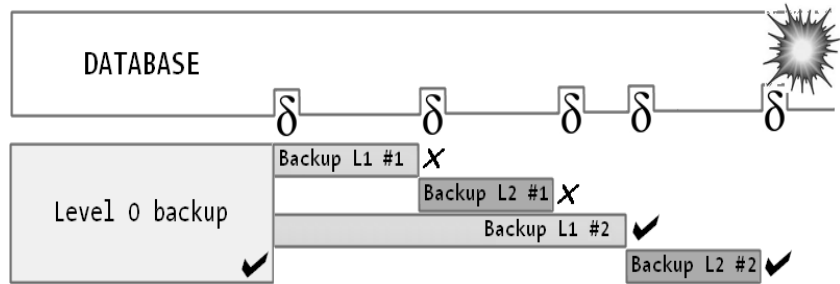
nBackup assumes that all arguments after the <database> argument are backup files. It is thus essential that all other switches and their arguments precede the -R switch and that you are careful to place the <database> argument before the first file name argument.

To simplify the restore, it is suggested that you

- create a short alias in `aliases.conf` for the database path that you will supply as the <database> argument to the `-R` switch; and
- run the restore from the directory where the files are located
- map out a diagram of the files you will need to achieve the point-in-time that you want to restore to

Sample backup map

Figure 39.2 illustrates a sample “map” of a restore sequence.

Figure 39.2 Restore sequence map for *nBackup*

The best case restore will consist of the Level 0 backup plus the #2 Level 1 and #2 Level 2 difference files. If it appears the damage occurred earlier, exclude the #2 L2 file. If the restore still produces a corrupt database, go back to a point-in-time restore, using the L0 backup, the #1 L1 and the #2 L2 files—and excluding the later difference files.

Bad restore chains

Providing a bad chain of file names—such as omitting a file or naming two files of the same level—does not result in a bad database. The **nBackup -B** process has stored a unique identifier in each file it created. In each difference file (level 1 or higher backup) it has also stored the identifier of the previous-level file whose differences it has collected.

If the restore process encounters a mismatch in the chain between a level (n-1) identifier and a level n identifier, it will not link them. It will cancel the process and deliver an error message.

Using the Freeze Utility

The “freeze” utility that comes with *nBackup* enables you to divert database changes temporarily to a “delta” file for the duration of a file-copying operation.

A freeze does not prevent users from reading or updating anything; but it prevents the database file from being written to as long as the freeze continues. The utility consists of a group of tools that are implemented as switches to the *nBackup* command.

The usual reason why you might want to freeze a database in this way is to allow the database file to be safely copied by another backup or copying utility whilst normal work continues in the database. Without this physical freeze, such utilities can corrupt an active database by placing arbitrary write locks on blocks of disk space over which the Firebird engine expects to have full access and control.



Do NOT be tempted to try using these tools on a multi-file database.

Starting a Freeze—nBackup -L

When you are about to start running your copy or backup utility over the partition where the database resides, you start a Freeze by issuing an *nBackup* command with the following syntax:

```
nbackup [-U <user> -P <password>] -L <database>
```

The `-L` switch stands for the verb “lock”. What it actually does do is to set a state flag on the database header page to “Locked” to signal the engine subsystems and other applications that writes are being redirected.



If you do a `gstat -h` on the database when this flag is set, you will see ‘LOCKED’ amongst the optional database attributes listed at the end of the output.

As with other `nBackup` switches, the user name and password of a user with escalated privileges will be required in some form. If the environment variables `ISC_USER` and `ISC_PASSWORD` are not suitably exposed and you are not using a system log-in that exempts you from Firebird native authentication.

The `<database>` argument can be an alias (defined in `aliases.conf`). If it is not an alias, it must be a properly qualified file path to the database.

When the `nBackup -L` command is complete, it will have created the “.delta” file, ready for receiving changed pages from the database.

Conditions are now right for running your backup or copying software safely. Users can continue to work in the database without any risk of that software interfering with their interaction with the Firebird database.

Ending a Freeze—`nBackup -N`

When the backup or copying software has finished its tasks, you need to “unfreeze” the database, using `nBackup` with its `-N` (uNlock) switch. The syntax pattern is similar to that for the Freeze option:

```
nbackup [-U <user> -P <password>] -N <database>
```

This “unfreeze” causes `nBackup` to merge the differences from the “delta” file into the main database file and then to delete the “delta” file. When it is done, it resets the status on the database header to “NORMAL”.

Resetting a Restored Database—`nBackup -F`

The backup copy made by your external software during a freeze will be a byte-by-byte copy of the database in its “frozen” state. When you retrieve the file from the backup and, if necessary, decompress it, the state flag on its database header page will still be “LOCKED” and clients will be unable to attach to it.

The tool for resetting the database state to “NORMAL” under these conditions is `nBackup`, using the `-F` (“fix-up”) switch. It is a file-level operation that does not even require Firebird to be running.

Fix what?

It is not really a “fix” tool at all, since nothing is broken! The intent of the tool is simply to reset that state flag on the header page of a locked database that doesn’t have a “delta” file, for whatever reason. Typically, that would be a database file that you retrieved from an external backup.

You could use the tool to “fix” a frozen but undamaged database whose “delta” was deleted before the “unfreeze” directive was issued. It’s a non-fix, insofar as it has no magic powers that can rescue data from a deleted “delta”.



Do NOT use the `-F` switch to unfreeze a database whose “delta” is still alive. Use `nBackup -N`.

Syntax

The syntax pattern is:

```
nbackup -F <database>
```

If -U or -P arguments are included, they are simply ignored.

After the command completes, the database status becomes “NORMAL” and clients will be able to attach as usual.

SQL Support for nBackup

The introduction and development of *nBackup* brought a little SQL cargo with it in the form of a statement syntax for ALTER DATABASE.

ALTER DATABASE

The ALTER DATABASE statement has three syntax patterns applicable to *nBackup*:

```
ALTER DATABASE {BEGIN | END} BACKUP
```

and

```
ALTER DATABASE ADD DIFFERENCE FILE <file-spec>
```

also

```
ALTER DATABASE DROP DIFFERENCE FILE
```

Note also:

```
CREATE DATABASE {...}
...
ADD DIFFERENCE FILE <file-spec>
...
```

BEGIN | END BACKUP

From Firebird 2.0, ALTER DATABASE BEGIN BACKUP could be called to set up the conditions for preparing a database for a “freeze”: setting a flag on the header page of the database indicating a “LOCKED” state and preparing a “delta” file to which changes to database pages are written during a freeze. That statement does not perform a backup.

The purpose for the “freeze” is to prevent writes to the database pages while a backup is being performed. It is also used as a tool in its own right, to manually freeze the database while an external copying, compression or backup program is doing arbitrary deeds in the file system that could put write locks on blocks of disk that Firebird expects to be under its control.

The companion statement ALTER DATABASE END BACKUP merges the “delta” file back into the database and resets the state flag to “NORMAL”.

Users of nBackup have no need to use these statements, since the the *nBackup* application executes them in the appropriate places, according to the switches passed to it.

ADD/DROP DIFFERENCE FILE

From Firebird 2.5, both ALTER DATABASE and CREATE DATABASE were enhanced to provide the option to customise the location and name of the “delta” file, the temporary difference file that is created by *nBackup* operations to store changes to database pages that occur during a “freeze”.



In previous versions, or if the DIFFERENCE FILE attribute is unused or dropped, the delta file is named using a hard algorithm as XXXXXXXX.delta (where XXXXXXXX would be substituted by the main part of the database file name). The file is written to the current working directory.

The logged-in user should be SYSDBA (or equivalent) or the database Owner. The syntax pattern is:

```
alter database ADD DIFFERENCE FILE '<path-and-filename>'
```

Notice that the `<path-and-filename>` argument is a string, so it must be single-quoted. The path must exist—there is no provision for the engine to create a directory here.

In a CREATE DATABASE statement, the DIFFERENCE FILE can be included as an optional parameter:

```
CREATE DATABASE ...  
...  
    DIFFERENCE FILE '<path-and-filename>'  
...
```

A database can have only one DIFFERENCE FILE target. If you want to abandon the DIFFERENCE FILE attribute and revert to the defaults, an ALTER DATABASE syntax is available for dropping it:

```
alter database DROP DIFFERENCE FILE
```

To change the custom location and/or name used, first drop the existing attribute, then add the new one.

nBackup command-line options summary

Table 39.4 presents a summary of the nBackup switches and their arguments.

Table 39.4 Command-line options for *nBackup*

Option switch	Used By	Effect
-B <level> <database> [<filename>]	Backup utility	Create full or difference (incremental) backup
-D [ON OFF]	Backup utility	Use or disabke direct I/O when scanning database
-F <database>	Freeze utility	Reset database state flag on file copy backup that was done while the database was in a locked (“frozen”) state.
-FE <file>	All	Fetch password from file
-L <database>	Freeze utility	Lock database to freeze before filesystem copy
-N <database>	Freeze utility	Unlock previously frozen database
-P <password>	All	Password
-R <database> [<file0> [<file1>...]]	Backup utility	Restore incremental backup
-S	Freeze utility	Print database size in pages after lock
-T	All	Suppress database triggers

Option switch	Used By	Effect
-U <user>	All	User name
-Z	All	Print program version

Database Shadowing

Firebird has a capability which can enable immediate recovery of a database in case of disk failure or accidental filesystem deletion of the database. Called *shadowing*, it is an internal process that maintains a real-time, physical copy of a database. Whenever changes are written to the database, the shadow receives the same changes simultaneously.

An active shadow always reflects the current state of the database. However, although shadowing has obvious benefits as a hedge against hardware failure, it is not an on-line replication system and it should never be used as a substitute for a backup system.

Benefits and limitations of shadowing

The main benefit of shadowing is that it provides a quick way to recover from a hardware disaster. Activating a shadow makes it available immediately. It runs invisibly to users as an extra loop in the data-writing process, with minimal attention from the DBA.

Creating a shadow does not require exclusive access to the database and the shadow can be maintained in one or many files, in controlled spaces of the server's storage system.

However, shadowing is not a protection against corruption. Data written to the shadow are an exact copy of what is written to the database, warts and all. If user errors, intermittent disk faults or software failures have corrupted data, then the same corrupt data will show up in the shadow.

Shadowing is an “all or nothing” recovery method. It has no provision to recover some pieces and ignore others nor to revert to a specific point in time. It must live in the same filesystem as the server: a shadow must be written to fixed hard drives local to the server. It cannot be written to shares, non-local filesystems or removable media.



On systems that support NFS, it is possible to maintain the shadow on an NFS filesystem. However, it is not recommended because the shadow will become detached—and thus, useless—if the network connection is lost.

Important caveats

1 Shadowing is not a substitute for backup

Do not be lulled into the belief that shadowing is in any way a substitute for regular backup and periodic restores.

- A shadow file is no less vulnerable to the “slings and arrows of outrageous fortune” than is any other file in your system.
- One lost or damaged shadow file makes the whole shadow useless.

- A dying disk or a flaky memory module is capable of great mischief before it brings your database down totally. Every piece of mischief will be faithfully written to the shadow.

2 A shadow cannot accept connections

Never try to connect to a shadow nor interfere with it using system or database tools. The server knows what it has to do to a shadow to convert it to an active database.



It is not a serious problem if a shadow is accidentally damaged or deleted. As long as you know the accident has occurred, a shadow for a healthy database can be regenerated at any time by simply dropping the shadow and creating it again.

Implementing shadowing

Firebird has DDL syntax for creating and dropping shadows with various optional clauses to specify location, operating mode and file sizes. Altering a shadow requires dropping the existing one and creating a new one with new specifications.

A shadow that is a single disk file is referred to as a shadow file. A multiple-file shadow—which can be distributed across more than one disk—is known as a shadow set. Shadow sets are grouped by a shadow set number.

Shadow file location and distribution

A shadow should be created in different fixed disk spaces from the location or locations of the active database files, since one of the main purposes of shadowing is to provide fallback in case of disk failure!

Disk space for shadowing must be physically attached to the Firebird server's host machine. Files in a shadow set can be distributed across multiple disks to improve file I/O and space allocation. Like file specifications for databases, those for shadows are platform-specific.

Shadow options

The shadowing subsystem comes with a variety of configuration options.

Mode (AUTO or MANUAL)

Mode settings—automatic (with or without the conditional attribute) or manual—determine what occurs if something happens to make the shadow unavailable.

AUTO mode is the default. It allows the database to keep running in the event that the shadow becomes inoperable.

- There will be a “window of time” during which no shadowing happens and the DBA may be unaware of the loss.
- If the lost shadow was created with the conditional attribute, Firebird creates a new shadow automatically, if it can.
- If shadowing is not conditional, it will be necessary to recreate a shadow manually.

MANUAL mode prevents further access to the database in the event that the shadow becomes unavailable. Choose it if continuous shadowing is more important than continuous operation of the database.

- To allow connections to resume, the administrator must remove the old shadow file, delete references to it, and create a new shadow.

Conditional shadowing

One of the ways a shadow could become unavailable is by taking over as the main database in the event of the hardware “death” of the existing database—that is, after all, the whole idea of shadowing!



*The **CONDITIONAL** attribute should also cause a new shadow to be created automatically if the existing shadow becomes the operational database. However, practice shows that it does not always happen. Be aware of the possibility that it won't work as expected and be prepared to subject it to an emergency drill!*

Single-file vs multi-file shadows

By default, a shadow is created as a “set” consisting of one file. However, a shadow set can comprise multiple files. As a database—and hence its shadow—grow in size, the shadow can be re-specified and regenerated with more files to accommodate the increased space requirements.

Creating a shadow

A shadow can be created without needing exclusive access or affecting any connected users. The DDL statement `CREATE SHADOW` creates a shadow for the database to which you are currently connected.

Syntax

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL] 'filespec'
    [LENGTH [=] int [PAGE[S]]] [<secondary_file>];
```

```
<secondary_file> = FILE 'filespec' [<fileinfo>][<secondary_file>]
<fileinfo> = {LENGTH[=]int [PAGE[S]] | STARTING [AT [PAGE]] int } [<fileinfo>]
```

As with `CREATE DATABASE`, the file specs for shadow files are always specific to platform.

Single-file shadow

Let's assume we are on a Linux server, logged in to the database `employee.gdb`, which is located in the `examples/empbuild` directory beneath your Firebird root. We have decided to shadow your database to a partition named `/shadows`. To create a single-file shadow for it there, we use the statement

```
CREATE SHADOW 1 '/shadows/employee.shd';
```

The shadow set number can be any integer. A `page_size` is not included, since the page size is taken from the specification of the database itself.

Use the `isql` command `SHOW DATABASE` to check that the shadow now exists:

```
SQL> SHOW DATABASE;
Database: /usr/local/firebird/examples/empbuild/employee.gdb
Shadow 1: '/shadows/employee.shd' auto
PAGE_SIZE 4096
Number of DB pages allocated = 392
Sweep interval = 20000
...
```

Multi-file shadow

The syntax for creating a multi-file shadow is quite similar to that for creating a multi-file database: the secondary shadow file specifications are “chained” to the primary file specification, with file specs and size limits for each file.

In this example, assume we are logging into employee.gdb in its default Win32 location. We are going to create a three-file shadow on drives F, H and J, which are partitions on hard-drives in the server's filesystem.



File sizes allocated for secondary shadow files need not correspond to those of the database's secondary files.

The primary file, employee1.shd, is 10,000 database pages in length and the first secondary file, employee2.shd is 20,000 database pages long. As with the database, the shadow's final secondary file, employee3.shd will grow as needed until space is exhausted on partition J or the filesystem limit is reached.

```
CREATE SHADOW 25 'F:\shadows\employee1.shd' LENGTH 10000
FILE 'H:\shadows\employee2.shd' LENGTH 20000
FILE 'J:\shadows\employee3.shd';
```

Alternatively, we can specify starting pages for the secondary files instead of absolute lengths for the primary and non-final secondary files:

```
CREATE SHADOW 25 'F:\shadows\employee1.shd'
FILE 'H:\shadows\employee2.shd' STARTING AT 10001
FILE 'J:\shadows\employee3.shd' STARTING AT 30001;
```

Verifying in isql:

```
SQL> SHOW DATABASE;
Database: C:\Progra~1\firebird\examples\employee.gdb
Owner: SYSDBA
Shadow 25: 'F:\SHADOWS\EMPLOYEE1.SHD' auto length 10000
file H:\SHADOWS\EMPLOYEE2.SHD starting 10001
file J:\SHADOWS\EMPLOYEE3.SHD starting 30001
PAGE_SIZE 1024
Number of DB pages allocated = 462
Sweep interval = 20000
...
```

Manual mode

The examples above created shadows in the default auto mode. Now, suppose we have a requirement that work in the database must be stopped at all times when shadowing is disabled for any reason. In this case, we want to stipulate that the shadow is created in manual mode (refer to notes earlier in this topic). In order to tell the Firebird engine that we want this rule enforced, we create the shadow using the keyword **MANUAL** in the **CREATE SHADOW** statement:

```
CREATE SHADOW 9 MANUAL '/shadows/employee.shd';
```

Now, if the shadow kicks in because we lose the main database file, or if the shadow becomes unavailable for some other reason, the administrator must drop the old shadow file and create a new shadow before clients can resume connecting.

A conditional shadow

In both examples above, the `CREATE SHADOW` specifications leave the database unshadowed after the shadow becomes unavailable—whether by becoming detached from the database or by its having kicked in as the active database upon the physical death of the original database.

In the case of a `MANUAL` mode shadow, that is the situation we want. We choose this mode because we want database connections to stay blocked until we manually get a new shadow going.

In the case of an `AUTO` mode shadow, database connections can resume after the death, as soon as the shadow kicks in. No shadowing will occur from this point until the admin manually creates a new shadow. Also, if a shadow becomes detached for some reason, the admin will not know about it. Either way, we have a window where shadowing has lapsed.

As described earlier in this topic, we can improve this situation by including the `CONDITIONAL` keyword in the `AUTO` mode shadow's definition. The effect will be that, when the old shadow becomes unavailable, the Firebird engine will do the necessary housekeeping and create a new shadow automatically.

For example:

```
CREATE SHADOW 33 CONDITIONAL '/shadows/employee.shd';
```

Increasing the size of a shadow

At some point, it may become necessary to increase the sizes or numbers of files in a shadow. Simply drop the shadow (see next) and create a new one.

Dropping a shadow

A shadow needs to be dropped in the following situations:

- It is a manual shadow that has been detached from the system for some reason. Dropping the compromised shadow is a necessary precursor to creating a new shadow and resuming database service.
- It is an unconditional automatic shadow that has been offline because of some system event. It needs to be regenerated to restore its integrity.
- You need to resize the files in a shadow, add more files or set up a new shadow with different attributes.
- Shadowing is no longer required. Dropping the shadow is the means to disable shadowing.

Dropping a shadow removes not just the physical files but also the references to it in the metadata of the database. To be eligible to run the command, you must be logged in to the database as the user who created the shadow, the `SYSDBA`, or a user with superuser privileges.

Syntax `DROP SHADOW shadow_set_number;`

The shadow set number is a compulsory argument to the `DROP SHADOW` command. To discover the number, use the `isql/SHOW DATABASE` command while logged in to the database.

The following example drops all of the files associated with the shadow set number 25:

```
DROP SHADOW 25;
```

Using *gfx*–kill to drop a shadow

The command-line *gfx* housekeeping utility tool (see Chapter 35, *Configuring and Managing Databases*) has a `-kill` switch that conveniently submits a `DROP SHADOW` command internally to clean up an unavailable shadow. After executing this command, it will be possible to proceed with creating a new shadow.

For example, to drop the unavailable shadow on our employee database on POSIX:

```
[root@coolduck bin]# ./gfx -kill ../examples/empbuild/employee.fdb
-user SYSDBA -password masterkey
```

On Win32:

```
C:\Program Files\Firebird\..\bin> gfx -kill ..\examples\empbuild\employee.fdb
-user SYSDBA -password masterkey
```

Replication

Another way to provide disaster fallback for your databases is replication. Properly implemented replication engines do not copy data: they recreate them by reading a list or log of changed records in a source database and writing the changes to one or a number of target databases located on other servers, using multi-database transactions.

A well-featured replication module will provide many ways to keep these target servers in “synch” and in close to real time. The principal objective of replication is to facilitate the sharing of the same data, or subsets of them, by servers that are local to their respective remote sites, often involving a centralised “head office” database in the topology.

“Warm Backups”

However, replication can provide fast disaster fallback for 24/7 systems by being set up to maintain “warm backups” on separate servers that are geographically local.

Another benefit of using an external replication module for backup is that, being a client application, it need not run on the same host machine as the Firebird server and databases.

Firebird is not distributed with a replication module but several third-party modules are available, including one or two non-commercial or shareware ones with uses in low-end deployments.



*The most advanced replication software in the field for Firebird at the time of this writing is the IBPhoenix commercial product, **IB Replication Suite**. It is multi-platform, well-supported and is constantly being enhanced in response to requests from its user community. It is well worth a look if frequent use of other backup methods tends to cause problems for site users.*

THE SERVICES MANAGER

Firebird has a subsystem known as the *Services Manager*. It provides an application programming interface that enables client programs to perform a number of administration tasks—“services”—such as backup, user management and troubleshooting. Most Firebird drivers and host language components provide wrappers for the Services API calls.

About the Services Manager

Through the Services API, programs submit requests to the Services Manager, which then accesses components of the Firebird engine on behalf of the calling client.

Firebird inherited the initial Services Manager for Superserver with the open sourced InterBase 6 code. Over the years, the implementation matured so that, by the “2” series, all the functions were available for the Classic server model and, latterly, for Superclassic. The Services API has grown, too, and continues to do so.

Accessing the Services Manager API

The Services API is “out there” for developers to use in applications. In general, Services API functions do not have an SQL equivalent.¹ The API defines *service* functions that are furnished with *service parameter blocks* (SPBs), the vehicle for passing parameters through the interface.

1. The exceptions are the functions `add_user`, `modify_user` and `delete_usr`. From v.2.5, these tasks can be executed in DSQL from any database except `security.fdb` using `CREATE/ALTER/DROP USER` statements. More information, see [Using DDL to Manage User Accounts](#) in Chapter 13, *Data Definition Language—DDL*.

Services Manager Clients

The Firebird command-line tools *gfix*, *gbak* and *gsec* are embedded (ESQL) client applications that use the Services Manager. When used locally on server models that support a direct, local connection,, they all invoke the Services Manager automatically. Remote clients, and also any local clients that use the TCP/IP local loopback server, localhost, connect explicitly to the Services Manager subsystem, identified by the symbol **service_mgr**. In the case of *gbak*, the remote client must specify it in its connection parameters, via the **-se[rvic]** switch, and then provide the local path or the stand-alone alias to designate the database. For example,

```
gbak -b -se myhost:service_mgr d:\data\stocks.fdb f:\backups\stocks\stocks.20140715.fbk
gbak -b -se myhost:service_mgr d:mystocks f:\backups\stocks\stocks.20140715.fbk
```

The other command-line tools will connect automatically to the Services Manager, given the full network path to the database (or to the security database, in the case of *gsec*).

Many popular graphical tools for Firebird provide interactive interfaces to the services managed by the Services Manager, including the free, open source Flamerobin².

The Firebird driver for Java, Jaybird, implements the Services API, as do the Firebird .NET providers.

The old Inprise company made a beta implementation of the Services API available for Delphi (ObjectPascal) with the open source InterBase 6 beta, as part of its InterBaseXpress (IBX) connectivity component set. Although IBX is no longer suitable for use with Firebird, several commercial and open source versions of those “service components” have been developed over the intervening years: IBOjects, FIBPlus and the cross-platform, open source Free Pascal for the Lazarus development environment, to name just a few.

From Firebird 2.1 onward, Firebird comes with a “bare bones” command-line interface to the Services API.

The *fbsvcmgr* Utility

v.2.1 +

The command-line utility *fbsvcmgr* provides a command-line interface to the Services API, enabling access to any service that is implemented in Firebird.³

This tool does not claim to compete with the many satisfactory graphical tool-sets available that make the Services API functions available in user-friendly interfaces. It addresses the problem for skilled administrators needing to access remote Unix servers in broad networks through a text-only connection. Previously, meeting such a requirement needed a programmer.

Using *fbsvcmgr*

Unlike the traditional “g*” utilities, *fbsvcmgr* does not provide “switches” that encapsulate the parameters available to functions. Rather, it is just a front end through which the

2. FlameRobin’s connectivity layer is the free, open source IBPP interface for C++. IBPP provides full wrappers for the Services API.

3. The incorporation of documentation supplied by Alexander Peshkov and the Firebird Project team is acknowledged.

Services API functions and parameters can pass. Users therefore need to be familiar with the Services API as it stands currently.

ibase.h

The API header file—`ibase.h`, in the `../include` directory of your Firebird installation— should be regarded as the primary source of information about what is available, backed up by the InterBase 6.0 beta API Guide and the “API and ODS” chapters of all release notes from Firebird 1.5 onwards. Download links for all of those documents can be found at <http://www.firebirdsql.org/en/reference-manuals/> (scroll down) and <http://www.firebirdsql.org/en/release-notes/>, respectively.



The format described for some parameters in the InterBase 6 beta documentation is buggy. When in doubt, treat `ibase.h` as the primary source for the correct form.

Also, everything to do with licensing for InterBase was removed from the original open source code and is therefore not supported either in Firebird or by `fbsvcmgr`. The old Config file view/modification functions have been unsupported since `firebird.conf` replaced them in v.1.5 and are not implemented by `fbsvcmgr`.

SPB Syntax

The SPB syntax that `fbsvcmgr` understands closely matches with what you would encounter in the `../include/ibase.h` file or the InterBase 6.0 API documentation, except that a slightly abbreviated form is used to reduce typing and shorten the command lines a little.

Here's how it works.

All SPB parameters have one of two forms:

- 1 `isc_spb_VALUE`
- 2 `isc_VALUE1_svc_VALUE2`.

For `fbsvcmgr` you just need to pick out the `VALUE`, `VALUE1` or `VALUE2` part[s] when you supply your parameter.

Accordingly, for (1) you would type simply `VALUE`, while for (2) you would type `VALUE1_VALUE2`. For example:

```
isc_spb_dbname -you type- dbname
isc_action_svc_backup -you type- action_backup
isc_spb_sec_username -you type- sec_username
isc_info_svc_get_env_lock -you type- info_get_env_lock
```

and so on.



*An exception is `isc_spb_user_name`: it can be specified as either **user_name** or simply **user**.*

Parameters

It is not realistic to attempt to describe all of the SPB parameters here. In the InterBase 6.0 beta documentation it takes about 40 pages! This section describes how to specify the API parameters using this tool.

Specify the Services Manager

The first required parameter for a command line call is the Services Manager you want to connect to:

- For a local connection use the simple symbol `service_mgr`

- To attach to a remote host, use the format `hostname:service_mgr`

Specify subsequent service parameter blocks (SPBs)

Subsequent SPBs, with values if required, follow. Any SPB can be optionally prefixed with a single `'` symbol. For the long command lines that are typical for *fbsvcmgr*, use of the `'` improves the readability of the command line. Compare, for example, the following (each a single command line despite the line breaks printed here):

```
# fbsvcmgr service_mgr user sysdba password masterke action_db_stats dbname employee
sts_hdr_pages

and

# fbsvcmgr service_mgr -user sysdba -password masterke -action_db_stats -dbname employee
-sts_hdr_pages
```

fbsvcmgr specifics

The next section highlights some known differences between the operation of *fbsvcmgr* and what you might otherwise infer from the old beta *API Guide*.

“Do's and Don'ts”

With *fbsvcmgr* you can perform a single action—and get its results if applicable—or you can use it to retrieve multiple information items from the Services Manager. You cannot do both in a single command.

For example,

```
# fbsvcmgr service_mgr -user sysdba -password masterke -action_display_user
```

will list all current users on the local Firebird server:

```
SYSDBA Sql Server Administrator 0 0
QA_USER1                        0 0
QA_USER2                        0 0
QA_USER3                        0 0
QA_USER4                        0 0
QA_USER5                        0 0
GUEST                           0 0
SHUT1                           0 0
SHUT2                           0 0
QATEST                          0 0
```

...and...

```
# fbsvcmgr service_mgr -user sysdba -password masterke -info_server_version
-info_implementation
```

will report both the server version and its implementation:

```
Server version: LI-T2.1.0.15740 Firebird 2.1 Alpha 1
Server implementation: Firebird/linux AMD64
```

But an attempt to mix all of this in single command line:

```
# fbsvcmgr service_mgr -user sysdba -password masterke
-action_display_user -info_server_version -info_implementation
```

raises an error:

```
Unknown switch “-info_server_version”
```


Services API Functions

The following table is a selection of the tasks that you can request.

Table 40.1 Services API tasks

Task	Function or Parameter	Description
Initiate a session with Service Manager	service_attach	Connects to the Service Manager of a specified server, returning a service handle required by any Service Manager functions.
Perform a service task	service_start	Performs a service task on the server to which the requestor has connected using service_attach with the appropriate parameters.
Get server information	service_query	Requests and retrieves information about the Firebird server from which the request is sent
Close session with Service Manager	service_detach	Terminates the connection to the Services Manager
Set various properties on the database header page	action_svc_properties	Invokes various database configuration options (like gfix with various switches)
Back up a database	action_svc_backup	Invokes the same code as gbak -b
Restore a database	action_svc_restore	Invokes the restore-from-backup code, same as gbak -r/gbak -rep
Read the database log	action_svc_get_fb_log	Generates a view of the firebird.log file
Get statistics reports for database	action_svc_db_stats	Produces the same reports as gstat
Analyse and repair	action_svc_repair	Invokes validation and repair operations, like gfix with specific options
Get info about one user or a list of all users from the security database	action_svc_display_users	Outputs same as gsec -display
Add a new user	action_svc_add_user	Adds a new user to security2.fdb. Invokes same code as gsec -add
Modify a user	action_svc_modify_user	Allows some changes to a user's record in security2.fdb. Invokes same code as gsec -modify
Delete a user	action_svc_delete_user	Deletes a user from security2.fdb. Invokes same code as gsec -delete
Force Windows trusted authentication	spb_trusted_auth ¹	Forces Firebird to use Windows trusted authentication. Not applicable to other platforms

Task	Function or Parameter	Description
Set a database name when accessing security service remotely	spb_dbname ²	Equivalent to supplying the -database switch to the gsec utility
List limbo transactions	spb_rpr_list_limbo_trans	Equivalent to gfix -l

- 1. From v.2.1 +
- 2. From v.2.1 +

SECOND EDITION

PART IX



Appendices

APPENDIX



INTERNAL AND EXTERNAL FUNCTIONS

In this Appendix are summarised descriptions of the internal and external functions available in Firebird.

At the end of this Appendix, for your convenience, we provide the guidelines for *Building regular expressions* extracted from the **Firebird 2.5 Language Reference Update**, published by the Firebird Project and usually included in the binary download kits. Regular expressions are used with the SIMILAR TO comparison predicate which is supported in Firebird 2.5+ versions.

Internal Functions

At version 2.1, a large number of functions that had previously been implemented as external (UDFs) were coded internally and, in many cases, made compliant with the ISO standards with respect to input arguments. Some new functions were added also.

Conditional Logic Functions

COALESCE COALESCE (<value1>, <value2> [, <valueN> ...]) -- return type depends on input

Takes a comma-separated list of two or more arguments (columns, expressions, constants, etc.) that resolve to values of the same type and returns the value of the first argument that evaluates to a non-null value. If all the input arguments resolve as NULL, the result is NULL.

Example This example tries to create an identifying description for a vehicle by running through a list of descriptive fields until it finds one that has a value in it. If all fields are NULL, it returns 'Unknown'.

```
select
  coalesce (Make, Model, Series, 'Unknown') as Listing
from Catalog;
```

Comment *Available from v.1.5+. In Firebird 1.0.x, to accomplish the same effect, refer to the external functions INVL and sNVL.*

DECODE DECODE(<expression>, <search>, <result> [, <search>, <result> ...] [, <default>]
DECODE is a shortcut for a CASE ... WHEN ...ELSE expression.

Example select
 decode(state, 0, 'deleted', 1, 'active', 'unknown') as retval
from things;

IIF IIF (<condition>, <value_if_true>, <value_if_false>)
Returns <value_if_true> if condition is met, otherwise returns <value_if_false>. Arguments <value_if_true> and <value_if_false> can be columns, constants, variables or expressions and must be of compatible types. The <condition> argument must be a valid predication that returns a logical result.

Example The following statement returns PACKAGE_SIZE as 'LARGE' if the value in the Qty field is 100 or greater; otherwise, it returns 'SMALL'.
 SELECT
 PRODUCT,
 IIF ((Qty >= 100), 'LARGE', 'SMALL') AS PACKAGE_SIZE
FROM INVENTORY;

Comment *Available from Firebird 1.5 onward. Its usage is somewhat limited. From v.2.0 on, it is recommended to use a CASE expression instead.*

NULLIF NULLIF (value1, value2)
Returns NULL if value1 and value2 resolve to a match; if there is no match, then value1 is returned. Arguments value1 and value2 can be columns, constants, variables or expressions.

Example This statement will cause the STOCK value in the PRODUCTS table to be set to NULL on all rows where it is currently stored as 0:
 UPDATE PRODUCTS
 SET STOCK = NULLIF(STOCK, 0)

Comment *Available from Firebird 1.5 onward.*

Date and Time Functions

DATEADD DATEADD(<number> <timestamp_part> TO <date_time>)
 DATEADD(<timestamp_part>, <number>, <date_time>)

 timestamp_part ::= { YEAR | MONTH | | WEEK | DAY | HOUR | MINUTE | SECOND | MILLISECOND }
Returns a date, time or timestamp value incremented (or decrementsed, when negative) by the specified amount of time.

Examples select dateadd(day, -1, current_date) as yesterday from rdb\$database;

or (expanded syntax)
select dateadd(-1 day to current_date) as yesterday from rdb\$database;

- Comments
- 1. WEEK was added at v.2.5.
 - 2. YEAR, MONTH, WEEK and DAY cannot be used with time values.
 - 3. In versions prior to v.2.5, HOUR, MINUTE, SECOND and MILLISECOND cannot be used with date values.
 - 4. All timestamp_part values can be used with timestamp values.

DATEDIFF DATEDIFF(<timestamp_part> FROM <date_time> TO <date_time>)
DATEDIFF(<timestamp_part>, <date_time>, <date_time>)
timestamp_part ::= { YEAR | MONTH | WEEK | DAY | HOUR | MINUTE | SECOND | MILLISECOND }
Returns an exact numeric value representing the interval of time from the first date, time or timestamp value to the second one.

Example
select
datediff(DAY, (cast('TOMORROW' as date) -10), current_date)
as datediffresult from rdb\$database;

- Comments
- 1. WEEK was added at v.2.5.
 - 2. Returns a positive value if the second value is greater than the first one, negative when the first one is greater, or zero when they are equal.
 - 3. Comparison of date with time values is invalid.
 - 4. YEAR, MONTH,, WEEK and DAY cannot be used with time values. In versions prior to v.2.5, HOUR, MINUTE, SECOND and MILLISECOND cannot be used with date values.
 - 5. All timestamp_part values can be used with timestamp values.

EXTRACT EXTRACT (<time_part> FROM <datetime>) -- returns a SMALLINT or DECIMAL(6,4)
Extracts and returns an element from a DATE, TIME or TIMESTAMP expression as a numerical value. The return value will be a SMALLINT or DECIMAL(6,4), depending on which <time_part> is requested.
The <datetime> argument is a column or expression that resolves to a DATE, TIME or TIMESTAMP. Valid values for the <time_part> argument are:

<time_part>	Description	Type and Range
YEAR	The year represented by a DATE or TIMESTAMP input. Not valid for a TIME input.	SMALLINT in the range 1-9999
MONTH	The month represented by a DATE or TIMESTAMP input. Not valid for a TIME input.	SMALLINT in the range 1-12
DAY	The day of the month represented by a DATE or TIMESTAMP input. Not valid for a TIME input.	SMALLINT in the range 1-31

<time_part>	Description	Type and Range
WEEKDAY	The day of the week represented by a DATE or TIMESTAMP input. Not valid for a TIME input.	SMALLINT in the range 0-6, where 0 is Sunday.
YEARDAY	The day of the year represented by a DATE or TIMESTAMP input. Not valid for a TIME input.	SMALLINT in the range 0-365, where 0 is January 1.
HOURL	The hour of the day represented by a TIME or TIMESTAMP input. Returns zero if the input is a DATE type.	SMALLINT in the range 0-23
MINUTE	The minutes part of a TIME or TIMESTAMP input. Returns zero if the input is a DATE type.	SMALLINT in the range 0-59
SECOND	The seconds and sub-seconds parts of a TIME or TIMESTAMP input. Returns zero if the input is a DATE type.	DECIMAL(6,4) in the range 0.0000–59.999

Example In this trigger fragment, a document code based on the current year and a generator value is created from elements of a `TIMESTAMP` value:

```
...
if (new.DocCode is NULL) then
begin
  varDocCode = CAST (EXTRACT(YEAR FROM CURRENT_DATE AS CHAR(4)));
  varDocCode = varDocCode || '-' || CAST (GEN_ID (GenDocID,1) as VARCHAR (18));
  new.DocCode = varDocCode;
end
...
```

Comment Available in all versions.

String and Character Functions

ASCII_CHAR ASCII_CHAR(<number>)

Returns the ASCII character with the specified decimal code. The argument to `ASCII_CHAR` must be in the range 0 to 255. The result is returned in character set `NONE`.

Example select ascii_char(x) from y;

ASCII_VAL ASCII_VAL(<string>)

Returns the decimal ASCII code of the first character of the specified string.

- Returns 0 if the string is empty
- Throws an error if the first character is multi-byte
- The argument may be a text BLOB of 32,767 bytes or less

Example `select ascii_val(x) from y;`

BIT_LENGTH `BIT_LENGTH(<string> | <string_expr>)`

Returns the length of a string in bits.

Example `select
 rdb$relation_name,
 bit_length(rdb$relation_name),
 bit_length(trim(rdb$relation_name))
from rdb$relations;`

CHAR_LENGTH interchangeable with **CHARACTER_LENGTH**

`CHAR_LENGTH(<string> | <string_expr>)`

Returns the number of characters in a string or expression result.

Examples `select
 rdb$relation_name,
 character_length(rdb$relation_name),
 char_length(trim(rdb$relation_name))
from rdb$relations;`

Comment *See also OCTET_LENGTH*

GEN_UUID `GEN_UUID()` -- no arguments

Returns a universal unique number.

Example `insert into records (id)
 value (gen_uuid());`

HASH `HASH(<string>)`

Returns a HASH of a string.

Example `select hash(x) from y;`

LEFT `LEFT(<string>, <number>)`

Returns the substring of a specified length that appears at the start of a left-to-right string.

Example `select left(name, char_length(name) - 10)`

from people
where name like '% OLIVEIRA';

Comments

1. The first position in a string is 1, not 0.

2. If the <number> argument evaluates to a non-integer, banker's rounding is applied.

See also RIGHT.

LOWER LOWER(<string>)

Returns the input argument converted to all lower-case characters.

Example isql -q -ch dos850

SQL> create database 'test.fdb';

SQL> create table t (c char(1) character set dos850);

SQL> insert into t values ('A');

SQL> insert into t values ('E');

SQL> insert into t values ('Á');

SQL> insert into t values ('É');

SQL> select c, lower(c) from t;

C LOWER

=====

A a

E e

Á á

É é

Comments Available in v.2.0.x

The misrepresentation of the accute accent on the upper case 'A' in our example is a shortcoming of the rendering of this font.

LPAD LPAD(<string1>, <n> [, <string2>])

A “left-padding” function—prepends <string2> to the beginning of <string1> until the length of the result string becomes equal to <n>.

Example select lpad(x, 10) from y;

Comment See also RPAD

OCTET_LENGTH OCTET_LENGTH(<string> | <string_expr>)

Returns the length of a string or expression result in bytes.

Examples select

rdb\$relation_name,

octet_length(rdb\$relation_name),

octet_length(trim(rdb\$relation_name))

from rdb\$relations;

Comment See also CHARACTER_LENGTH | CHAR_LENGTH

OVERLAY OVERLAY(<string1> PLACING <string2> FROM <startpos> [FOR <length>])

Returns <string1> modified so that the substring that occupied the <length> positions FROM <startpos> are replaced by <string2>.

If <length> is not specified, CHAR_LENGTH(<string2>) is implied.

Example SELECT
 OVERLAY('Today is Tuesday'
 PLACING ' not ' FROM 9 FOR 1
 AS DENIAL
FROM RDB\$DATABASE
-- returns 'Today is not Tuesday'

Comments 1. The first position in a string is 1, not 0.
 2. If the <startpos> and/or <length> argument evaluates to a non-integer, banker's rounding is applied.
 3. The function will fail with text BLOBs of a multi-byte character set if the length of the result would be greater than 1024 bytes.

PI PI() -- no arguments

Returns the Pi constant (3.1459...).

Example val = PI();

POSITION POSITION(<string2> IN <string1>)
 POSITION(<string2>, <string1> [, <offset-position>])

Returns the start position of the <string2> inside <string1>, relative to the beginning of <string1>. In the second form, an offset position may be supplied so that the function will ignore any matches occurring before the offset position and return the first match following that.

Examples select rdb\$relation_name from rdb\$relations
 where position('RDB\$' IN rdb\$relation_name) = 1;
 /* */
 position ('be', 'To be or not to be', 10)

returns 17. The first occurrence of 'be' occurs within the offset and is ignored.

 position ('be', 'To buy or not to buy', 10)

returns 0 because the searched substring was not found.

REPLACE REPLACE(<stringtosearch>, <findstring>, <replstring>)

Replaces all occurrences of <findstring> in <stringtosearch> with <replstring>.

Example select replace(x, ' ', ',') from y;

REVERSE REVERSE(<value>)
Returns a string in reverse order—useful for creating an expression index that indexes strings from right to left.

Examples create index people_email on people
 computed by (reverse(email));
 select * from people
 where reverse(email) starting with reverse('.br');

RIGHT RIGHT(<string>, <number>)
Returns the substring, of the specified length, from the right-hand end of a string.

Example select
 right(rdb\$relation_name,
 char_length(rdb\$relation_name) - 4)
 from rdb\$relations
 where rdb\$relation_name like 'RDB\$%';

Comment See also LEFT

RPAD RPAD(<string1>, <length> [, <string2>])
“Right-padding” function for strings—appends <string2> to the end of <string1> until the length of the result string becomes equal to <length>.

Example select rpad(x, 10) from y;

Comments 1. If <string2> is omitted the default value is one space.
 2. If the result string would exceed the length, the final application of <string2> is truncated.

 See also LPAD

SUBSTRING SUBSTRING(<str> FROM startpos [FOR <n>]) -- returns a CHAR(n) or a text BLOB
Extracts a substring from a string <str>, starting at the position indicated by startpos and ending at the end of <str>. If the optional FOR <n> argument is supplied, the extracted substring will end when its length is equal to <n> if the end of <str> has not been reached already.

The arguments startpos and <n> (length for the substring) must resolve to integer types, while <str> can be any expression that evaluates to a string of 32,767 bytes or less or a BLOB not exceeding that byte-length limit. The first byte in the input string is at position 1.

Up to and including v.2.0, the return value is a string, regardless of whether the input <str> is a string or a BLOB. From v.2.1 forward, a text BLOB is returned if the input is a text or binary BLOB.

Important Prior to v.2.1, a SUBSTRING of a text BLOB does not work if the BLOB is stored in a multi-byte character set.

Examples `insert into Abbreviations(PID, Abbreviation)`
 `select`
 `PID,`
 `substring(Surname from 1 for 3)`
 `from PERSON;`

Comment *Available in versions 1.5+.*

TRIM `TRIM <left paren> [[<trim specification>] [<trim character>]`
 `FROM] <value expression> <right paren>`
 `<trim specification> ::= LEADING | TRAILING | BOTH`
 `<trim character> ::= <value expression>`

Trims characters (default: blanks) from the left and/or right of a string.

- Rules
- 1 If <trim specification> is not specified, BOTH is assumed.
 - 2 If <trim character> is not specified, ' ' is assumed.
 - 3 If <trim specification> and/or <trim character> is specified, FROM should be specified.
 - 4 .If <trim specification> and <trim character> is not specified, FROM should not be specified.
 - 5 If a text BLOB substring is specified as <value expression>, the value returned must not exceed 32,767 bytes.

Examples `select`
 `rdb$relation_name,`
 `trim(leading 'RDB$' from rdb$relation_name)`
 `from rdb$relations`
 `where rdb$relation_name starting with 'RDB$';`
 `/* */`
 `select`
 `trim(rdb$relation_name) || ' is a system table' from rdb$relations`
 `where rdb$system_flag = 1;`

Comment *Available in v.2.0.x*

BLOB Functions

Virtually all internal functions that are available to character types are also available to BLOBS of sub_type 1 (TEXT). Care should be taken always to ensure that string inputs and outputs never exceed the maximum length limit of the VARCHAR type, which is 32,765 bytes.

Mathematical Functions

<i>ABS</i>	ABS(<number>) Returns the absolute value of a number.
Example	<code>select abs(amount) from transactions;</code>
<i>CEIL CEILING</i>	{ CEIL CEILING }(<number>) Returns a value representing the smallest integer that is greater than or equal to the input argument.
Examples	1) <code>select ceil(val) from x;</code> 2) <code>select ceil(2.1), ceil(-2.1) from rdb\$database;-- returns 3, -2</code>
Comment	See also <i>FLOOR</i>
<i>EXP</i>	EXP(<number>) Returns the exponential <i>e</i> to the argument.
Example	<code>select exp(x) from y;</code>
<i>FLOOR</i>	FLOOR(<number>) Returns a value representing the largest integer that is less than or equal to the input argument.
Examples	1) <code>select floor(val) from x;</code> 2) <code>select floor(2.1), floor(-2.1) from rdb\$database; -- returns 2, -3</code>
Comment	See also <i>CEILING</i>
<i>LN</i>	LN(<number>) Returns the natural logarithm of a number.
Example	<code>select ln(x) from y;</code>
<i>LOG</i>	LOG(<number>, <number>) LOG (x, y) returns the logarithm base x of y.
Example	<code>select log(x, 10) from y;</code>

LOG10 LOG10(<number>)

Returns the logarithm base ten of a number.

Example select log10(x) from y;

MAXVALUE MAXVALUE(<value> [, <value> ...])

Returns the maximum value of a list of values.

Example select maxvalue(v1, v2, 10) from x;

MINVALUE MINVALUE(<value> [, <value> ...])

Returns the minimum value of a list of values.

Example select minvalue(v1, v2, 10) from x;

MOD MOD(<number>, <number>)

Modulo: MOD(X, Y) returns the ‘remainder’ part of the division of X by Y.

Example select mod(x, 10) from y;

POWER POWER(<number>, <number>)

POWER(X, Y) returns X to the power of Y.

Example select power(x, 10) from y;

RAND RAND() -- no argument

Returns a random number between 0 and 1.

Examples select * from x order by rand();

ROUND ROUND(<number>, [<number>])

Returns a number rounded to the specified scale.

Example select round(salary * 1.1, 0) from people;

Comment *If the scale (second parameter) is negative or is omitted, the integer part of the value is rounded. E.g., ROUND(123.456, -1) returns 120.000.*

SIGN SIGN(<number>)

Returns 1, 0, or -1 depending on whether the input value is positive, zero or negative, respectively.

Example select sign(x) from y;

SQRT `SQRT(<number>)`

Returns the square root of a number.

Example `select sqrt(x) from y;`

TRUNC `TRUNC(<number> [, <number>])`

Returns the integral part (up to the specified scale) of a number.

Examples 1) `select trunc(x) from y;`
 2) `select trunc(-2.8), trunc(2.8)`
 `from rdb$database;-- returns -2, 2`
 3) `select trunc(987.65, 1), trunc(987.65, -1)`
 `from rdb$database;-- returns 987.60, 980.00`

Trigonometrical Functions

ACOS `ACOS(<number>)`

Returns the arc cosine of a number. Argument to ACOS must be in the range -1 to 1.
 Returns a value in the range 0 to Pi.

Example `select acos(x) from y;`

ASIN `ASIN(<number>)`

Returns the arc sine of a number. The argument to ASIN must be in the range -1 to 1. It returns a result in the range -Pi/2 to Pi/2.

Example `select asin(x) from y;`

ATAN `ATAN(<number>)`

Returns the arc tangent of a number. Returns a value in the range -Pi/2 to Pi/2.

Example `select atan(x) from y;`

ATAN2 `ATAN2(<number1>, <number2>)`

Returns the arctangent of (<number1>/<number2>). Returns a value in the range -Pi to Pi.

Example `select atan2(x, y) from z;`

COS `COS(<number>)`

Returns the cosine of a number. The angle is specified in radians and returns a value in the range -1 to 1.

Example `select cos(x) from y;`

COSH `COSH(<number>)`

Returns the hyperbolic cosine of a number.

Example `select cosh(x) from y;`

COT `COT(<number>)`

Returns the reciprocal of the tangent of <number>.

Example `select cot(x) from y;`

SIN `SIN(<number>)`

Returns the sine of an input number that is expressed in radians.

Example `select sin(x) from y;`

SINH `SINH(<number>)`

Returns the hyperbolic sine of a number.

Example `select sinh(x) from y;`

TAN `TAN(<number>)`

Returns the tangent of an input number that is expressed in radians.

Example `select tan(x) from y;`

TANH `TANH(<number>)`

Returns the hyperbolic tangent of a number.

Example `select tanh(x) from y;`

Binary Functions

BIN_AND `BIN_AND(<number> [, <number> ...])`

Returns the result of a binary AND operation performed on all arguments.

Example `select bin_and(flags, 1) from x;`

BIN_NOT `BIN_NOT(<number>)`

v.2.5+ Returns the result of a binary NOT operation performed on the argument.

Example `select bin_not(flags) from x;`

BIN_OR `BIN_OR(<number> [, <number> ...])`
Returns the result of a binary OR operation performed on all arguments.

Example `select bin_or(flags1, flags2) from x;`

BIN_SHL `BIN_SHL(<number>, <number>)`
Returns the result of a binary shift left operation performed on the arguments (first << second).

Example `select bin_shl(flags1, 1) from x;`

BIN_SHR `BIN_SHR(<number>, <number>)`
Returns the result of a binary shift right operation performed on the arguments (first >> second).

Example `select bin_shr(flags1, 1) from x;`

BIN_XOR `BIN_XOR(<number> [, <number> ...])`
Returns the result of a binary XOR operation performed on all arguments.

Example `select bin_xor(flags1, flags2) from x;`

Miscellaneous Functions

GEN_ID `GEN_ID(<generator-name>, <step>)`
Returns a value from the named generator. The number returned depends on the <step> argument (a SMALLINT):

- a step of 0 returns the last-generated value
- a step of 1 returns a unique, new number that is 1 greater than the last generated
- a step of 10 returns a unique, new number that is 10 greater than the last generated
- a step of -1 returns a number that is 1 less than the last generated and ALSO decrements the last-generated value—usually a recipe for disaster

Example `select gen_id (MY_GENERATOR, 1) from RDB$DATABASE;`

Comment From v.2.0 forward, you might prefer to use the alternative method for retrieving the next value from a generator, alias SEQUENCE in standard SQL. Refer to the NEXT VALUE FOR syntax for sequences.

External Functions

This summary covers the external function libraries that are distributed officially with the Firebird binary kits, viz., `fbudf[.dll]` and `ib_udf[.dll]`.

The popular, original 32-bit Windows-only *FreeUDFLib* created by Gregory Deatz in Delphi in the 1990s is not strenuously maintained any more and, at the time of writing, a 64-bit version was not known to exist.

Many of the functions contained in it have been taken up and improved as *FreeAdHocUDF*, an open source project led by Christoph Theuring. *FreeAdHocUDF* is actively maintained and is available for multiple OS platforms in both 64-bit and 32-bit versions. Many of the string functions support multi-byte character sets, including UTF8. For more information, including documentation, visit <http://freedhocudf.org>.

Conditional Logic Functions

The external functions in this group should be considered deprecated for Firebird versions 1.5 and higher.

INULLIF **INULLIF (value1, value2) -- returns a NUMERIC or NULL**
(fbudf)

Arguments **value1** and **value2** can be columns, constants, variables or expressions. The function returns NULL if **value1** and **value2** resolve to a match; if there is no match, then **value1** is returned.

Applicable to fixed numeric types only and should be used only with Firebird 1.0.x. With Firebird 1.5 and higher, use the internal function `NULLIF()`.

Example This statement will cause the STOCK value in the PRODUCTS table to be set to NULL on all rows where it is currently stored as 0:

```
UPDATE PRODUCTS
SET STOCK = iNULLIF(STOCK, 0)
```

Comment See also `sNullIf()`, internal function `NULLIF()`

INVL **INVL(value1, value2) -- return a NUMERIC or NULL**
(fbudf)

This function attempts to mimic the `NVL()` function of Oracle, for fixed numeric types only. It will cause **value2** to be output if **value1** resolves to NULL, otherwise it returns **value1**. If both **value1** and **value2** resolve to NULL, then NULL is returned.

The **value1** argument is a column or an expression involving a column. Floating-point types are not supported. If necessary, use `CAST()` in your expression to transform the value to a numeric type. **value2** is an expression or constant that will resolve to the value which is to be output if **value1** resolves to NULL.

Example The following query outputs 0 if STOCK is NULL:

```
SELECT PRODUCT_ID, PRODUCT_NAME, INVL(STOCK, 0) FROM PRODUCTS;
```

Comment See also `sNVL()`, `iNullIf()`, internal function `COALESCE()`.

<i>SNULLIF</i> <i>(fbudf)</i>	<p>SNULLIF (value1, value2) -- returns a string or NULL</p> <p>Arguments value1 and value2 can be columns, constants, variables or expressions. The function returns NULL if value1 and value2 resolve to a match; if there is no match, then value1 is returned.</p> <p>Applicable to character types only and should be used only with Firebird 1.0.x. With Firebird 1.5 and higher, use the internal function NULLIF().</p>
Example	<p>This query will set the column IS_REGISTERED to NULL in all rows where its value is 'T' and REGISTRATION_DATE is NULL:</p> <pre>UPDATE ATABLE SET IS_REGISTERED = SNULLIF(IS_REGISTERED, 'T') WHERE REGISTRATION_DATE IS NULL;</pre>
Comment	<p><i>Deprecated for versions 1.5 onward.</i></p> <p><i>The string length limit of 32,765 bytes applies.</i></p> <p><i>See also iNullIf(). For Firebird 1.5 and higher, see internal function NULLIF().</i></p>
<i>SNVL</i> <i>(fbudf)</i>	<p>SNVL(value1, value2) -- returns a string or NULL</p> <p>This function attempts to mimic the NVL() function of Oracle, for character types only. It will cause value2 to be output if value1 resolves to NULL, otherwise it returns value1. If both value1 and value2 resolve to NULL, then NULL is returned.</p> <p>The value1 argument is a column or an expression involving a column. value2 is an expression or constant that will resolve to the value which is to be output if value1 resolves to NULL.</p>
Example	<p>The following query calculates and outputs a runtime column, BIRTH_YEAR, for each student. Where this value resolves to NULL, the string "Not known" is output instead:</p> <pre>SELECT FIRST NAME, LAST NAME, SNVL(CAST(EXTRACT(YEAR FROM BIRTH_DATE) AS VARCHAR(9)), 'Not known') AS BIRTH_YEAR FROM STUDENT_REGISTER;</pre>
Comment	<p><i>See also iNVL(), sNullIf(), internal function COALESCE().</i></p>

Mathematical Functions

In many, if not all cases, external math functions have been superseded by inbuilt (internal) functions from v.2.1 onward.

<i>ABS</i> <i>(ib_udf)</i>	<p>ABS(value) -- returns DOUBLE PRECISION</p> <p>Returns the absolute value of a number. The argument value is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number.</p>
Example	<p>This statement will calculate a total of values that are negative and return it as a positive value:</p> <pre>SELECT ABS(SUM(ASSET_VALUE)) AS LIABILITY FROM ASSET_REGISTER WHERE ASSET_VALUE < 0;</pre>
Comment	<p><i>See the inbuilt function ABS().</i></p>

BIN_AND **BIN_AND(value1, value2) -- returns INTEGER**

(ib_udf)

Returns the result of a binary (bitwise) AND operation performed on the two input values, **value1** and **value2**, which are columns or expressions that evaluate to SMALLINT or INTEGER type.

Example **SELECT BIN_AND(128,24) AS ANDED_RESULT FROM RDB\$DATABASE;**

BIN_OR **BIN_OR (value1, value2) -- returns INTEGER**

(ib_udf)

Returns the result of a binary (bitwise) OR operation performed on the two input values, **value1** and **value2**, which are columns or expressions that evaluate to SMALLINT or INTEGER type.

Example **SELECT BIN_AND(128,24) AS ORED_RESULT FROM RDB\$DATABASE;**

BIN_XOR **BIN_XOR (value1, value2) -- returns INTEGER**

(ib_udf)

Returns the result of a binary (bitwise) XOR operation performed on the two input values, **value1** and **value2**, which are columns or expressions that evaluate to SMALLINT or INTEGER type.

Example **SELECT BIN_XOR(128,24) AS EXORED_RESULT FROM RDB\$DATABASE;**

CEILING **CEILING (value) -- returns DOUBLE PRECISION**

(ib_udf)

Returns the smallest integer that is greater than or equal to the input value, a column or expression that evaluates to a number of DOUBLE PRECISION type.

Example **SELECT CEILING(LAST_TOTAL) AS ROUND_UP_NEAREST
 FROM SALES_HISTORY;**

Comment *Return value is a DOUBLE PRECISION number with a zero decimal part.*

DIV **DIV (value1, value2) -- returns DOUBLE PRECISION**

(ib_udf)

Divides the two integer inputs and returns the quotient, discarding the decimal part. Arguments **value1** and **value2** are columns or expressions that evaluate to numbers of SMALLINT or INTEGER type. The function behaves almost like integer-by-integer division in databases created in dialect 3—the distinction being that the division returns a double.

Example **SELECT DIV(TERM, (CURRENT_DATE - START_DATE)/365) AS YEARS_REMAINING
 FROM MORTGAGE_ACCOUNT
 WHERE ACCOUNT_ID = 12345;**

Comment *Return value is a DOUBLE PRECISION number with a zero decimal part.*

DPOWER (<i>fbudf</i>)	DPOWER (value, exponent) -- returns DOUBLE PRECISION Takes a number value or expression and an exponent and returns to exponential value.
Example	SELECT DPOWER(2.64575,2) AS NEARLY_7 FROM RDB\$DATABASE;
Comment	See also SQRT().
FLOOR (<i>ib_udf</i>)	FLOOR(value) -- returns DOUBLE PRECISION Returns a floating-point value representing the largest integer that is less than or equal to value. The value argument is a column or expression that evaluates to a number of DOUBLE PRECISION type.
Example	SELECT FLOOR(CURRENT_DATE - START_DATE) AS DAYS_ELAPSED FROM DVD_LOANS;
Comment	Return value is a DOUBLE PRECISION number with a zero decimal part.
LN (<i>ib_udf</i>)	LN (value) -- returns DOUBLE PRECISION Returns the natural logarithm of a number. The argument value is a column or expression that evaluates to a number of DOUBLE PRECISION type.
Example	SELECT LN((CURRENT_TIMESTAMP - LEASE_DATE)/7) AS NLOG_WEEKS FROM LEASE_ACCOUNT;
Comment	For v.2.1 +, see internal function LN().
LOG (<i>ib_udf</i>)	LOG(value1, value2) -- returns DOUBLE PRECISION Returns the logarithm base x= value1 of number y= value2 . Argument value1 (the logarithm base) and value2 (the number to be operated on) are columns or expressions that evaluate to numbers of DOUBLE PRECISION type.
Example	SELECT LOG(8, (CURRENT_TIMESTAMP - LEASE_DATE)/7) AS LOG_WEEKS FROM LEASE_ACCOUNT;
Comment	The Firebird 1.0.x version of this function has a bug that was inherited from InterBase: LOG(x,y) erroneously inverts the arguments and returns the log base y of number x. It was corrected in the v.1.5 library. Be aware that very old stored procedures and application code may incorporate workaround code for the bug. For v.2.1 +, see internal function LOG().
LOG10 (<i>ib_udf</i>)	LOG10 (value) -- returns DOUBLE PRECISION Returns the logarithm base 10 of the input value , a column or expression that evaluates to a number of DOUBLE PRECISION type.
Example	SELECT LOG10((CURRENT_TIMESTAMP - LEASE_DATE)/7) AS LOG10_WEEKS FROM LEASE_ACCOUNT;
Comment	For v.2.1 +, see internal function LOG10().

MODULO MODULO (value1, value2) -- returns DOUBLE PRECISION
(ib_udf) Modulo function—returns the remainder from a division between two integers. Arguments **value1** and **value2** are columns or expressions that evaluate to numbers of SMALLINT or INTEGER type.

Example A snippet from a trigger:
 ...IF (MODULO(NEW.HOURS * 100, 775) > 25.0) THEN
 NEW.OVERTIME_HOURS = MODULO(NEW.HOURS * 100, 775) / 100;

Comment For v.2.1 +, see internal function MOD().

PI PI() -- returns DOUBLE PRECISION
(ib_udf) Returns the value of Pi = 3.14159 . . . This function takes no arguments but the parentheses are required.

Example SELECT PI() AS PI_VALUE
 FROM RDB\$DATABASE;

Comment For v.2.1 +, see internal function PI().

RAND RAND() -- returns FLOAT
(ib_udf) Returns a random number between 0 and 1. Function takes no arguments, but parentheses are required.

Until Firebird 2.0, the current time is used to seed the random number generator. The older implementation can cause identical numbers to be generated if the function is called more than once in a second.

From Firebird 2.0, seeding is itself random and the problem of duplicate numbers is avoided. If you want to retain the old behaviour on a v.2.0 or higher server version, use SRAND() instead.

Example SELECT RAND() AS RANDOM_NUMBER FROM RDB\$DATABASE;

Comment Note that this function does not work in early sub-releases of Firebird 1.5.
 For v.2.1 +, see internal function RAND().

ROUND ROUND (value) -- returns a number of an integer type
(fbudf) Rounds a fixed numeric number up or down to the nearest integer.
 The **value** argument is a column or expression that evaluates to a fixed numeric type with a scale > 0.

This is plain rounding—if the digit immediately to the right of the decimal is equal to or greater than 5, it adds 1 to the digit at the left of the decimal point, and then truncates any digits to the right. Otherwise, it truncates all digits to the right of decimal point.

Example The following statement calculates an estimate based on the result of rounding the product of two NUMERIC(11,2) numbers up or down:

```
SELECT
    JOB_NO,
```

```
ROUND(RATE * HOURS) + 1 AS ESTIMATE
FROM QUOTATION
WHERE RATE IS NOT NULL AND HOURS IS NOT NULL;
```

Comment See also *TRUNCATE()*. For v.2.1 +, see internal function *ROUND()*.

SIGN *SIGN*(value) -- returns SMALLINT (1=positive | 0=zero | -1=negative)
(ib_udf) Returns 1, 0, or -1 depending on whether the input value is positive, zero, or negative, respectively. The **value** argument is a column or expression that evaluates to a number of DOUBLE PRECISION type.

Example A snippet from a trigger:
 e...IF (*SIGN*(NEW.CURRENT_VALUE) < 1) THEN ...

Comment For v.2.1 +, see internal function *SIGN()*.

SQRT *SQRT* (value) -- returns DOUBLE PRECISION
(ib_udf) Returns the square root of a number, **value**, that is a column or expression resolving to a number of DOUBLE PRECISION type.
 ~TSR

Example A snippet from a trigger:
 ...
 IF (*SQRT*(NEW.HYPOTENUSE) = *SQRT*(NEW.SIDE1) + *SQRT*(NEW.SIDE2)) THEN
 NEW.RIGHT_ANGLED_TRIANGLE = 'T';

Comment For v.2.1 +, see internal function *SQRT()*.

SRAND *SRAND*() -- returns FLOAT
(ib_udf) Returns a random number between 0 and 1. Function takes no arguments, but parentheses are required. The current time is used to seed the random number generator.

Example SELECT *SRAND*() AS RANDOM_NUMBER FROM RDB\$DATABASE;

Comment Note that this function was introduced at v.2.0 to retain the original seeding mechanism that was used by *RAND()* in older versions.
 For v.2.1 +, see internal function *RAND()*.

TRUNCATE *TRUNCATE* (value) -- returns a number of an integer type
(fbudf) Truncates a fixed numeric type, **value**, to the next lowest integer. The value argument is a column or expression that resolves to a fixed numeric type with a scale > 0.
As with some other functions in this library, you need to make two declarations to get “coverage” for both 32-bit and 64-bit inputs. Check the declarations in the script *fdudf.sql* for *truncate* and *i64truncate*.

Example The following statement calculates an estimate based on the result of truncating the product of two NUMERIC(11,2) numbers:


```

SELECT
  JOB_NO,
  TRUNCATE(RATE * HOURS) + 1 AS ESTIMATE
FROM QUOTATION
WHERE RATE IS NOT NULL AND HOURS IS NOT NULL;

```

Comment *For v.2.1 +, see internal function TRUNCATE().*

Date and Time Functions

DOW
(fbudf) DOW (value) -- returns a string

Takes a **TIMESTAMP** and returns the day of the week (in English) as a mixed-case string. The **value** argument is a column or expression that evaluates to a **TIMESTAMP** type.

Return values 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', or 'Sunday'.

Example This statement adds four days and returns the day of the week for that adjusted date:

```

SELECT DOW(CURRENT_DATE + 4)
FROM RDB$DATABASE;

```

Comment *See also SDOW(), internal function EXTRACT().*

SDOW
(fbudf) SDOW (value) -- returns a string

Takes a **TIMESTAMP** and returns the abbreviated day of the week (in English) as a mixed-case string. The **value** argument is a column or expression that evaluates to a **TIMESTAMP** type.

Return values 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', or 'Sun'.

Example This statement adds four days and returns the day of the week for that adjusted date:

```

SELECT SDOW(CURRENT_DATE + 4)
FROM RDB$DATABASE;

```

Comment *See also DOW(), also internal function EXTRACT().*

ADDDAY
(fbudf) ADDDAY (value1, value2) -- returns a **TIMESTAMP**

Adds a whole number of days to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument value1 is a column or expression that evaluates to a date or time type, value2 is the number of days to add (integer) or an integer expression.

If the input is a **TIME** type, the days are added to that time of day on the current date. If it is a **DATE** type, the time of day will be midnight.

Example This statement adds 4 days and returns the adjusted date and time as midnight, 4 days later:

```

SELECT ADDDAY(CURRENT_DATE, 4)
FROM RDB$DATABASE;

```

Comment *See also ADDHOUR(), ADDMINUTE(), etc. For v.2.1 +, see internal function DATEADD().*

ADDDHOUR (value1, value2) -- returns a **TIMESTAMP***(fbudf)*

Adds a whole number of hours to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument value1 is a column or expression that evaluates to a date or time type, value2 is the number of hours to add (integer) or an integer expression.

If the input is a **TIME** type, the hours are added to that time of day on the current date. If it is a **DATE** type, they are added to midnight of the current date. The adjusted **TIMESTAMP** is equivalent to **value1 + (value2/12)**.

Example This statement adds 10 hours and returns the adjusted date and time:

```
SELECT ADDHOUR(CURRENT_TIMESTAMP, 10)
FROM RDB$DATABASE;
```

Comment See also **ADDDAY()**, **ADDMINUTE()**, etc. For v.2.1 +, see internal function **DATEADD()**.

ADDMILLISEC (value1, value2) -- returns a **TIMESTAMP***(fbudf)*

Adds a whole number of milliseconds to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument value1 is a column or expression that evaluates to a date or time type, value2 is the number of hours to add (integer) or an integer expression.

If the input is a **TIME** type, the milliseconds are added to that time of day on the current date. If it is a **DATE** type, the input time of day will be midnight.

Example This statement adds 61,234 milliseconds to the current system time. It effectively adds the milliseconds to the current system timestamp:

```
SELECT ADDMILLISECOND(CURRENT_TIME, 61234)
FROM RDB$DATABASE;
```

Comment See also **ADDDAY()**, **ADDHOUR()**, etc. For v.2.1 +, see internal function **DATEADD()**.

ADDMINUTE (value1, value2) -- returns a **TIMESTAMP***(fbudf)*

Adds a whole number of minutes to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument value1 is a column or expression that evaluates to a date or time type, value2 is the number of hours to add (integer) or an integer expression.

If the input is a **TIME** type, the minutes are added to that time of day on the current date. If it is a **DATE** type, the input time of day will be midnight.

Example This statement adds 45 minutes to the current system time. It effectively adds the minutes to the current system timestamp:

```
SELECT ADDMINUTE(CURRENT_TIME, 45)
FROM RDB$DATABASE;
```

Comment See also **ADDDAY()**, **ADDHOUR()**, etc. For v.2.1 +, see internal function **DATEADD()**.

ADDMONTH **ADDMONTH(value1, value2) -- returns a TIMESTAMP**

(fbudf)

Adds a whole number of months to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument **value1** is a column or expression that evaluates to a date or time type, **value2** is the number of months to add (integer) or an integer expression.

The result is an adjusted **TIMESTAMP** that is (**value2**) calendar months later than **value1**. If the input is a **TIME** type, the months are added to that time of day on the current date. If it is a **DATE** type, the input time of day will be midnight.

Example This statement uses **ADDMONTH()** to calculate the term for a contract:

```
UPDATE CONTRACT SET FINAL_DATE =
CASE CONTRACT_TERM
  WHEN 'HALF-YEARLY' THEN ADDMONTH(START_DATE, 6)
  --- (Cont.)
  WHEN 'YEARLY' THEN ADDMONTH(START_DATE, 12)
ELSE
  ADDWEEK(START_DATE, TRUNCATE(CONTRACT_AMT/WEEKLY_FEE))
END
WHERE START_DATE IS NOT NULL
AND AMT_PAID IS NOT NULL
AND WEEKLY_FEE IS NOT NULL
AND CONTRACT_ID = 12345;
```

Comment See also **ADDDAY()**, **ADDHOUR()**, etc. For v.2.1 +, see internal function **DATEADD()**.

ADDSECOND **ADDSECOND (value1, value2) -- returns a TIMESTAMP**

(fbudf)

Adds a whole number of seconds to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument **value1** is a column or expression that evaluates to a date or time type, **value2** is the number of seconds to add (integer) or an integer expression.

If the input is a **TIME** type, the seconds are added to that time of day on the current date. If it is a **DATE** type, the input time of day will be midnight.

Example This statement adds 120 seconds to the current system date. It effectively adds the seconds to midnight of the current system date:

```
SELECT ADDSECOND(CURRENT_DATE, 120) FROM RDB$DATABASE;
```

Comment See also **ADDDAY()**, **ADDHOUR()**, etc. For v.2.1 +, see internal function **DATEADD()**.

ADDWEEK **ADDWEEK (value1, value2) -- returns a TIMESTAMP**

(fbudf)

Adds a whole number of weeks to a date or time type value and returns the adjusted date as a **TIMESTAMP**. The argument **value1** is a column or expression that evaluates to a date or time type, **value2** is the number of weeks to add (integer) or an integer expression.

The returned value is an adjusted **TIMESTAMP**, equivalent to (**value1** + (7 * **value2**)). If the input is a **TIME** type, the weeks are added to that time of day on the current date. If it is a **DATE** type, the input time of day will be midnight.

Example This statement calculates how many weeks' fees were paid, and uses it with **ADDWEEK()** to calculate the final date of a contract:

```
UPDATE CONTRACT
SET FINAL_DATE = ADDWEEK(START_DATE, TRUNCATE(CONTRACT_AMT/WEEKLY_FEE))
WHERE START_DATE IS NOT NULL
AND AMT_PAID IS NOT NULL
AND WEEKLY_FEE IS NOT NULL
AND CONTRACT_ID = 12345;
```

Comment See also `ADDDAY()`, `ADDHOUR()`, etc. For v.2.1 +, see internal function `DATEADD()`.

ADDYEAR `ADDYEAR (value1, value2) -- returns a TIMESTAMP`
(fbudf)

Adds a whole number of years to a date or time type value and returns the adjusted date as a `TIMESTAMP`. The argument **value1** is a column or expression that evaluates to a date or time type, **value2** is the number of years to add (integer) or an integer expression.

If the input is a `TIME` type, the years are added to that time of day on the current date. If it is a `DATE` type, the input time of day will be midnight.

Example This statement calculates the final date of a lease, given the starting date:

```
UPDATE LEASE
SET FINAL_DATE = ADDYEAR(START_DATE, 5)
WHERE START_DATE IS NOT NULL AND LEASE_ID = 12345;
```

Comment See also `ADDDAY()`, `ADDHOUR()`, etc. For v.2.1 +, see internal function `DATEADD()`.

GetExactTime `GETEXACTTIMESTAMP() -- returns a TIMESTAMP`
stamp
(fbudf)

Returns the system time as a `TIMESTAMP`, to the nearest millisecond. It takes no arguments.

In versions prior to v.2.1, the date and time context variable `CURRENT_TIMESTAMP` and the predefined date literal `'NOW'` return system time only to the nearest second. For the older versions, `GETEXACTTIMESTAMP()` is the only way to get the exact system time.

Example `SELECT GETEXACTTIMESTAMP() AS TSTAMP FROM RDB$DATABASE;`

String and Character Functions

If you are upgrading to Firebird 2.0 or higher, you should note that a number of the string functions in the *ib_udf* library have been enhanced to accept `NULL` inputs. The affected functions are `ASCII_CHAR`, `LOWER`, `"LOWER"`, `LPAD`, `LTRIM`, `RPAD`, `RTRIM`, `SUBSTR` and `SUBSTRLEN`.

To behave correctly when passing `NULL`, these functions must be declared as in this example for `ASCII_CHAR()`:

```
DECLARE EXTERNAL FUNCTION ascii_char
INTEGER NULL
RETURNS CSTRING(1) FREE_IT
ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME 'ib_udf';
```

ASCII_CHAR (<i>ib_udf</i>)	<p>ASCII_CHAR (value) -- returns a string</p> <p>Returns the single-byte printable or non-printable ASCII character corresponding to the decimal value passed in value, which can be a column, constant, or expression of type SMALLINT or INTEGER.</p>
Example	<p>This statement will insert a carriage return and a line feed into a column on every row of an external table:</p> <pre>UPDATE EXT_FILE SET EOL= ASCII_CHAR(13) ASCII_CHAR(10);</pre>
Comment	<p>Firebird 2.1 +, see internal function ASCII_CHAR().</p> <p>To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.</p>
ASCII_VAL (<i>ib_udf</i>)	<p>ASCII_VAL(value) -- returns an integer type</p> <p>Returns the decimal ASCII value of the character passed in. The arguments value is a column, constant, or expression of type CHAR.</p>
Example	<pre>SELECT ASCII_VAL('&') AS ASC_NUM FROM RDB\$DATABASE;</pre>
Comment	<p>Firebird 2.1 +, see internal function ASCII_VAL()</p>
LOWER (<i>ib_udf</i>)	<p>LOWER (value) -- returns a string</p> <p>Returns the input string converted to lower case characters. It works only with ASCII characters. The argument value is a column or expression that evaluates to an ASCII string of 32,765 bytes or less. The return value is a CHAR(n) or VARCHAR(n) of the same size as the input string.</p>
Example	<p>The following statement will return the string 'come and sit at my table':</p> <pre>SELECT LOWER('Come and Sit at MY TABLE') AS L_STRING FROM RDB\$DATABASE;</pre>
Comment	<p>This function can receive and return up to 32,767 characters, the limit for a Firebird character string. Treat the limit as 32,765 characters unless you are certain that the result will never be returned into a VARCHAR() column or variable.</p> <p>To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.</p> <p>In Firebird 2.0, it is recommended you drop the old declaration of LOWER() and use the internal function LOWER() instead.</p> <p>If you must continue to use the external function under Firebird 2.0 or later, drop the old declaration and redeclare it as "LOWER" (with the double quotes, see below).</p>

"LOWER" "LOWER" (value) -- returns a string

(ib_udf)

In Firebird 2.0+, the redeclaration of the external function LOWER() with a double-quoted identifier is strongly recommended if you must keep using an external function for lower-casing a string, to avoid conflicting with the internally implemented LOWER() function.

It is an identifier change only: the function itself works exactly like the LOWER() external function as described above. It is a case of dropping the function declaration and declaring it afresh with the surrounding double quotes.

Tip *The file `ib_sql2.sql` in the `/UDF/` directory of v.2+ installations has a script for this redeclaration.*

Comment *To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.*
If you have a choice, use the internal function LOWER().

LPAD LPAD(value, length, in_char) -- returns a string

(ib_udf)

Prepends the given character `in_char` to the beginning of the input string, `value`, until the length of the result string becomes equal to the given number, `length`.

The `value` argument should be a column or expression that evaluates to a string not longer than (32767 - `length`) bytes.

The `length` argument should be an integer type or expression.

The `in_char` argument should be a single character, to be used as the padding character.

The result is A CHAR(n) OR VARCHAR(n), where n is the supplied length argument.

Example The following statement will return the string '#####RHUBARB':

```
SELECT LPAD('RHUBARB', 17, '#') AS LPADDED_STRING FROM RDB$DATABASE;
```

Comment *This function can receive and return up to 32,767 characters, the limit for a Firebird character string. Treat the limit as 32,765 characters unless you are certain that the result will never be returned into a VARCHAR() column or variable.*
To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.
See also RPAD(). For v.2.1 +, see the internal function LPAD().

LTRIM LTRIM (value) -- returns a string

(ib_udf)

Removes leading spaces from the input string. The input argument `value` is a column or expression that evaluates to a string not longer than 32,767 bytes. The string returns a CHAR(n) OR VARCHAR(n) with no leading space characters.

This function can accept up to 32,765 bytes, including space characters, the limit for a Firebird VARCHAR.

Example The following Before Insert trigger fragment will trim any leading spaces from the input:

```
NEW.CHARACTER_COLUMN = LTRIM(NEW.CHARACTER_COLUMN);
```

Comment *To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.*
 See also RTRIM() and, for v.2.0+, the internal function TRIM().

RPAD RPAD(value, length, in_char) -- returns a string
(ib_udf) Appends the given character **in_char** to the end of the input string, **value**, until the length of the result string becomes equal to the given number, **length**.
 The **value** argument should be a column or expression that evaluates to a string not longer than (32767 - **length**) bytes.
 The **length** argument should be an integer type or expression.
 The **in_char** argument should be a single character, to be used as the padding character.
 The result is A CHAR(n) OR VARCHAR(n), where n is the supplied length argument.

Example The following statement will return the string 'Framboise*****'
 TSRSELECT RPAD('Framboise, 20, '*') AS RPADDED_STRING
 FROM RDB\$DATABASE;

Comment *This function can receive and return up to 32,767 characters, the limit for a Firebird character string. Treat the limit as 32,765 characters unless you are certain that the result will never be returned into a VARCHAR() column or variable.*
 To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.
 See also LPAD(). For v.2.1 +, see the internal function RPAD().

RTRIM RTRIM (value) -- returns a string
(ib_udf) Removes trailing spaces from the input string. The input argument **value** is a column or expression that evaluates to a string not longer than 32,767 bytes. The string returns a CHAR(n) OR VARCHAR(n) with no trailing space characters.
 This function can accept up to 32,765 bytes, including space characters, the limit for a Firebird VARCHAR.

Example The following Before Insert trigger fragment will trim anytrailing spaces from the input:
 NEW.CHARACTER_COLUMN = RTRIM(NEW.CHARACTER_COLUMN);

Comment *To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.*
 See also LTRIM() and, for v.2.0+, the internal function TRIM().

SRIGHT **SRIGHT (value, length) -- returns a string**

(fbudf)

Returns a substring from the supplied **value**, being the rightmost **length** characters in **value**. The **value** argument is a column or expression that evaluates to a string not longer than 32,767 bytes; **length** is an integer type or expression.

This function can accept up to 32,765 bytes, the limit for a Firebird VARCHAR.

Example The following statement will return the string 'fox jumps over the lazy dog':
 SELECT SRIGHT('The quick brown fox jumps over the lazy dog.', 28) AS R_STRING
 FROM RDB\$DATABASE;

Comment See also SUBSTR(), SUBSTRLEN(), internal function SUBSTRING().

STRLEN **STRLEN (value) -- returns INTEGER**

(ib_udf)

Returns the length of a string. The argument **value** is a column or expression that evaluates to a string not longer than 32,765 bytes. The integer return value is the count of characters in the string.

This function can accept up to 32,765 bytes, including space characters, the limit for a Firebird VARCHAR.

Example The following PSQL fragment returns the length of a column to a local variable:

```
...
DECLARE VARIABLE LEN INTEGER;
...
SELECT COL1, COL2, COL3 FROM ATABLE
INTO :V1, :V2, :V3;
LEN = STRLEN(V3);
...;
```

Comment See also SUBSTRLEN(). For v.2.1+, see the internal function CHARACTER_LENGTH().

SUBSTR **SUBSTR (value, pos1, pos2) -- returns a string**

(ib_udf)

Returns a string consisting of the characters from **pos1** to **pos2** inclusively. If **pos2** is past the end of the string, the function will return all characters from **pos1** to the end of the string.

The **value** argument is a column or expression that evaluates to a string. **pos1** and **pos2** are columns or expressions that evaluate to integer types. The function can accept up to 32,765 bytes, including space characters, the limit for a Firebird VARCHAR.

Example The following statement strips the first three characters from the string in COLUMNB and sets its value to be the string starting at position 4 and ending at position 100. If the string ends before position 100, the result will be all of the characters from position 4 to the end of the string:

```
UPDATE ATABLE
SET COLUMNB = SUBSTR(COLUMNB, 4, 100)
WHERE...
```

Comment If you are porting a legacy application written for InterBase, be aware that this implementation of SUBSTR() differs from that of SUBSTR() in the Borland distributions of the ib_udf library, which returns NULL when pos2 is past the end of the input string.

To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.

This function is deprecated from v.1.5 forward, in favour of the internal function SUBSTRING().

See also SUBSTRLEN(), RTRIM().

SUBSTRLEN SUBSTRLEN(value, startpos, length) -- returns a string
(*ib_udf*)

This function, deprecated by the internal function SUBSTRING() from v.1.5 forward, does not (as its name implies) return the length of anything. It returns a string of size **length** starting at **startpos**. The actual length of the string will be the lesser of the **length** argument and the number of characters from **startpos** to the end of the input **value**. The argument **value** is a column or expression that evaluates to a string not longer than 32,765 bytes; the other two arguments are integer constants or expressions.

Example The following statement takes the value of a column and updates it by removing the first three characters and, after that, removing any trailing characters if the remaining string is longer than 20 characters:

```
UPDATE ATABLE
  SET COLUMNB = SUBSTRLEN(COLUMNB, 4, 20)
WHERE...
```

Comment *To ensure correct behaviour with the NULL signal enhancement for the input parameters of this function under v.2.0+ servers, refer to the note at the head of this section. If upgrading from an older server version, the declaration must be dropped and the function redeclared to include the NULL signal.*
See also SUBSTR(), RTRIM(), and the internal functions SUBSTRING() (v.1.5+) and TRIM() (v.2.0+).

BLOB Functions

STRING2BLOB STRING2BLOB (value) -- returns a BLOB of sub_type 1 (TEXT)
(*fbudf*)

Takes a string field (column, variable, expression) and returns a text BLOB. The value argument is a column or expression that evaluates to a VARCHAR type of 300 characters or less.

Under most conditions in Firebird 1.5+, it will not be necessary to call this function. Firebird will accept a string that is within the byte-length limit of a VARCHAR directly as input to a BLOB.

Example This PSQL fragment concatenates two strings and converts the result to a text BLOB:

```
...
DECLARE VARIABLE V_COMMENT1 VARCHAR(250);
DECLARE VARIABLE V_COMMENT2 VARCHAR(45);
DECLARE VARIABLE V_MEMO VARCHAR(296) = '';
...
-- (Cont.)
```

```

SELECT <..other fields...>, COMMENT1, COMMENT2 FROM APPLICATION
WHERE APPLICATION_ID = :APP_ID
INTO <..other variables...>, :V_COMMENT1, V_COMMENT2;
IF (V_COMMENT1 IS NOT NULL) THEN
    V_MEMO = V_COMMENT1;
IF (V_COMMENT2 IS NOT NULL) THEN
BEGIN
    IF (V_MEMO = '') THEN
        V_MEMO = V_COMMENT2;
    ELSE
        V_MEMO = V_MEMO || ' ' || V_COMMENT2;
END
IF (V_MEMO <> '') THEN
    INSERT INTO MEMBERSHIP (FIRST_NAME, LAST_NAME, APP_ID, BLOB_MEMO)
    VALUES (
        :FIRST_NAME, :LAST_NAME, :APP_ID, STRING2BLOB(:V_MEMO));
...

```

Trigonometrical Functions

ACOS
(*ib_udf*)

ACOS (value) -- returns DOUBLE PRECISION

Calculates the arccosine (inverse of cosine) of a number between -1 and 1 . If the number is out of bounds, it returns *NaN*. The **value** argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a valid cosine value. The return value is a DOUBLE PRECISION number, in degrees.

Example This snippet from a trigger converts a raw cosine value to degrees:

```

...
IF (NEW.RAW_VALUE IS NOT NULL) THEN
    NEW.READING1 = ACOS(NEW.RAW_VALUE);

```

Comment See also COS(), COSH(), and other trigonometric functions. For v.2.1+, refer to the internal function of the same name.

ASIN
(*ib_udf*)

ASIN (value) -- returns DOUBLE PRECISION

Calculates the arcsine (inverse of sine) of a number between -1 and 1 . If the number is out of range, it returns *NaN*. The **value** argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a valid sine value. The return value is a DOUBLE PRECISION number, in degrees.

Example This snippet from a trigger converts a raw cosine value to degrees:

```

...
IF (NEW.RAW_VALUE IS NOT NULL) THEN
    NEW.READING1 = ASIN(NEW.RAW_VALUE);

```

Comment *See also SIN(), SINH(), and other trigonometric functions. For v.2.1+, refer to the internal function of the same name.*

ATAN ATAN (value) -- returns DOUBLE PRECISION
(ib_udf) Returns the arctangent (inverse of tangent) of the input argument **value**, which is a column or expression resolving to a valid tan(gent) value compatible with a signed or unsigned DOUBLE PRECISION number. The return value is a DOUBLE PRECISION number, in degrees.

Example This snippet from a trigger converts a raw tan(gent) value to an arc, in degrees:

```
...
IF (NEW.RAW_VALUE IS NOT NULL) THEN
    NEW.READING1 = ATAN(NEW.RAW_VALUE);
```

Comment *See also ATAN2(), TAN(), TANH(), and other trigonometric functions. For v.2.1+, refer to the internal function of the same name.*

ATAN2 ATAN2 (value1, value2) -- returns DOUBLE PRECISION
(ib_udf) Returns a value that is an arc, in degrees, calculated as the arctangent of the result of dividing one tangent by another. The arguments **value1** and **value2** are both numeric columns or expressions evaluating as DOUBLE PRECISION numbers that are valid tan(gent) values. The return value is a DOUBLE PRECISION number, the result the arctangent of (value1/value2) in degrees.

Example This PSQL snippet stores a value that is an arc, in degrees, calculated as the arctangent of the result of dividing one tangent by another:

```
...
UPDATE HEAVENLY_HAPPENINGS
    SET INCREASE_RATIO = ATAN2(INITIAL_TAN, FINAL_TAN)
WHERE HAPPENING_ID = :happening_id;
```

Comment *See also ATAN(), TAN(), TANH(). For v.2.1+, refer to the internal function of the same name.*

COS COS (value) -- returns DOUBLE PRECISION
(ib_udf) Returns the cosine of value. The **value** argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.

Important *If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0.*

Example This snippet from a trigger calculates and stores the cosine of an angular measurement input in degrees:

```
...
IF (NEW.READING1 IS NOT NULL) THEN
    NEW.RDG_COSINE = COS(NEW.READING1);
```

Comment See also *SIN()*, *COS()*, *ACOS()*, *COSH()*. For v.2.1+, refer to the internal function of the same name.

COSH *COSH* (value) -- returns DOUBLE PRECISION
(ib_udf) Returns the hyperbolic cosine of value. The value argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.

Important *If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0.*

Example This snippet from a trigger calculates and stores the hyperbolic cosine of an angular measurement input in degrees:

```
...
IF (NEW.READING1 IS NOT NULL) THEN
    NEW.RDG_COS_HYP = COSH(NEW.READING1);
```

Comment See also *SINH()*, *TANH()*, and other trigonometric functions. For v.2.1+, refer to the internal function of the same name.

COT *COT* (value) -- returns DOUBLE PRECISION
(ib_udf) Returns the cotangent of value. The value argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.

Important *If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0 ("a quiet NaN").*

Example This snippet from a trigger calculates and stores the cotangent of an angular measurement input in degrees:

```
...
IF (NEW.READING1 IS NOT NULL) THEN
    NEW.RDG_COTAN = COT(NEW.READING1);
```

Comment See also *TAN()*, *ATAN()*, *TANH()*. For v.2.1+, refer to the internal function of the same name.

SIN *SIN* (value) -- returns DOUBLE PRECISION
(ib_udf) Returns the sine of value. The value argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.

Important *If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0 ("a quiet NaN").*

Example This snippet from a trigger calculates and stores the sine of an angular measurement input in degrees:

```
...
IF (NEW.READING1 IS NOT NULL) THEN
    NEW.RDG_SINE = SIN(NEW.READING1);
```

Comment See also COS(), ASIN(), SINH(). For v.2.1+, refer to the internal function of the same name.

SINH SINH (value) -- returns DOUBLE PRECISION

(ib_udf) Returns the hyperbolic sine of **value**. The **value** argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.

Notes If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0.

Important *If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0 ("a quiet NaN").*

Example This snippet from a trigger calculates and stores the hyperbolic sine of an angular measurement input in degrees:

```
...
IF (NEW.READING1 IS NOT NULL) THEN
    NEW.RDG_SIN_HYP = SINH(NEW.READING1);
```

Comment See also SIN(), TANH(), COSH(). For v.2.1+, refer to the internal function of the same name.

TAN TAN (value) -- returns DOUBLE PRECISION

(ib_udf) Returns the tangent of **value**. The **value** argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.

Important *If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0 ("a quiet NaN").*

Example This snippet from a trigger calculates and stores the tangent of an angular measurement input in degrees:

```
...
IF (NEW.READING1 IS NOT NULL) THEN
    NEW.RDG_TAN = TAN(NEW.READING1);
```

Comment See also COT(), ATAN(), TANH(). For v.2.1+, refer to the internal function of the same name.

TANH <i>(ib_udf)</i>	TANH (value) -- returns DOUBLE PRECISION Returns the hyperbolic tangent of value . The value argument is a column or expression that is compatible with a signed or unsigned DOUBLE PRECISION number, evaluating to a value (in degrees) between -263 and 263. The return value is a DOUBLE PRECISION number, or 0 if the input is out of range.
Important	<i>If value is greater than or equal to 263, or less than or equal to -263, there is a loss of significance in the result of the call, and the function generates a _TLOSS error and returns a 0.</i>
Example	This snippet from a trigger calculates and stores the hyperbolic tangent of an angular measurement input in degrees: ... IF (NEW.READING1 IS NOT NULL) THEN NEW.RDG_TAN_HYP = TANH(NEW.READING1);
Comment	See also TAN(), ATAN(). For v.2.1+, refer to the internal function of the same name.

Building regular expressions

These guidelines are extracted from the **Firebird 2.5 Language Reference Update**, by Paul Vinkenoog and other Firebird Project authors.

Characters

Within regular expressions, most characters represent themselves. The only exceptions are the special characters below:

[] () | ^ - + * % _ ? {
...and the escape character, if it is defined.

A regular expression that doesn't contain any special or escape characters only matches strings that are identical to itself (subject to the collation in use). That is, it functions just like the “=” operator:

'Apple' similar to 'Apple' -- true
'Apples' similar to 'Apple' -- false
'Apple' similar to 'Apples' -- false
'APPLE' similar to 'Apple' -- depends on collation

Wildcards

The known SQL wildcards _ and % match any single character and a string of any length, respectively:

'Birne' similar to 'B_rne' -- true
'Birne' similar to 'B_ne' -- false
'Birne' similar to 'B%ne' -- true
'Birne' similar to 'Bir%ne%' -- true
'Birne' similar to 'Birr%ne' -- false

Notice how % also matches the empty string.

Character classes

A bunch of characters enclosed in brackets define a character class. A character in the string matches a class in the pattern if the character is a member of the class:

```
'Citroen' similar to 'Cit[arju]oen' -- true
'Citroen' similar to 'Ci[tr]oen' -- false
'Citroen' similar to 'Ci[tr][tr]oen' -- true
```

As can be seen from the second line, the class only matches a single character, not a sequence.

Within a class definition, two characters connected by a hyphen define a range. A range comprises the two endpoints and all the characters that lie between them in the active collation. Ranges can be placed anywhere in the class definition without special delimiters to keep them apart from the other elements.

```
'Datte' similar to 'Dat[q-u]e' -- true
'Datte' similar to 'Dat[abq-uy]e' -- true
'Datte' similar to 'Dat[bcg-km-pwz]e' -- false
```

The following predefined character classes can also be used in a class definition:

[:ALPHA:]

Latin letters a..z and A..Z. With an accent-insensitive collation, this class also matches accented forms of these characters.

[:DIGIT:]

Decimal digits 0..9.

[:ALNUM:]

Union of [:ALPHA:] and [:DIGIT:].

[:UPPER:]

Uppercase Latin letters A..Z. Also matches lowercase with case-insensitive collation and accented forms with accent-insensitive collation.

[:LOWER:]

Lowercase Latin letters a..z. Also matches uppercase with case-insensitive collation and accented forms with accent-insensitive collation.

[:SPACE:]

Matches the space character (ASCII 32).

[:WHITESPACE:]

Matches vertical tab (ASCII 9), linefeed (ASCII 10), horizontal tab (ASCII 11), formfeed (ASCII 12), carriage return (ASCII 13) and space (ASCII 32).

Including a predefined class has the same effect as including all its members. Predefined classes are only allowed within class definitions. If you need to match against a predefined class and nothing more, place an extra pair of brackets around it.

```
'Erdbeere' similar to 'Erd[[:ALNUM:]]eere' -- true
'Erdbeere' similar to 'Erd[[:DIGIT:]]eere' -- false
'Erdbeere' similar to 'Erd[a[:SPACE:]b]eere' -- true
'Erdbeere' similar to [[:ALPHA:]] -- false
'E' similar to [[:ALPHA:]] -- true
```

If a class definition starts with a caret, everything that follows is excluded from the class. All other characters match:

```
'Framboise' similar to 'Fra[^ck-p]boise' -- false
'Framboise' similar to 'Fr[^a][^a]boise' -- false
```

```
'Framboise' similar to 'Fra^[[:DIGIT:]]boise' -- true
```

If the caret is not placed at the start of the sequence, the class contains everything before the caret, except for the elements that also occur after the caret:

```
'Grapefruit' similar to 'Grap[a-m^f-i]fruit' -- true
'Grapefruit' similar to 'Grap[abc^xyz]fruit' -- false
'Grapefruit' similar to 'Grap[abc^de]fruit' -- false
'Grapefruit' similar to 'Grap[abe^de]fruit' -- false
'3' similar to '[[[:DIGIT:]]^4-8]' -- true
'6' similar to '[[[:DIGIT:]]^4-8]' -- false
```

Lastly, the already mentioned wildcard “_” is a character class of its own, matching any single character.

Quantifiers

A question mark immediately following a character or class indicates that the preceding item may occur 0 or 1 times in order to match:

```
'Hallon' similar to 'Hal?on' -- false
'Hallon' similar to 'Hal?lon' -- true
'Hallon' similar to 'Hall?on' -- true
'Hallon' similar to 'Hallll?on' -- false
'Hallon' similar to 'Halx?lon' -- true
'Hallon' similar to 'H[a-c]?llon[x-z]?' -- true
```

An asterisk immediately following a character or class indicates that the preceding item may occur 0 or more times in order to match:

```
'Icaque' similar to 'Ica*que' -- true
'Icaque' similar to 'Icar*que' -- true
'Icaque' similar to 'I[a-c]*que' -- true
'Icaque' similar to '*' -- true
'Icaque' similar to '[[[:ALPHA:]]]*' -- true
'Icaque' similar to 'Ica[xyz]*e' -- false
```

A plus sign immediately following a character or class indicates that the preceding item must occur 1 or more times in order to match:

```
'Jujube' similar to 'Ju_+' -- true
'Jujube' similar to 'Ju+jube' -- true
'Jujube' similar to 'Jujuber+' -- false
'Jujube' similar to 'J[jux]+be' -- true
'Jujube' similar to 'J[[[:DIGIT:]]+ujube' -- false
```

If a character or class is followed by a number enclosed in braces, it must be repeated exactly that number of times in order to match:

```
'Kiwi' similar to 'Ki{2}wi' -- false
'Kiwi' similar to 'K[ipw]{2}i' -- true
'Kiwi' similar to 'K[ipw]{2}' -- false
'Kiwi' similar to 'K[ipw]{3}' -- true
```

If the number is followed by a comma, the item must be repeated at least that number of times in order to match:


```
'Limoné' similar to 'Li{2,}mone' -- false
'Limoné' similar to 'Li{1,}mone' -- true
'Limoné' similar to 'Li[nezom]{2,}' -- true
```

If the braces contain two numbers separated by a comma, the second number not smaller than the first, then the item must be repeated at least the first number and at most the second number of times in order to match:

```
'Mandarijn' similar to 'M[a-p]{2,5}rijn' -- true
'Mandarijn' similar to 'M[a-p]{2,3}rijn' -- false
'Mandarijn' similar to 'M[a-p]{2,3}arijn' -- true
```

The quantifiers `?`, `*` and `+` are shorthand for `{0,1}`, `{0,}` and `{1,}`, respectively.

OR-ing terms

Regular expression terms can be OR'ed with the `|` operator. A match is made when the argument string matches at least one of the terms:

```
'Nektarin' similar to 'Nek|tarin' -- false
'Nektarin' similar to 'Nektarin|Persika' -- true
'Nektarin' similar to 'M_|N_|P_+' -- true
```

Sub-expressions

One or more parts of the regular expression can be grouped into sub-expressions (also called sub-patterns) by placing them between parentheses. A sub-expression is a regular expression in its own right. It can contain all the elements allowed in a regular expression, and can also have quantifiers added to it.

```
'Orange' similar to 'O(ra|ri|ro)nge' -- true
'Orange' similar to 'O(r[a-e])+nge' -- true
'Orange' similar to 'O(ra){2,4}nge' -- false
'Orange' similar to 'O(r(an|in)g|rong)?e' -- true
```

Escaping special characters

In order to match against a character that is special in regular expressions, that character has to be escaped. There is no default escape character; rather, the user specifies one when needed:

```
'Peer (Poire)' similar to 'P[^ ]+ \([^\ ]+\)' escape '\' -- true
'Pera [Pear]' similar to 'P[^ ]+ #[P[^ ]+#]' escape '#' -- true
'Päron-Äppledryck' similar to 'P%$-Ä%' escape '$' -- true
'Pärondryck' similar to 'P%-Ä%' escape '-' -- false
```

The last line demonstrates that the escape character can also escape itself, if needed.

APPENDIX

II

RESERVED AND NON-RESERVED
KEYWORDS

Table II.1 contains keywords that are, or have been, reserved in some way in Firebird.

Not all keywords used in Firebird are covered by the SQL standards. The standard keywords are shown in **bold**.

Of those that are non-standard, most that were formerly reserved have been made non-reserved as releases have progressed. You can recognise them in the table by the designation “Y, NR” which translates as “Yes, it is a keyword but No, it is not reserved”.

In some cases, you will notice a designation that includes the abbreviation “cont.” That means the keyword is reserved in some contexts. What rules their “reservedness” is whether they are used as identifiers in a context where they conflict with language elements used in that context.

Keywords shown in square brackets, e.g. [BOOLEAN] are generally reserved words in the standards that Firebird has not implemented yet. It is not wise to use them as identifiers, since they are likely to be implemented in Firebird at some point. For example, it would be unwise to create a domain called BOOLEAN, since it is very likely that Firebird will have a native type of that name in future.

Keywords

Table II.1 Keywords Reserved and Not Reserved in Firebird SQL

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
A					
ABS	Y, NR	Y, NR	-	-	-
ACCENT	Y, NR	Y, NR	-	-	-

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
ACOS	Y, NR	Y, NR	-	-	-
ACTION	Y, NR	Y, NR	Y, NR	Y	Y
ACTIVE	Y	Y	Y	Y	Y
ADD	Y	Y	Y	Y	Y
ADMIN	Y	Y	Y	Y	Y
AFTER	Y	Y	Y	Y	Y
ALL	Y	Y	Y	Y	Y
ALTER	Y	Y	Y	Y	Y
ALWAYS	Y, NR	Y, NR	-	-	-
AND	Y	Y	Y	Y	Y
ANY	Y	Y	Y	Y	Y
ARE	Y	Y	Y	Y	Y
AS	Y	Y	Y	Y	Y
ASC ASCENDING	Y	Y	Y	Y	Y
AT	Y	Y	Y	Y	Y
ASCII_CHAR	Y, NR	Y, NR	-	-	-
ASCII_VAL	Y, NR	Y, NR	-	-	-
ASIN	Y, NR	Y, NR	-	-	-
ATAN	Y, NR	Y, NR	-	-	-
ATAN2	Y, NR	Y, NR	-	-	-
AUTO	Y (cont.)	Y (cont.)	Y	Y	Y
AUTONOMOUS	Y, NR	-	-	-	-
AUTODDL	Y (cont.)	Y (cont.)	Y (cont.)	Y	Y
AVG	Y	Y	Y	Y	Y
B					
BACKUP	Y, NR	Y, NR	Y, NR	-	-
BASE	?	?	?	Y	Y
BASENAME	N	N	N	Y	Y
BASE_NAME	-	-	-	Y	Y
BEFORE	Y	Y	Y	Y	Y
BEGIN	Y	Y	Y	Y	Y
BETWEEN	Y	Y	Y	Y	Y
BIGINT	Y, NR	Y	Y	Y	Y
BIN_AND	Y, NR	Y, NR	-	-	-
BIN_NOT	Y, NR	Y, NR (v.2.1.1)	-	-	-
BIN_OR	Y, NR	Y, NR	-	-	-
BIN_SHL	Y, NR	Y, NR	-	-	-
BIN_SHR	Y, NR	Y, NR	-	-	-
BIN_XOR	Y, NR	Y, NR	-	-	-

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
BIT_LENGTH	Y	Y	Y	-	-
BLOB				Y	Y
BLOBEDIT				Y	Y
BLOCK	Y, NR	Y, NR	Y, NR	-	-
[BOOLEAN	-	-	-	-	-
BOTH	Y, NR	Y, NR	Y	-	-
BREAK				Y, NR	Y
BUFFER				Y	Y
BY				Y	Y
C					
CACHE	N	N	N	Y	Y
CALLER	Y, NR	-	-	-	-
CASCADE	Y, NR	Y, NR	Y, NR	Y	Y
CASE	Y	Y	Y	Y	-
CAST				Y	Y
CEIL CEILING	Y, NR	Y, NR	-	-	-
CHAR CHARACTER				Y	Y
CHAR_LENGTH	Y	Y	Y	-	-
CHARACTER_LENGTH	Y	Y	Y	-	-
CHAR_TO_UUID	Y, NR	-	-	-	-
CHECK	Y	Y	Y	Y	Y
CHECK_POINT_LEN	N	N	N	Y	Y
CHECK_POINT_LENGTH				Y	Y
CLOSE	Y	Y	Y	-	-
COALESCE	Y(con.)	Y(con.)	Y (con.)	Y	Y
COLLATE				Y	Y
COLLATION	Y, NR	Y, NR	Y, NR	-	-
COLUMN				Y	Y
COMMENT	Y, NR	Y, NR	Y, NR	-	-
COMMIT				Y	Y
COMMITTED				Y	Y
COMMON	Y, NR				
COMPILETIME				Y	Y
COMPUTED				Y	Y
CONDITIONAL				Y	Y
CONNECT	Y	Y	-	-	-
CONSTRAINT				Y	Y
CONTAINING				Y	Y
CONTINUE				Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
COS	Y, NR	Y, NR	-	-	-
COSH	Y, NR	Y, NR	-	-	-
COT	Y, NR	Y, NR	-	-	-
COUNT	Y	Y	Y	Y	Y
CREATE	Y	Y	Y	Y	Y
CROSS	Y	Y	-	-	-
CSTRING	N	N	N	Y	Y
CURRENT	Y	Y	Y	Y	Y
CURRENT_CONNECTION	Y, NR	Y	Y	Y	Y
CURRENT_DATE	Y	Y	Y	Y	Y
CURRENT_ROLE	Y, NR	Y	Y	Y	Y
CURRENT_TIME	Y	Y	Y	Y	Y
CURRENT_TIMESTAMP	Y	Y	Y	Y	Y
CURRENT_TRANSACTION	Y, NR	Y	Y	Y	Y
CURRENT_USER	Y	Y	Y	Y	Y
CURSOR	Y	Y	Y	Y	Y
D					
DATA	Y, NR	-	-	-	-
DATABASE	Y, NR	Y	Y	Y	Y
DATE	Y	Y	Y	Y	Y
DATEADD	Y, NR	Y, NR	-	-	-
DATEDIFF	Y, NR	Y, NR	-	-	-
DAY	Y	Y	Y	Y	Y
DB_KEY	Y	Y	Y	Y	Y
DEBUG	Y, NR	Y	Y	Y	Y
DEC	Y	Y	Y	Y	Y
DECIMAL	Y	Y	Y	Y	Y
DECLARE	Y	Y	Y	Y	Y
DECODE	Y, NR	Y, NR	-	-	-
DEFAULT	Y	Y	Y	Y	Y
[DEFERRED]	-	-	-	N	N
DELETE	Y	Y	Y	Y	Y
DELETING (con.)	Y (con.)	Y (con.)	Y (con.)	Y	-
DESC DESCENDING				Y	Y
DIFFERENCE	Y, NR	Y, NR	Y, NR	-	-
DESCRIBE	Y	Y	Y	Y	Y
DESCRIPTOR	Y, NR	Y, NR	Y, NR	Y, NR	Y
DISCONNECT	Y	Y	-	-	-
DISPLAY	N	N	N	Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
DISTINCT	Y	Y	Y	Y	Y
DO	Y	Y	Y	Y	Y
DOMAIN	Y	Y	Y	Y	Y
DOUBLE	Y	Y	Y	Y	Y
DROP	Y	Y	Y	Y	Y
E					
ECHO	Y, NR	Y, NR	Y	Y	Y
EDIT	Y, NR	Y, NR	Y	Y	Y
ELSE	Y	Y	Y	Y	Y
END	Y	Y	Y	Y	Y
ENTRY_POINT	Y, NR	Y, NR	Y	Y	Y
ESCAPE	Y	Y	Y	Y	Y
EVENT	Y, NR	Y, NR	Y	Y	Y
EXCEPTION	Y	Y	Y	Y	Y
EXECUTE	Y	Y	Y	Y	Y
EXISTS	Y	Y	Y	Y	Y
EXIT	Y	Y	Y	Y	Y
EXP	Y, NR	Y, NR	-	-	-
EXTERN	?	?	?	Y	Y
EXTERNAL	Y	Y	Y	Y	Y
EXTRACT	Y	Y	Y	Y	Y
F					
[FALSE]	-	-	-	-	-
FETCH	Y	Y	Y	Y	Y
FILE				Y	Y
FILTER				Y	Y
FIRST	Y, NR	Y, NR	Y, NR	Y, NR	Y
FIRSTNAME	Y, NR	-	-	-	-
FLOAT	Y	Y	Y	Y	Y
FLOOR	Y, NR	Y, NR	-	-	-
FOR	Y	Y	Y	Y	Y
FOREIGN	Y	Y	Y	Y	Y
FOUND	Y	Y	Y	Y	Y
FREE_IT	Y, NR	Y, NR	Y	Y	Y
FROM	Y	Y	Y	Y	Y
FULL	Y	Y	Y	Y	Y
FUNCTION	Y	Y	Y	Y	Y
G					
GDSCODE	Y	Y	Y	Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
GENERATED	Y, NR	Y, NR	-	-	-
GENERATOR	Y, NR	Y, NR	Y, NR	Y	Y
GEN_ID	Y	Y	Y	Y	Y
GEN_UUID	Y, NR	Y, NR	-	-	-
GLOBAL	Y	Y	N	N	N
GOTO	Y, NR	Y, NR	Y, NR	Y	Y
GRANT	Y	Y	Y	Y	Y
GRANTED	Y, NR	-	-	-	-
GROUP	Y	Y	Y	Y	Y
GROUP_COMMIT_WAIT	N	N	N	Y	Y
GROUP_COMMIT_WAIT_TIME	?	?	?	Y	Y
H					
HASH	Y, NR	Y, NR	-	-	-
HAVING	Y	Y	Y	Y	Y
HEADING	?	?	?	Y	Y
HELP	?	?	?	Y	Y
HOURL	Y	Y	Y	Y	Y
I					
IF	Y	Y	Y	Y	Y
IIF	Y, NR	Y, NR	Y, NR	N	Y
IMMEDIATE	Y	Y	Y	Y	Y
IN	Y	Y	Y	Y	Y
INACTIVE	Y (con.)	Y (con.)	Y	Y	Y
INDEX	Y	Y	Y	Y	Y
INDICATOR	Y	Y	Y	Y	Y
INIT	Y, NR	Y, NR	Y, NR	Y	Y
INNER	Y	Y	Y	Y	Y
INPUT	Y	Y	Y	Y	Y
INPUT_TYPE	?	?	?	Y	Y
INSENSITIVE	Y	Y	-	-	-
INSERT	Y	Y	Y	Y	Y
INSERTING	Y (con.)	Y (con.)	Y (con.)	Y	-
INT	Y	Y	Y	Y	Y
INTEGER	Y	Y	Y	Y	Y
INTO	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y
ISOLATION	Y	Y	Y	Y	Y
ISQL	?	?	?	Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
J					
JOIN	Y	Y	Y	Y	Y
K					
KEY	Y	Y	Y	Y	Y
L					
LAST	Y (con.)	Y (con.)	Y (con.)	Y	-
LASTNAME	Y, NR	-	-	-	-
LC_MESSAGES	?	?	?	Y	Y
LC_TYPE	?	?	?	Y	Y
LEADING	Y	Y	-	-	-
LEAVE	Y (con.)	Y (con.)	Y (con.)	Y	-
LEFT	Y	Y	Y	Y	Y
LENGTH	Y, NR	Y, NR	Y, NR	Y	Y
LEV	?	?	?	Y	Y
LEVEL	Y	Y	Y	Y	Y
LIKE	Y	Y	Y	Y	Y
LIST	Y, NR	Y, NR	-	-	-
LN	Y, NR	Y, NR	-	-	-
LOCK	Y (con.)	Y (con.)	Y (con.)	Y	-
LOG	Y, NR	Y, NR	-	-	-
LOG10	Y, NR	Y, NR	-	-	-
LOGFILE	N	N	N	Y	Y
LOG_BUFFER_SIZE	?	?	?	Y	Y
LOG_BUF_SIZE	N	N	N	Y	Y
LONG	Y	Y	Y	Y	Y
LOWER	Y	Y	Y	-	-
LPAD	Y, NR	Y, NR	-	-	-
M					
MANUAL	?	?	?	Y	Y
MAPPING	Y, NR	-	-	-	-
MATCHED	Y, NR	Y, NR	-	-	-
MATCHING	Y, NR	Y, NR	-	-	-
MAX	Y	Y	Y	Y	Y
MAXIMUM	?	?	?	Y	Y
MAXIMUM_SEGMENT	?	?	?	Y	Y
MAX_SEGMENT	?	?	?	Y	Y
MAXVALUE	Y, NR	Y, NR	-	-	-
MERGE	Y	Y	Y	Y	Y
MESSAGE	?	?	?	Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
MIDDLENAME	Y, NR	-	-	-	-
MIN	Y	Y	Y	Y	Y
MINIMUM	?	?	?	Y	Y
MINUTE	Y	Y	Y	Y	Y
MINVALUE	Y, NR	Y, NR	-	-	-
MOD	Y, NR	Y, NR	-	-	-
MODULE_NAME	?	?	?	Y	Y
MONTH	Y	Y	Y	Y	Y
N					
NAMES	Y	Y	Y	Y	Y
NATIONAL	Y	Y	Y	Y	Y
NATURAL	Y	Y	Y	Y	Y
NCHAR	Y	Y	Y	Y	Y
NEXT	Y, NR	Y, NR	Y, NR	-	-
NO	Y	Y	Y	Y	Y
NOAUTO	?	?	?	Y	Y
NOT	Y	Y	Y	Y	Y
NULL	Y	Y	Y	Y	Y
NULLIF	Y (con.)	Y (con.)	Y (con.)	Y	-
NULLS	Y (con.)	Y (con.)	Y (con.)	Y	-
NUM_LOG_BUFS	N	N	N	Y	Y
NUM_LOG_BUFFERS	?	?	?	Y	Y
NUMERIC	Y	Y	Y	Y	Y
O					
OCTET_LENGTH	Y	Y	Y	-	-
OF	Y	Y	Y	Y	Y
ON	Y	Y	Y	Y	Y
ONLY	Y	Y	Y	Y	Y
OPEN	Y	Y	Y	-	-
OPTION	Y	Y	Y	Y	Y
OR	Y	Y	Y	Y	Y
ORDER	Y	Y	Y	Y	Y
OS_NAME	Y, NR	-	-	-	-
OUTER	Y	Y	Y	Y	Y
OUTPUT	Y	Y	Y	Y	Y
OUTPUT_TYPE	?	?	?	Y	Y
OVERFLOW	?	?	?	Y	Y
OVERLAY	Y, NR	Y, NR	-	-	-

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
P					
PAD	Y, NR	Y, NR	-	-	-
PAGE	?	?	?	Y	Y
PAGELength	?	?	?	Y	Y
PAGES	?	?	?	Y	Y
PAGE_SIZE	?	?	?	Y	Y
PARAMETER	Y	Y	Y	Y	Y
PASSWORD	?	?	?	Y	Y
PI	Y, NR	Y, NR	-	-	-
PLACING	Y, NR	Y, NR	-	-	-
PLAN	Y	Y	Y	Y	Y
POSITION	Y	Y	Y	Y	Y
POST_EVENT	Y	Y	Y	Y	Y
POWER	Y, NR	Y, NR	-	-	-
PRECISION	Y	Y	Y	Y	Y
PREPARE	Y	Y	Y	Y	Y
PRESERVE	Y, NR	Y, NR	-	-	-
PRIMARY	Y	Y	Y	Y	Y
PRIVILEGES	Y	Y	Y	Y	Y
PROCEDURE	Y	Y	Y	Y	Y
PUBLIC	Y	Y	Y	Y	Y
Q					
QUIT	Y (cont.)	Y (cont.)	Y (cont.)	Y	Y
R					
RAND	Y, NR	Y, NR	-	-	-
RAW_PARTITIONS	N	N	N	Y	Y
RDB\$DB_KEY	?	?	?	Y	Y
READ	Y	Y	Y	Y	Y
REAL	Y	Y	Y	Y	Y
RECORD_VERSION	Y	Y	Y	Y	Y
RECREATE	Y	Y	Y	Y	Y
RECURSIVE	Y	Y	-	-	-
REFERENCES	Y	Y	Y	Y	Y
RELEASE	Y	Y	Y	Y	Y
REPLACE	Y, NR	Y, NR	-	-	-
RESERVE	Y	Y	Y	Y	Y
RESERVING	Y	Y	Y	Y	Y
RESTART	Y, NR	Y, NR	Y, NR	-	-
RESTRICT	-	-	-	Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
RETAIN	Y	Y	Y	Y	Y
RETURN	?	?	?	Y	Y
RETURNING	Y, NR	Y, NR	Y, NR	-	-
RETURNING_VALUES	Y	Y	Y	Y	Y
RETURNS	Y	Y	Y	Y	Y
REVERSE	Y, NR	Y, NR	-	-	-
REVOKE	Y	Y	Y	Y	Y
RIGHT	Y	Y	Y	Y	Y
ROLE	Y, NR	Y, NR	Y, NR	Y	Y
ROLLBACK	Y	Y	Y	Y	Y
ROUND	Y, NR	Y, NR	-	-	-
ROW_COUNT	Y	Y	Y	Y	-
ROWS	Y	Y	Y	-	-
RPAD	Y, NR	Y, NR	-	-	-
RUNTIME	?	?	?	Y	Y
S					
SAVEPOINT	Y	Y	Y	Y	-
SCALAR_ARRAY	Y, NR	Y, NR	Y, NR	-	-
SCHEMA	[Y]	[Y]	[Y]	[Y]	[Y]
SECOND	Y	Y	Y	Y	Y
SELECT	Y	Y	Y	Y	Y
SENSITIVE	Y	Y	-	-	-
SEQUENCE	Y, NR	Y, NR	Y, NR	-	-
SET	Y	Y	Y	Y	Y
SHADOW	Y, NR	Y, NR	Y	Y	Y
SHARED	?	?	?	Y	Y
SHELL	?	?	?	Y	Y
SHOW	?	?	?	Y	Y
SIGN	Y, NR	Y, NR	-	-	-
SIMILAR	Y	-	-	-	-
SIN	Y, NR	Y, NR	-	-	-
SINGULAR	Y	Y	Y	Y	Y
SIZE	Y	Y	Y	Y	Y
SKIP	Y, NR	Y, NR	Y, NR	Y, NR	Y
SMALLINT	Y	Y	Y	Y	Y
SNAPSHOT	Y, NR	Y, NR	Y	Y	Y
SOME	Y	Y	Y	Y	Y
SORT	?	?	?	Y	Y
SOURCE	Y, NR	-	-	-	-

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
SPACE	Y, NR	Y, NR	-	-	-
SQL	Y, NR	Y, NR	Y, NR	Y	Y
SQRT	Y, NR	Y, NR	-	-	-
SQLCODE	Y	Y	Y	Y	Y
SQLERROR	Y	Y	Y	Y	Y
SQLWARNING	Y	Y	Y	Y	Y
SQLSTATE	Y	-	-	-	-
STABILITY	Y	Y	Y	Y	Y
START	Y	Y	-	-	-
STARTING	Y	Y	Y	Y	Y
STARTS	Y	Y	Y	Y	Y
STATEMENT	Y (con.)	Y (con.)	Y (con.)	Y	-
STATIC	?	?	?	Y	Y
STATISTICS	Y, NR	Y, NR	Y, NR	Y	Y
SUB_TYPE				Y	Y
SUBSTRING	Y, NR	Y, NR	Y, NR	Y, NR	Y
SUM	Y	Y	Y	Y	Y
SUSPEND	Y (con.)	Y (con.)	Y	Y	Y
T					
TABLE	Y	Y	Y	Y	Y
TAN	Y, NR	Y, NR	-	-	-
TANH	Y, NR	Y, NR	-	-	-
TEMPORARY	Y, NR	Y, NR	-	-	-
TERM	Y (con.)	Y (con.)	Y (con.)	Y	Y
TERMINATOR	Y (con.)	Y (con.)	Y (con.)	Y	Y
THEN	Y	Y	Y	Y	Y
TIME	Y	Y	Y	Y	Y
TIMESTAMP	Y	Y	Y	Y	Y
TO	Y	Y	Y	Y	Y
TRAILING	Y	Y	Y	-	-
TRANSACTION	Y	Y	Y	Y	Y
TRANSLATE	Y	Y	Y	Y	Y
TRANSLATION	Y	Y	Y	Y	Y
TRIGGER	Y	Y	Y	Y	Y
TRIM	Y	Y	Y	-	-
[TRUE]	-	-	-	-	-
TRUNC	Y, NR	Y, NR	-	-	-
TWO_PHASE	Y, NR	-	-	-	-
TYPE	N	N	N	Y	Y

Keyword	Firebird 2.5	2.1	2.0	1.5	1.0
U					
UNCOMMITTED	?	?	?	Y	Y
UNION	Y	Y	Y	Y	Y
UNIQUE	Y	Y	Y	Y	Y
[UNKNOWN]	-	-	-	-	-
UPDATE	Y	Y	Y	Y	Y
UPDATING	Y (con.)	Y (con.)	Y (con.)	Y	-
UPPER	Y	Y	Y	Y	Y
USER	Y	Y	Y	Y	Y
USING	Y	Y (con.)	Y (con.)	N	-
UUID_TO_CHAR	Y, NR	-	-	-	-
V					
VALUE	Y	Y	Y	Y	Y
VALUES	?	?	?	Y	Y
VARCHAR	Y	Y	Y	Y	Y
VARIABLE	Y	Y	Y	Y	Y
VARYING	Y	Y	Y	Y	Y
VERSION	?	?	?	Y	Y
VIEW	Y	Y	Y	Y	Y
W					
WAIT	Y, NR	Y	Y	Y	Y
WEEK	Y, NR	Y, NR	-	-	-
WEEKDAY	Y, NR	Y, NR	Y, NR	Y	Y
WHEN	Y	Y	Y	Y	Y
WHENEVER	Y	Y	Y	Y	Y
WHERE	Y	Y	Y	Y	Y
WHILE	Y	Y	Y	Y	Y
WITH	Y	Y	Y	Y	Y
WORK	Y	Y	Y	Y	Y
WRITE	Y	Y	Y	Y	Y
Y					
YEAR	Y	Y	Y	Y	Y
YEARDAY	Y, NR	Y, NR	Y, NR	Y	Y

APPENDIX

III

CONTEXT VARIABLES

Context variables are system-maintained variable values in the context of the current client connection and its activity.

About Context Variables

The context variables listed below in Table III.1, **Available context variables**, are available for use in DDL, DML and PSQL. Some are available only in PSQL and most are available only in Dialect 3 databases.

Table III.1 Available context variables

Context variable	Data type	Description	Availability
CURRENT_CONNECTION	INTEGER	System ID of the connection that is making this query	Firebird 1.5 onward, DSQL and PSQL
CURRENT_DATE	DATE	Current date on the host server's clock	All versions, all SQL environments
CURRENT_ROLE	VARCHAR(31)	The ROLE name under which the CURRENT_USER is logged in, returns empty string if the current log-in isn't using a role.	All versions, all SQL environments
CURRENT_TIME	TIME	Current time on the server's clock, expressed as seconds since midnight	All versions, all SQL environments, Dialect 3 only
CURRENT_TIMESTAMP	TIMESTAMP	Current date and time on the host server's clock to the nearest second	All versions, all SQL environments
CURRENT_TRANSACTION	INTEGER	System ID of the transaction in which this query is being requested	Firebird 1.5 onward, DSQL and PSQL

Context variable	Data type	Description	Availability
CURRENT_USER	VARCHAR(128)	User name that is communicating through this instance of the client library	All versions, all SQL environments
ROW_COUNT	INTEGER	Count of rows changed/deleted/added by a DML statement, available when the operation is complete	Firebird 1.5 onward, stored procedure language (PSQL) only
UPDATING	BOOLEAN	Returns true if an UPDATE statement is executing	Firebird 1.5 onward, trigger dialect of PSQL only
INSERTING	BOOLEAN	Returns true if an INSERT statement is executing	Firebird 1.5 onward, trigger dialect of PSQL only
DELETING	BOOLEAN	Returns true if a DELETE statement is executing	Firebird 1.5 onward, trigger dialect of PSQL only
SQLCODE	INTEGER	Returns the SQLCODE inside a WHEN exception block. For usage see Chapter 32, Handling and Events	Firebird 1.5 onward, procedure language (PSQL) only
GDSCODE	INTEGER NOTE: use the symbolic name to test.	Returns the GDSCODE inside a WHEN exception block. For usage see Chapter 32, Error Handling and Events	Firebird 1.5 onward, procedure language (PSQL) only
USER	VARCHAR(128)	User name that is communicating through this instance of the client library	All versions, all SQL environments. Available in Dialects 1 and 3.

APPENDIX

IV

FIREBIRD LIMITS

Most of the actual limits on a Firebird database are practical rather than defined by the software. For example, you can define up to 32,767 columns in a table, but why would you want to? Listed in Table IV.1 are a number of theoretical and practical limits applicable to the various versions of Firebird. Limits do change as on-disk structure versions progress, so make a point of studying release notes to track changes.

Table IV.1 Firebird Limits

Object	Item	Version	Limit	Comments
Identifiers	Maximum length	All	31 characters for almost all object types	Cannot use characters outside the range of US ASCII (ASCIIZ) unless double-quoted. Double-quoted identifiers cannot be used in dialect 1 databases.
		ODS < 11	27 characters for constraint names	As above
Dates	Earliest date	All	January 1, 100 A.D.	It is believed that the engine is susceptible to crashing if the system date of the server is set higher than the year 2039.
	Latest date	All	December 31, 9999 A.D	
Server	Maximum connected clients: TCP/IP	All	1,024	Work on a base maximum of about 150 concurrent Superserver clients for a normal interactive application on a server of low-to-medium specification, with low-to-moderate contention, before performance might make you consider upgrading. For Classic server, the numbers may be lower because each client consumes more resources.

Object	Item	Version	Limit	Comments
Server (cont.)	Maximum connected clients: NetBEUI	All	930 approx.	
	Maximum number of databases open in one transaction	All		The number of databases opened during a transaction started by <code>isc_start_multiple()</code> is limited only by available system resources. A transaction started by <code>isc_start_transaction()</code> limits concurrent database attachments to 16.
Database	Number of tables	All	32,767	
	Maximum size	All	7 TB approx.	Theoretical limit, approximate. There is no known record of a Firebird database as large as 7TB.
	Maximum file size	All		Depends on the file system. FAT32 and ext2 are 2GB. Older NTFS and some ext3 are usually 4GB. Many 64-bit file systems place no limit on the size of a shared-access file.
	Maximum number of files per database	All	Theoretically, 216 (65,536), including shadow files)	The limitation is more likely to be imposed by the operating system's limit on the number of files that can be opened simultaneously by one process. Some permit the limit to be raised.
	Maximum page_size	v.1.5+ (ODS 10.1+)	16,384 bytes	Other sizes are 1,024, 2,048, 4,096 (default), and 8192 bytes. V.2.0 and higher will not create databases with page_size smaller than 4,094 bytes.
	Maximum cache buffers	ODS 10.0	8,192 bytes	Practical limit depends on available RAM. The total size (cache pages * page_size on Superserver; cache pages * page_size * no. of concurrent users on Classic server) should never be more than half of the available RAM. Consider 10,000 pages as a practical limit and tweak backward or forward from there as performance dictates.
		ODS 11+	128,000 pages	
		ODS 10.x	65,536 pages	
Tables	Maximum versions per table	All	255	Firebird keeps account of up to 255 formats for each table. The format version steps up by 1 each time a metadata change is done. When any table reaches the limit, the whole database becomes unavailable—it must be backed up and then work resumed on the restored version.

Object	Item	Version	Limit	Comments
Tables (cont.)	Maximum row size: user tables	All	64 KB	Count bytes. BLOB and ARRAY columns each cost 8 bytes to store the ID; VARCHARs, byte length + 2; CHARs, byte-length; SMALLINT, 2; INTEGER, FLOAT, DATE and TIME, 4; BIGINT, DOUBLE PRECISION and TIMESTAMP, 8; NUMERIC and DECIMAL, 4 or 8, depending on precision.
	Maximum row size: system tables	All	128 KB	
	Maximum number of rows	ODS 11+	$2^{63} - 1$	Rows are enumerated with a 32-bit unsigned integer per table and a 32-bit row slot index. A table with long rows—either a lot of fields or very long fields—will store fewer records than a table with very short rows. All rows—including deleted ones—use up numbers; BLOBs and BLOB fragments stored on table data pages use up numbers, too.
		ODS 10.x	2^{32} rows, more or less, to a maximum table size of 30GB	
	Maximum number of columns	All		Depends on data types used (see Maximum row size).
	Maximum indexes per table	ODS 10.1+		256
		ODS 10.0		64
	Maximum size of external file			4GB on Windows NTFS, 2GB on Windows FAT32, Linux ext2 and ext3, and Solaris.
Indexes	Theoretical maximum size: applies to a single-column index, where the character set is single-byte and uses the default (binary) collation sequence.	ODS 11+	One quarter of the page_size	Count bytes, not characters. The practical maximum is reduced by compound indexing, multi-byte character sets, and complex collation sequences. A single-column index using 3-byte UNICODE_FSS characters, for example, can have a maximum of $(2048/3) = 682$ characters on a database with a page_size of 8192. Some ISO8859 collation sequences consume up to 4 bytes per character just for the sorting attributes.
		ODS 10.x	252 bytes	...
	Maximum number of segments	All	16	

Object	Item	Version	Limit	Comments
Queries	Maximum joined tables	All	Theoretically, 256	Other factors come to bear, such as the number of Boolean evaluations required by the joins. From the point of view of resources and performance, the largest practicable number of table references is probably around 16. Always test with realistic volumes and kinds of data.
	Maximum nesting level	Any	No theoretical limit	Deep nesting of subqueries is sub-optimal for performance. Performance and resource consumption will determine your practical limit, on a query-by-query basis.
	Maximum size of ORDER BY key-set data	All	32KB	...
PSQL modules	Maximum size of BLR	All	48KB	Stored procedure and trigger sources are compiled into BLR bytecode, which is more dense than the PSQL source. Still, if you hit this limit, try to break up your monumental procedure into a “master” procedure with callable chunks.
	Maximum number of events per module	Any	No limit	Practical limit is related to the length limit for BLR byte code (above).
	Levels of embedded calls: POSIX	All	1,000	...
	Levels of embedded calls: Windows	All	750	...
BLOBs	Maximum BLOB size	Any	Depends on page_size	For a 2KB page size, the BLOB size maximum is 512MB. For a 4KB page size, the BLOB size maximum is 4GB. For a 8KB page size, it is 32GB, and for a 16KB page size, it is 256GB.
	Maximum segment size	Any	Theoretically, 64KB	BLOBs are stored in segments. However, in DSQL, it is not essential to define a non-default segment size, since client settings cause BLOB data to be segmented for transport according to network packet size. Server economics determine the actual size of segments in storage.



SYSTEM TABLES AND VIEWS

When you create a database, Firebird begins by building its own tables in which to store the metadata for all database objects—not just your user-defined objects but also its own internal objects, whose identifiers are prefixed with ‘RDB\$’. All versions have the metadata tables.

From version 2.1 forward, databases also contain the definitions for the monitoring tables, prefixed with the characters “MON\$”. See **Monitoring Tables** on page 942.

Metadata Tables

The descriptions in this section are intended to assist with designing queries that you can use to help you understand and administer your databases.



There are DDL statements (CREATE, ALTER, etc.) for changing metadata. It is not recommended at all to use DML statements to update system tables. The risk of corruption from doing so is extreme.

In a future release—probably v.3.0—the system tables will be made read-only.

Following the metadata definitions for the system tables, you will find DDL listings for a number of views over the system that you might find useful.

The following abbreviations are used in the tables:

- IDX: Indexed
- UQ: Unique

Where there are compound indexes, numbers are given to indicate the precedence of the index segments.

RDB\$BACKUP_HISTORYStores history of *nBackup* backups (from v.2.0 onward).**Table V.1** RDB\$BACKUP_HISTORY

rdp\$backup_history.column	Data Type	IDX	UQ	Description
RDB\$BACKUP_ID	INTEGER	Y(1)	Y	Unique ID of one backup
RDB\$BACKUP_LEVEL	INTEGER	Y(2)	(compound)	Indicates which backup level this backup file applies to
RDB\$FILE_NAME	VARCHAR(255)	—	—	Name of file where backup is stored. Character set=NONE
RDB\$GUID	VARCHAR(38)	—	—	?. Character set=NONE
RDB\$SCN	INTEGER	—	—	Scan identifier
RDB\$TIMESTAMP	TIMESTAMP	—	—	Timestamp of backup start

RDB\$CHARACTER_SETS

Stores records for character sets available to the database

Table V.2 RDB\$CHARACTER_SETS.

rdp\$character_sets.column	Data Type	IDX	UQ	Description
RDB\$CHARACTER_SET_NAME	CHAR(31)	Y	Y	Name of a character set known to Firebird.
RDB\$FORM_OF_USE	CHAR(31)	—	—	Not used
RDB\$NUMBER_OF_CHARACTERS	INTEGER	—	—	Number of characters in the set (not used for the available character sets).
RDB\$DEFAULT_COLLATE_NAME	CHAR(31)	—	—	Name of the binary collation sequence for the character set. It is always the same as the character set name.
RDB\$CHARACTER_SET_ID	SMALLINT	Y	Y	Unique identifier for this character set, wherever it is used.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	Will be 1 if the character set is defined by the system at database create; 0 for a user-defined character set.
RDB\$DESCRIPTION	BLOB TEXT	—	—	For storing documentation.

rdb\$character_sets.column	Data Type	IDX	UQ	Description
RDB\$FUNCTION_NAME	CHAR(31)	—	—	Not used, but may become available for user-defined character sets that are accessed via an external function.
RDB\$BYTES_PER_CHARACTER	SMALLINT	—	—	Size of characters in the set, in bytes. For example, UNICODE_FSS uses 3 bytes per character.

RDB\$CHECK_CONSTRAINTS

Cross-references names and triggers for CHECK and NOT NULL constraints.

Table V.3 RDB\$CHECK_CONSTRAINTS.

rdb\$check_constraints.column	Data Type	IDX	UQ	Description
RDB\$CONSTRAINT_NAME	CHAR(31)	Y	—	Name of a constraint.
RDB\$TRIGGER_NAME	CHAR(31)	Y	—	For a CHECK constraint, this is the name of the trigger that enforces the constraint. For a NOT NULL constraint, this is the name of the column to which the constraint applies—the table name can be found through the constraint name.

RDB\$COLLATIONS

Stores definitions of collation sequences for character sets

Table V.4 RDB\$COLLATIONS.

rdb\$collations.column	Data Type	IDX	UQ	Description
RDB\$COLLATION_NAME	VARCHAR(31)	Y	Y	Name of the collation sequence
RDB\$COLLATION_ID	SMALLINT	Y(1)	Y(1)	With the character set ID, unique collation identifier
RDB\$BASE_COLLATION_NAME	VARCHAR(31)			Name of collation that is extended by this collation
RDB\$CHARACTER_SET_ID	SMALLINT	Y(2)	Y(2)	With the collation ID, unique collation identifier
RDB\$COLLATION_ATTRIBUTES	SMALLINT	—	—	Not used externally
RDB\$SPECIFIC_ATTRIBUTES	TEXT BLOB			Holds attributes added to or modified from base collation. Character set UNICODE_FSS

rdbscollations.column	Data Type	IDX	UQ	Description
RDB\$SYSTEM_FLAG	SMALLINT	—	—	User-defined=0; system-defined=1 or higher
RDB\$DESCRIPTION	BLOB TEXT	—	—	For storing documentation
RDB\$FUNCTION_NAME	CHAR(31)	—	—	Not currently used

RDB\$DATABASE

A single-record table containing basic information about the database.

Table V.5 RDB\$DATABASE

rdbsdatabase.column	Data Type	IDX	UQ	Description
RDB\$DESCRIPTION	BLOB TEXT	—	—	Comment text included with the CREATE DATABASE/CREATE SCHEMA statement is supposed to be written here. It doesn't happen. However, you can add any amount of text to it by way of documentation and it will survive a gbak and restore.
RDB\$RELATION_ID	SMALLINT	—	—	A number that steps up by 1 each time a new table or view is added to the database.
RDB\$SECURITY_CLASS	CHAR(31)	—	—	Can refer to a security class defined in RDB\$SECURITY_CLASSES, to apply database-wide access control limits.
RDB\$CHARACTER_SET_NAME	CHAR(31)	—	—	Default character set of the database. NULL if the character set is NONE.

RDB\$DEPENDENCIES

Stores dependencies between database objects.

Table V.6 RDB\$DEPENDENCIES

rdbsdependencies.column	Data Type	IDX	UQ	Description
RDB\$DEPENDENT_NAME	CHAR(31)	Y	—	Names the view, procedure, trigger, or computed column tracked by this record.
RDB\$DEPENDED_ON_NAME	CHAR(31)	Y	—	The table that the view, procedure, trigger, or computed column refers to.
RDB\$FIELD_NAME	VARCHAR(31)	—	—	Names one column in the depended-on table that the view, procedure, trigger, or computed column refers to.
RDB\$DEPENDENT_TYPE	SMALLINT	—	—	Identifies the object type (view, procedure, trigger, computed column). The number comes from the table RDB\$TYPES—objects are enumerated where RDB\$FIELD_NAME = 'RDB\$OBJECT_TYPE'.
RDB\$DEPENDED_ON_TYPE	SMALLINT	—	—	Identifies the type of the object depended on (same object paradigm as for RDB\$DEPENDENT_TYPE).

RDB\$EXCEPTIONS

Stores custom exceptions, i.e., created with CREATE EXCEPTION.

Table V.7 RDB\$EXCEPTIONS

rdbsexceptions.column	Data Type	IDX	UQ	Description
RDB\$EXCEPTION_NAME	CHAR(31)	Y	Y	Name of the custom exception
RDB\$EXCEPTION_NUMBER	INTEGER	Y	Y	System-assigned unique exception number
RDB\$MESSAGE	VARCHAR(78)	—	—	Custom message text
RDB\$DESCRIPTION	BLOB TEXT	—	—	Can be used for documentation
RDB\$SYSTEM_FLAG	SMALLINT	—	—	User-defined=0; system-defined=1 or higher

RDB\$FIELD_DIMENSIONS

Stores information about dimensions of array columns.

Table V.8 RDB\$FIELD_DIMENSIONS

rdbsfield_dimensions.column	Data Type	IDX	UQ	Description
RDB\$FIELD_NAME	CHAR(31)	Y	—	Name of the array column. It must be a RDB\$FIELD_NAME in the table RDB\$FIELDS.
RDB\$DIMENSION	SMALLINT	—	—	Identifies one dimension in the array column. The first has the identifier 0.
RDB\$LOWER_BOUND	INTEGER	—	—	Lower bound of this dimension.
RDB\$UPPER_BOUND	INTEGER	—	—	Upper bound of this dimension.

RDB\$FIELDS

Stores definitions of domains and of column names for tables and views. Each row for a non-domain column has a corresponding row in RDB\$RELATION_FIELDS. In reality, every instance of RDB\$FIELDS is a domain. You can, for example, do this:

```
CREATE TABLE ATABLE (  
    EXAMPLE VARCHAR(10) CHARACTER SET ISO8859_1);  
COMMIT;  
SELECT RDB$FIELD_SOURCE FROM RDB$RELATION_FIELDS  
    WHERE RDB$RELATION_NAME = 'ATABLE'  
    AND RDB$FIELD_NAME = 'EXAMPLE';  
RDB$FIELD_SOURCE  
=====
```

```
SQL$99  
/* */  
ALTER TABLE ATABLE  
    ADD EXAMPLE2 SQL$99;  
COMMIT;
```

The new column is added, having the same attributes as the original.

Table V.9 RDB\$FIELDS

rdb\$fields.column	Data Type	IDX	UQ	Description
RDB\$FIELD_NAME	CHAR(31)	Y	Y	For domains, it is the domain name. For table and view columns, it is the internal, database-unique field name, linking to RDB\$FIELD_SOURCE in RDB\$RELATION_FIELDS. NB Firebird creates a domain in this table for every column definition that is not derived from a user-defined domain.
RDB\$QUERY_NAME	CHAR(31)			Not used in Firebird.
RDB\$VALIDATION_BLR	BLOB BLR			Not used in Firebird.
RDB\$VALIDATION_SOURCE	BLOB TEXT			Not used in Firebird.
RDB\$COMPUTED_BLR	BLOB BLR	—	—	Binary language representation of the SQL expression that Firebird evaluates when a COMPUTED BY column is accessed.
RDB\$COMPUTED_SOURCE	BLOB TEXT	—	—	Original source text of the expression that defines a COMPUTED BY column.
RDB\$DEFAULT_VALUE	BLOB BLR	—	—	Default rule for the default value, in binary language representation.
RDB\$DEFAULT_SOURCE	BLOB TEXT	—	—	Ditto; in original form.
RDB\$FIELD_LENGTH	SMALLINT	—	—	Length of the column in bytes. Float, date, time, and integer are 4 bytes. Double precision, BigInt, timestamp, and blob_id are 8 bytes.
RDB\$FIELD_SCALE	SMALLINT	—	—	Negative number representing the scale of a NUMERIC or DECIMAL column.
RDB\$FIELD_TYPE	SMALLINT	—	—	Number code of the data type defined for the column: 7=smallint, 8=integer, 12=date, 13=time, 14=char, 16=bigint, 27=double precision, 35=timestamp, 37=varchar, 261=blob. Codes for numeric and decimal are the same as that of the integer-type that is used to store it.

rdb\$fields.column	Data Type	IDX	UQ	Description
RDB\$FIELD_SUB_TYPE	SMALLINT	—	—	BLOB subtype, namely 0=untyped, 1=text, 2=BLR (binary language representation), 3=ACL (access control list), 5=encoded table metadata, 6=description of a cross-database transaction that didn't complete normally.
RDB\$MISSING_VALUE	BLOB BLR			Not used in Firebird.
RDB\$MISSING_SOURCE	BLOB TEXT			Not used in Firebird.
RDB\$DESCRIPTION	BLOB TEXT	—	—	Available to use for documentation.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	1=system table; anything else, user-defined table.
RDB\$QUERY_HEADER	BLOB TEXT			Not used in Firebird.
RDB\$SEGMENT_LENGTH	SMALLINT	—	—	For BLOB columns, a suggested length for BLOB buffers. Not relevant in Firebird.
RDB\$EDIT_STRING	VARCHAR(125)			Not used in Firebird.
RDB\$EXTERNAL_LENGTH	SMALLINT	—	—	Length of the field as it is in an external table. Always 0 for regular tables.
RDB\$EXTERNAL_SCALE	SMALLINT	—	—	Scale factor of an integer field in an external table; represents the power of 10 by which the integer is multiplied.
RDB\$EXTERNAL_TYPE	SMALLINT	—	—	Data type of the field as it is in an external table. Data types are the same as for regular tables, but include 40=null-terminated text (CSTRING).
RDB\$DIMENSIONS	SMALLINT	—	—	Number of dimensions defined, if column is an array type. Always 0 for non-array columns.
RDB\$NULL_FLAG	SMALLINT	—	—	Indicates whether column is nullable (empty) or non-nullable (1).
RDB\$CHARACTER_LENGTH	SMALLINT	—	—	Length of a CHAR or VARCHAR column, in characters (not bytes).
RDB\$COLLATION_ID	SMALLINT	—	—	Number ID of the collation sequence (if defined) for a character column or domain.

rdb\$fields.column	Data Type	IDX	UQ	Description
RDB\$CHARACTER_SET_ID	SMALLINT	—	—	Number ID of the character set for character columns, BLOB columns, or domains. Links to RDB\$CHARACTER_SET_ID column in RDB\$CHARACTER_SETS.
RDB\$FIELD_PRECISION	SMALLINT	—	—	Indicates the number of digits of precision available to the data type of the column.

RDB\$FILES

Stores volume details of database secondary files and shadow files.

Table V.10 RDB\$FILES

rdb\$files.column	Data Type	IDX	UQ	Description
RDB\$FILE_NAME	VARCHAR(253)	—	—	Name of a database secondary file (volume) in a multi-volume database, or a shadow file.
RDB\$FILE_SEQUENCE	SMALLINT	—	—	Sequence in the volume order of database secondary files, or sequence in the shadow file set.
RDB\$FILE_START	INTEGER	—	—	Starting page number.
RDB\$FILE_LENGTH	INTEGER	—	—	File length, in database pages.
RDB\$FILE_FLAGS	SMALLINT	—	—	Internal use.
RDB\$SHADOW_NUMBER	SMALLINT	—	—	Shadow set number. Required to identify the file as a member of a shadow set. If it is null or 0, Firebird assumes the file is a secondary database volume.

RDB\$FILTERS

Stores and keeps track of information about BLOB filters.

Table V.11 RDB\$FILTERS

rdb\$filters.column	Data Type	IDX	UQ	Description
RDB\$FUNCTION_NAME	CHAR(31)	—	—	Unique name of the BLOB filter
RDB\$DESCRIPTION	BLOB TEXT	—	—	User-written documentation about the BLOB filter and the two subtypes it is meant to operate on
RDB\$MODULE_NAME	VARCHAR(253)	—	—	The name of the dynamic library/shared object where the BLOB filter code is located

rdb\$filters.column	Data Type	IDX	UQ	Description
RDB\$ENTRYPOINT	CHAR(31)	—	—	The entry point in the filter library for this BLOB filter
RDB\$INPUT_SUB_TYPE	SMALLINT	Y(1)	Y(1)	The BLOB subtype of the data to be transformed.
RDB\$OUTPUT_SUB_TYPE	SMALLINT	Y(2)	Y(2)	The BLOB subtype that the input data is to be transformed to.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	Externally defined (i.e., user-defined=0, internally defined=1 or greater)

RDB\$FORMATS

Keeps account of the number of metadata changes performed on tables. Each time a table or view gets a change, it gets a new format number. The purpose is to allow applications to access a changed table without the need to recompile. When the format number of any table reaches 255, the whole database becomes inaccessible for querying. It is then necessary to back up the database, restore it, and resume work in the newly built database.

Table V.12 RDB\$FORMATS

rdb\$formats.column	Data Type	IDX	UQ	Description
RDB\$RELATION_ID	SMALLINT	Y(1)	Y(1)	Number ID of a table or view in RDB\$RELATIONS.
RDB\$FORMAT	SMALLINT	Y(2)	Y(2)	Identifier of the table format. There can be up to 255 such rows for any particular table
RDB\$DESCRIPTOR	BLOB FORMAT	—	—	BLOB listing the columns and data attributes at the time the format record was created.

RDB\$FUNCTION_ARGUMENTS

Stores the attributes of arguments (parameters) of external functions.

Table V.13 RDB\$FUNCTION_ARGUMENTS

rdb\$functions_arguments.column	Data Type	IDX	UQ	Description
RDB\$FUNCTION_NAME	CHAR(31)	Y	—	Unique name of the external function, matching a function name in RDB\$FUNCTIONS.
RDB\$ARGUMENT_POSITION	SMALLINT	—	—	Position of the argument in the argument list: 1=first, 2=second, etc.
RDB\$MECHANISM	SMALLINT	—	—	Whether the argument is passed by value (0), by reference (1), by descriptor (2), or by BLOB descriptor (3).

rdb\$functions_arguments.column	Data Type	IDX	UQ	Description
RDB\$FIELD_TYPE	SMALLINT	—	—	Number code of the data type defined for the column: 7=smallint, 8=integer, 12=date, 13=time, 14=char, 16=bigint, 27=double precision, 35=timestamp, 37=varchar, 40=cstring (null-terminated string), 261=blob.
RDB\$FIELD_SCALE	SMALLINT	—	—	Scale of an integer or fixed numeric argument.
RDB\$FIELD_LENGTH	SMALLINT	—	—	Length of the argument in bytes. For lengths of non-character types, refer to RDB\$FIELDS.RDB\$FIELD_LENGTH .
RDB\$FIELD_SUB_TYPE	SMALLINT	—	—	For BLOB arguments, BLOB subtype.
RDB\$CHARACTER_SET_ID	SMALLINT	—	—	Numeric ID for the character set, for a character argument, if applicable.
RDB\$FIELD_PRECISION	SMALLINT	—	—	Digits of precision available to the data type of the argument.
RDB\$CHARACTER_LENGTH	SMALLINT	—	—	Length of a CHAR or VARCHAR argument, in characters (not bytes).

RDB\$FUNCTIONS

Stores information about external functions.

Table V.14 RDB\$FUNCTIONS

rdB\$functions.column	Data Type	IDX	UQ	Description
RDB\$FUNCTION_NAME	CHAR(31)	Y	Y	Unique name of an external function.
RDB\$FUNCTION_TYPE	SMALLINT			Not currently used.
RDB\$QUERY_NAME	CHAR(31)	—	—	This is meant to be an alternative name for the function, for use in isql queries. It doesn't work.
RDB\$DESCRIPTION	BLOB TEXT	—	—	Available for documentation.
RDB\$MODULE_NAME	VARCHAR(253)	—	—	Name of the dynamic library/shared object where the code for the function is located.
RDB\$ENTRYPOINT	CHAR(31)	—	—	Name of the entry point in the library where this function is to be found.
RDB\$RETURN_ARGUMENT	SMALLINT	—	—	Ordinal position of the return argument in the parameter list, relative to the input arguments.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	Externally defined (user-defined)=1; system-defined=0.

RDB\$GENERATORS

Stores names and IDs of generators and sequences.

Table V.15 RDB\$GENERATORS

rdB\$generators.column	Data Type	IDX	UQ	Description
RDB\$GENERATOR_NAME	CHAR(31)	Y	Y	Name of generator.
RDB\$GENERATOR_ID	BIGINT	—	—	System-assigned unique ID for the generator.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	0=user-defined; 1 or greater=system-defined. Firebird uses a number of generators internally.
RDB\$DESCRIPTION	BLOB TEXT	—	—	For storing documentation.

RDB\$INDEX_SEGMENTS

Stores the segments and positions of multi-segment indexes.

Table V.16 RDB\$INDEX_SEGMENTS

rdp\$index_segments.column	Data Type	IDX	UQ	Description
RDB\$INDEX_NAME	CHAR(31)	Y	—	Name of the index. Must be kept consistent with the corresponding master record in RDB\$INDICES.
RDB\$FIELD_NAME	CHAR(31)	—	—	Name of a key column in the index. Matches the RDB\$FIELD_NAME of the database column, in RDB\$RELATION_FIELDS.
RDB\$FIELD_POSITION	SMALLINT	—	—	Ordinal position of the column in the index (left to right).
RDB\$STATISTICS	DOUBLE PRECISION	—	—	Stores per-segment selectivity statistics (v.2+)

RDB\$INDICES

Stores definitions of all indexes.

Table V.17 RDB\$INDICES

rdp\$indices.column	Data Type	IDX	UQ	Description
RDB\$INDEX_NAME	CHAR(31)	Y	Y	Unique name of the index.
RDB\$RELATION_NAME	CHAR(31)	Y	—	Name of the table the index applies to. Matches a RDB\$RELATION_NAME in a record in RDB\$RELATIONS.
RDB\$INDEX_ID	SMALLINT	—	—	Internal number ID of the index. Writing to this column from an application will break the index.
RDB\$UNIQUE_FLAG	SMALLINT	—	—	Indicates whether the index is unique (1=unique, 0=not unique).
RDB\$DESCRIPTION	BLOB TEXT	—	—	Available for documentation.
RDB\$SEGMENT_COUNT	SMALLINT	—	—	Number of segments (columns) in the index.
RDB\$INDEX_INACTIVE	SMALLINT	—	—	Indicates whether the index is currently inactive (1=inactive, 0=active).
RDB\$INDEX_TYPE	SMALLINT	—	—	Distinguish regular indexes (0) from expression indexes (1). Not used in versions older than v.2.0.
RDB\$FOREIGN_KEY	VARCHAR(31)	Y	—	Name of the associated foreign key constraint, if any.

rdbs\$indices.column	Data Type	IDX	UQ	Description
RDB\$SYSTEM_FLAG	SMALLINT	—	—	Indicates whether the index is system-defined (1 or greater) or user-defined (0).
RDB\$EXPRESSION_BLR	BLOB BLR	—	—	Binary language representation of an expression, used for runtime evaluation for expression indexes. Not used in versions older than v.2.0.
RDB\$EXPRESSION_SOURCE	BLOB TEXT	—	—	Source of an expression used for an expression index. Not used in versions older than v.2.0.
RDB\$STATISTICS	DOUBLE PRECISION	—	—	Stores the latest selectivity of the index, as calculated at start-up or by SET STATISTICS.

RDB\$LOG_FILES

An obsolete system table.

RDB\$PAGES

Stores information about database pages.

Table V.18 RDB\$PAGES

rdbs\$pages.column	Data Type	IDX	UQ	Description
RDB\$PAGE_NUMBER	INTEGER	—	—	Unique number of a database page that has been physically allocated.
RDB\$RELATION_ID	SMALLINT	—	—	ID of table whose data are stored on the page
RDB\$PAGE_SEQUENCE	INTEGER	—	—	Sequence number of this page, relative to other pages allocated for this table.
RDB\$PAGE_TYPE	SMALLINT	—	—	Identifies the type of data stored on the page (table data, index, etc.)

RDB\$PROCEDURE_PARAMETERS

Stores the parameters for stored procedures.

Table V.19 RDB\$PROCEDURE_PARAMETERS

rdbs\$procedure_parameters.column	Data Type	IDX	UQ	Description
RDB\$PARAMETER_NAME	CHAR(31)	Y(2)	Y(2)	Name of the parameter.
RDB\$PROCEDURE_NAME	CHAR(31)	Y(1)	Y(1)	Name of the procedure
RDB\$PARAMETER_NUMBER	SMALLINT	—	—	Sequence number of parameter.

rdb\$procedure_parameters.column	Data Type	IDX	UQ	Description
RDB\$PARAMETER_TYPE	SMALLINT	—	—	Indicates whether parameter is input (0) or output (1).
RDB\$FIELD_SOURCE	CHAR(31)	—	—	System-generated unique column name
RDB\$DESCRIPTION	BLOB TEXT	—	—	Available for documentation.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	Indicates whether the parameter is system-defined (1 or greater) or user-defined (0).
RDB\$COLLATION_ID	SMALLINT	—	—	Character set UNICODE_FSS
RDB\$DEFAULT_SOURCE	BLOB TEXT	—	—	
RDB\$DEFAULT_VALUE	BLOB BLR	—	—	
RDB\$NULL_FLAG	SMALLINT			Indicates whether the parameter is non-nullable (1 or greater) or nullable (0).
RDB\$PARAMETER_MECHANISM	SMALLINT			Character set UNICODE_FSS
RDB\$FIELD_NAME	NCHAR(31)			
RDB\$RELATION_NAME	NCHAR(31)			

RDB\$PROCEDURES

Stores definitions of stored procedures.

Table V.20 RDB\$PROCEDURES

rdb\$procedures.column	Data Type	IDX	UQ	Description
RDB\$PROCEDURE_NAME	CHAR(31)	Y	Y	Name of procedure.
RDB\$PROCEDURE_ID	SMALLINT	Y	—	System-defined unique ID of procedure.
RDB\$PROCEDURE_INPUTS	SMALLINT	—	—	Indicates whether there are input parameters (1) or not (0)
RDB\$PROCEDURE_OUTPUTS	SMALLINT	—	—	Indicates whether there are output parameters (1) or not (0)
RDB\$DESCRIPTION	BLOB TEXT	—	—	Available for documentation.
RDB\$PROCEDURE_SOURCE	BLOB TEXT	—	—	Source code of the procedure
RDB\$PROCEDURE_BLR	BLOB BLR	—	—	Binary language representation (BLR) of the procedure code.
RDB\$SECURITY_CLASS	CHAR(31)	—	—	Can refer to a security class defined in RDB\$SECURITY_CLASSES, to apply access control limits.
RDB\$OWNER_NAME	VARCHAR(31)	—	—	User name of the procedure's owner.

rdp\$procedures.column	Data Type	IDX	UQ	Description
RDB\$RUNTIME	BLOB SUMMARY	—	—	Description of metadata of procedure, internal use for optimization
RDB\$SYSTEM_FLAG	SMALLINT	—	—	User-defined (0) or system-defined (1 or greater)
RDB\$PROCEDURE_TYPE	SMALLINT			
RDB\$DEBUG_INFO	BLOB BINARY			
RDB\$VALID_BLR	SMALLINT			

RDB\$REF_CONSTRAINTS

Stores actions for referential constraints.

Table V.21 RDB\$REF_CONSTRAINTS

rdp\$ref_constraints.column	Data Type	IDX	UQ	Description
RDB\$CONSTRAINT_NAME	CHAR(31)	Y	Y	Name of a referential constraint.
RDB\$CONST_NAME_UQ	CHAR(31)	—	—	Name of the primary key or unique constraint referred to in the REFERENCES clause of this constraint.
RDB\$MATCH_OPTION	CHAR(7)	—	—	Current value is FULL in all cases; reserved for future use.
RDB\$UPDATE_RULE	CHAR(11)	—	—	Referential integrity action applicable to this foreign key when the primary key is updated: NO ACTION CASCADE SET NULL SET DEFAULT.
RDB\$DELETE_RULE	CHAR(11)	—	—	Referential integrity action applicable to this foreign key when the primary key is deleted. Rule options as defined in the column RDB\$UPDATE_RULE

RDB\$RELATION_CONSTRAINTS

Stores information about table-level integrity constraints.

Table V.22 RDB\$RELATION_CONSTRAINTS

rdp\$relation_constraints.column	Data Type	IDX	UQ	Description
RDB\$CONSTRAINT_NAME	CHAR(31)	Y	Y	Name of a table-level constraint.
RDB\$CONSTRAINT_TYPE	CHAR(11)	Y(2)	—	Primary key/unique/foreign key/pcheck/not null

rdB\$relation_constraints.column	Data Type	IDX	UQ	Description
RDB\$RELATION_NAME	CHAR(31)	Y(1)	—	Name of the table this constraint applies to.
RDB\$DEFERRABLE	CHAR(3)	—	—	Currently NO in all cases; reserved for future implementation of deferred constraint.
RDB\$INITIALLY_DEFERRED	CHAR(3)	—	—	Ditto.
RDB\$INDEX_NAME	CHAR(31)	Y	—	Name of the index that enforces the constraint (applicable if constraint is PRIMARY KEY, UNIQUE, or FOREIGN KEY).

RDB\$RELATION_FIELDS

Stores the definitions of columns.

Table V.23 RDB\$RELATION_FIELDS

rdB\$relation_fields.column	Data Type	IDX	UQ	Description
RDB\$FIELD_NAME	CHAR(31)	Y(1)	Y(1)	Name of the column, unique in table or view.
RDB\$RELATION_NAME	CHAR(31)	Y(2)	Y(2)	Name of table or view.
		Y	—	(Another index)
RDB\$FIELD_SOURCE	CHAR(31)	Y	—	The system-generated name (SQL\$nn) for this column, correlated in RDB\$FIELDS. If the column is based on a domain, the two correlated RDB\$FIELD_SOURCE columns store the domain name.
RDB\$QUERY_NAME	CHAR(31)			Not used currently.
RDB\$BASE_FIELD	CHAR(31)	—	—	For a view only, the column name from the base table. The base table is identified by an internal ID in the column RDB\$VIEW_CONTEXT.
RDB\$EDIT_STRING	VARCHAR(125)			Not used in Firebird.
RDB\$FIELD_POSITION	SMALLINT	—	—	Position of column in table or view in relation to the other columns. Note that for tables, you can alter this using ALTER TABLE ALTER COLUMN POSITION n, where n is the new field_position.
RDB\$QUERY_HEADER	BLOB TEXT			Not used in Firebird.
RDB\$UPDATE_FLAG	SMALLINT			Not used in Firebird.

rdbs\$relation_fields.column	Data Type	IDX	UQ	Description
RDB\$FIELD_ID	SMALLINT	—	—	Transient number ID, used internally. It changes after backup and restore, so don't rely on it for queries in applications and don't change it.
RDB\$VIEW_CONTEXT	SMALLINT	—	—	For a view column, internal number ID for the base table where the field comes from. Don't modify this column.
RDB\$DESCRIPTION	BLOB TEXT	—	—	Can store documentation about the column.
RDB\$DEFAULT_VALUE	BLOB BLR	—	—	Binary language representation of the DEFAULT clause, if any.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	User-defined (0) or system-defined (1 or greater).
RDB\$SECURITY_CLASS	CHAR(31)	—	—	Can refer to a security class defined in RDB\$SECURITY_CLASSES, to apply access control limits to all users of this column.
RDB\$COMPLEX_NAME	CHAR(31)	—	—	Reserved for future implementation.
RDB\$NULL_FLAG	SMALLINT	—	—	Indicates whether column is nullable (empty) or non-nullable (1).
RDB\$DEFAULT_SOURCE	BLOB TEXT	—	—	Original source text of the DEFAULT clause, if any.
RDB\$COLLATION_ID	SMALLINT	—	—	ID of non-default collation sequence (if any) for column.

RDB\$RELATIONS

Stores tables and view definition header information.

Table V.24 RDB\$RELATIONS

rdbs\$relations.column	Data Type	IDX	UQ	Description
RDB\$VIEW_BLR	BLOB BLR	—	—	Binary language representation of the query specification for a view; null on tables.
RDB\$VIEW_SOURCE	BLOB TEXT	—	—	The query specification for a view.
RDB\$DESCRIPTION	BLOB TEXT	—	—	Optional documentation.
RDB\$RELATION_ID	SMALLINT	Y	—	Internal number ID for the table. Don't modify this column.

rdb\$relations.column	Data Type	IDX	UQ	Description
RDB\$SYSTEM_FLAG	SMALLINT	—	—	Indicates whether the table is user-created (0) or system-created (1 or greater). Don't modify this flag on user-defined or system tables.
RDB\$DBKEY_LENGTH	SMALLINT	—	—	For views, aggregated length of the DB_KEY. It is 8 bytes for tables. For views, it is 8 * number of tables the view definition refers to. Don't modify this column.
RDB\$FORMAT	SMALLINT	—	—	Internal use—don't modify.
RDB\$FIELD_ID	SMALLINT	—	—	Internal use—don't modify. It stores the number of columns in the table or view.
RDB\$RELATION_NAME	CHAR(31)	Y	Y	Name of the table or view.
RDB\$SECURITY_CLASS	CHAR(31)	—	—	Can refer to a security class defined in RDB\$SECURITY_CLASSES, to apply access control limits to all users of this table.
RDB\$EXTERNAL_FILE	VARCHAR(253)	—	—	Full path to the external data file, if any.
RDB\$RUNTIME	BLOB SUMMARY	—	—	Description of table's metadata. Internal use for optimization.
RDB\$EXTERNAL_DESCRIPTION	BLOB EFD	—	—	BLOB of sub_type external_file_description, a text BLOB type that can be used for documentation.
RDB\$OWNER_NAME	VARCHAR(31)	—	—	User name of table's or view's owner (creator), for SQL security purposes.
RDB\$DEFAULT_CLASS	CHAR(31)	—	—	Default security class, applied when new columns are added to a table.
RDB\$FLAGS	SMALLINT	—	—	Internal flags.
RDB\$RELATION_TYPE	SMALLINT			(V.2.5+) To distinguish GTTs from regular relations?

RDB\$ROLES

Stores role definitions.

Table V.25 RDB\$ROLES

rdbsroles.column	Data Type	IDX	UQ	Description
RDB\$ROLE_NAME	VARCHAR(31)	Y	Y	Role name.
RDB\$OWNER_NAME	VARCHAR(31)	—	—	User name of role owner
RDB\$DESCRIPTION				
RDB\$SYSTEM_FLAG				

RDB\$SECURITY_CLASSES

Stores and tracks access control lists.

Table V.26 RDB\$SECURITY_CLASSESRDB\$TRANSACTIONS

rdbssecurity_class.column	Data Type	IDX	UQ	Description
RDB\$SECURITY_CLASS	CHAR(31)	Y	Y	Name of security class. This name must stay consistent in all places where it is used (RDB\$DATABASE, RDB\$RELATIONS, RDB\$RELATION_FIELDS).
RDB\$ACL	BLOB ACL	—	—	Access control list associated with the security class. It enumerates users and their permissions.
RDB\$DESCRIPTION	BLOB TEXT	—	—	Documentation of the security class here defined.

RDB\$TRANSACTIONS

Tracks cross-database transactions.

Table V.27 RDB\$TRANSACTIONS

rdbstransactions.column	Data Type	IDX	UQ	Description
RDB\$TRANSACTION_ID	INTEGER	Y	Y	Unique ID of the transaction being tracked
RDB\$TRANSACTION_STATE	SMALLINT	—	—	State of the transaction: limbo(0), committed(1), rolled back (2)

rdb\$transactions.column	Data Type	IDX	UQ	Description
RDB\$TIMESTAMP	TIMESTAM P			For future implementation.
RDB\$TRANSACTION_DESCRIPTION	BLOB TD	—	—	BLOB of sub_type transaction_description, describing a prepared multi-database transaction, available in case a lost connection cannot be restored

RDB\$TRIGGER_MESSAGES

Stores trigger message definitions (system use).

Table V.28 RDB\$TRIGGER_MESSAGES

rdb\$trigger_messages.column	Data Type	IDX	UQ	Description
RDB\$TRIGGER_NAME	CHAR(31)	Y	—	Name of the trigger the message is associated with.
RDB\$MESSAGE_NUMBER	SMALLINT	—	—	Message number (1 to a maximum of 32767)
RDB\$MESSAGE	VARCHAR(78)	—	—	Trigger message text

RDB\$TRIGGERS

Stores definitions of all triggers.

Table V.29 RDB\$TRIGGERS

rdb\$triggers.column	Data Type	IDX	UQ	Description
RDB\$TRIGGER_NAME	CHAR(31)	Y	Y	Name of the trigger.
RDB\$RELATION_NAME	CHAR(31)	Y	—	Name of the table or view that the trigger is for.
RDB\$TRIGGER_SEQUENCE	SMALLINT	—	—	Sequence (position) of trigger. Zero usually means no sequence was defined.
RDB\$TRIGGER_SOURCE	BLOB TEXT	—	—	Stores the PSQL source code for the trigger.
RDB\$TRIGGER_BLR	BLOB BLR	—	—	Stores the binary language representation of the trigger.
RDB\$TRIGGER_INACTIVE	SMALLINT	—	—	Whether the trigger is currently inactive (1=inactive, 0=active).

rdB\$triggers.column	Data Type	IDX	UQ	Description
RDB\$TRIGGER_TYPE	SMALLINT	—	—	1=before insert, 2=after insert, 3=before update, 4=after update, 5=before delete, 6=after delete. Multi-event triggers (Firebird 1.5 and onward) have various trigger types using higher numbers. The actual type code depends on which events are covered and the order in which the events are presented. (NB: There is no apparent reason that the order of events should make a difference to the trigger_type code.)
RDB\$DESCRIPTION	BLOB TEXT	—	—	Optional documentation.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	User-defined (0) or system-defined (1 or greater).
RDB\$FLAGS	SMALLINT	—	—	Internal use.
RDB\$DEBUG_INFO	BLOB BINARY			V.2.0+
RDB\$VALID_BLR	SMALLINT			(V.2.1+) Will be flagged (1) if an object on which the trigger depends is altered

RDB\$TYPES

Stores definitions of enumerated types used around Firebird.

Table V.30 RDB\$TYPES

rdB\$types.column	Data Type	IDX	UQ	Description
RDB\$FIELD_NAME	CHAR(31)	—	—	Column name for which this enumeration is defined. Note that the same column name appears consistently in multiple system tables.
RDB\$TYPE	SMALLINT	—	—	Enumeration ID for type that RDB\$TYPE_NAME identifies. The series of number is unique within a single enumerated type (e.g., 0=table, 1=view, 2=trigger, 3=computed column, 4=validation, 5=procedure are all types of RDB\$OBJECT_TYPE).

rdB\$types.column	Data Type	IDX	UQ	Description
RDB\$TYPE_NAME	CHAR(31)	Y	—	The text representation of the type identified by the RDB\$FIELD_NAME value and the RDB\$TYPE value.
RDB\$DESCRIPTION	BLOB TEXT	—	—	Optional documentation.
RDB\$SYSTEM_FLAG	SMALLINT	—	—	User-defined (0) or system-defined (1 or greater).

RDB\$USER_PRIVILEGES

Stores SQL permissions.

Table V.31 RDB\$USER_PRIVILEGES

rdB\$user_privileges.column	Data Type	IDX	UQ	Description
RDB\$USER	CHAR(31)	Y	—	User who has been granted the permission.
RDB\$GRANTOR	CHAR(31)	—	—	Name of user who granted the permission.
RDB\$PRIVILEGE	CHAR(6)	—	—	The privilege that is granted by the permission.
RDB\$GRANT_OPTION	SMALLINT	—	—	Whether the permission carries WITH GRANT OPTION authority. 1=Yes, 0=No.
RDB\$RELATION_NAME	CHAR(31)	Y	—	The object on which the permission has been granted.
RDB\$FIELD_NAME	CHAR(31)	—	—	Name of a column to which a column-level privilege applies (UPDATE or REFERENCES privileges only).
RDB\$USER_TYPE	SMALLINT	—	—	Identifies the type of user that was granted the permission (e.g., a user, procedure, view, etc.)
RDB\$OBJECT_TYPE	SMALLINT	—	—	Identifies the type of object on which the privilege was granted.

RDB\$VIEW_RELATIONS

An obsolete table.

System Views

The following system views are a subset of those defined in the SQL-92 standard. They can provide useful information about your data. You might like to copy the listings into a script and install the views in all your databases.

CHECK_CONSTRAINTS

Lists all of the CHECK constraints defined in the database, with the source code of the constraint definition.

```
CREATE VIEW CHECK_CONSTRAINTS (
    CONSTRAINT_NAME,
    CHECK_CLAUSE )
AS
    SELECT RC.RDB$CONSTRAINT_NAME,
    RT.RDB$TRIGGER_SOURCE
    FROM RDB$CHECK_CONSTRAINTS RC
    JOIN RDB$TRIGGERS RT
    ON RT.RDB$TRIGGER_NAME = RC.RDB$TRIGGER_NAME;
```

CONSTRAINTS_COLUMN_USAGE

Lists columns used by PRIMARY KEY and UNIQUE constraints and those defining FOREIGN KEY constraints.

```
CREATE VIEW CONSTRAINTS_COLUMN_USAGE (
    TABLE_NAME,
    COLUMN_NAME,
    CONSTRAINT_NAME )
AS
    SELECT RC.RDB$RELATION_NAME, RI.RDB$FIELD_NAME, RC.RDB$CONSTRAINT_NAME
    FROM RDB$RELATION_CONSTRAINTS RC
    JOIN RDB$INDEX_SEGMENTS RI
    ON RI.RDB$INDEX_NAME = RC.RDB$INDEX_NAME;
```

REFERENTIAL_CONSTRAINTS

Lists all the referential constraints defined in a database.

```
CREATE VIEW REFERENTIAL_CONSTRAINTS (
    CONSTRAINT_NAME,
    UNIQUE_CONSTRAINT_NAME,
    MATCH_OPTION,
    UPDATE_RULE,
    DELETE_RULE )
AS
    SELECT RDB$CONSTRAINT_NAME, RDB$CONST_NAME_UQ, RDB$MATCH_OPTION,
    RDB$UPDATE_RULE, RDB$DELETE_RULE
    FROM RDB$REF_CONSTRAINTS;
```

TABLE_CONSTRAINTS

Lists the table-level constraints.

```
CREATE VIEW TABLE_CONSTRAINTS (  
    CONSTRAINT_NAME,  
    TABLE_NAME,  
    CONSTRAINT_TYPE,  
    IS_DEFERRABLE,  
    INITIALLY_DEFERRED )  
AS  
    SELECT RDB$CONSTRAINT_NAME, RDB$RELATION_NAME,  
        RDB$CONSTRAINT_TYPE, RDB$DEFERRABLE, RDB$INITIALLY_DEFERRED  
    FROM RDB$RELATION_CONSTRAINTS;
```

Monitoring Tables

v.2.1 and higher

Database monitoring was introduced at Firebird 2.1 and enhanced at v.2.5. It works on databases of ODS 11.1 and higher.

The changes in v.2.5 for handling of file specifications and other character parameter items in the DPB are reflected in a change to the character set of the related columns in the MON\$ tables, which are defined by the system domain RDB\$FILE_NAME2. The character set of that domain definition changes at v.2.5, from NONE to UNICODE_FSS.

The columns affected by the character set change are MON\$DATABASE_NAME, MON\$ATTACHMENT_NAME and MON\$REMOTE_PROCESS.

MON\$CONTEXT_VARIABLES

Retrieves known context variables

Table V.32 MON\$CONTEXT_VARIABLES

mon\$context_variables.column	Data Type	Description	Vers.
MON\$ATTACHMENT_ID	INTEGER	Attachment ID. Contains a valid ID only for session-level context variables. Transaction-level variables have this field set to NULL.	V.2.5
MON\$TRANSACTION_ID	INTEGER	Transaction ID. Contains a valid ID only for transaction-level context variables. Session-level variables have this field set to NULL.	V.2.5
MON\$VARIABLE_NAME	VARCHAR(31)	Name of context variable	V.2.5
MON\$VARIABLE_VALUE	(Varies)	Value of context variable	V.2.5

MON\$DATABASE

Retrieves properties of the connected database

Table V.33 MON\$DATABASE

mon\$database.column	Data Type	Description	Vers.
MON\$DATABASE_NAME	VARCHAR(253)	Database pathname or alias. Character set is NONE in v.2.1, UNICODE_FSS from v.2.5 onward.	v.2.1
MON\$PAGE_SIZE	SMALLINT	Page size	v.2.1
MON\$ODS_MAJOR	SMALLINT	Major ODS version	v.2.1
MON\$ODS_MINOR	SMALLINT	Minor ODS version	v.2.1
MON\$OLDEST_TRANSACTION	INTEGER	Transaction ID of the oldest [interesting] transaction (OIT)	v.2.1
MON\$OLDEST_ACTIVE	INTEGER	Transaction ID of the oldest active transaction (OAT)	v.2.1

mon\$database.column	Data Type	Description	Vers.
MON\$OLDEST_SNAPSHOT	INTEGER	Transaction ID of the Oldest Snapshot (OST), i.e., the number of the OAT when the last garbage collection was done)	v.2.1
MON\$NEXT_TRANSACTION	INTEGER	Transaction ID of the next transaction that will be started	v.2.1
MON\$PAGE_BUFFERS	INTEGER	Number of pages allocated in the page cache	v.2.1
MON\$SQL_DIALECT	SMALLINT	SQL dialect of the database	v.2.1
MON\$SHUTDOWN_MODE	SMALLINT	Current shutdown mode: 0: online 1: multi-user shutdown 2: single-user shutdown 3: full shutdown	v.2.1
MON\$SWEEP_INTERVAL	INTEGER	The sweep interval configured in the database header. Value 0 indicates that sweeping is disabled.	v.2.1
MON\$READ_ONLY	SMALLINT	Read-only/Read-write flag	v.2.1
MON\$FORCED_WRITES	SMALLINT	Synchronous/asynchronous writes flag	v.2.1
MON\$RESERVE_SPACE	SMALLINT	Reserve space/Use-all-space flag	v.2.1
MON\$CREATION_DATE	TIMESTAMP	Creation date and time, i.e., when the database was created or last restored.	v.2.1
MON\$PAGES	BIGINT	Number of pages allocated on disk. Multiply by page size to estimate the on-disk size of the database at snapshot time. Note that a database on a raw device always returns 0	v.2.1
MON\$BACKUP_STATE	SMALLINT	Current state of database with respect to <i>nbackup</i> physical backup: 0: normal 1: stalled 2: merge	v.2.1
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1

MON\$ATTACHMENTS

Retrieves attachments connected to the connected database

Table V.34 MON\$ATTACHMENTS

mon\$attachments.column	Data Type	Description	Vers.
MON\$ATTACHMENT_ID	INTEGER	Attachment ID	v.2.1
MON\$SERVER_PID	INTEGER	Server Process ID	v.2.1

mon\$attachments.column	Data Type	Description	Vers.
MON\$STATE	SMALLINT	Attachment state 0: idle 1: active	v.2.1
MON\$ATTACHMENT_NAME	VARCHAR(253)	Connection string. Character set is NONE in v.2.1, UNICODE_FSS from v.2.5 onward.	v.2.1
MON\$USER	CHAR(93)	User name	v.2.1
MON\$ROLE	CHAR(93)	Role name	v.2.1
MON\$REMOTE_PROTOCOL	VARCHAR(8)	Remote protocol name	v.2.1
MON\$REMOTE_ADDRESS	VARCHAR(253)	Remote address	v.2.1
MON\$REMOTE_PID	INTEGER	Remote client process ID, contains non-NULL values only if the client library is version 2.1 or higher	v.2.1
MON\$REMOTE_PROCESS	VARCHAR(253)	Remote client process pathname. Contains non-NULL values only if the client library is version 2.1 or higher. Can contain a non-pathname value if an application has specified a custom process name via the DPB. Character set is NONE in v.2.1, UNICODE_FSS from v.2.5 onward.	v.2.1
MON\$CHARACTER_SET_ID	SMALLINT	Attachment character set	v.2.1
MON\$TIMESTAMP	TIMESTAMP	Connection date and time	v.2.1
MON\$GARBAGE_COLLECTION	SMALLINT	Garbage collection flag, indicates whether GC is allowed for this attachment (as specified via the DPB in isc_attach_database)	v.2.1
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1

MON\$TRANSACTIONS

Retrieves transactions started in the connected database

Table V.35 MON\$TRANSACTIONS

mon\$transactions.column	Data Type	Description	Vers.
MON\$TRANSACTION_ID	INTEGER	Transaction ID	v.2.1
MON\$ATTACHMENT_ID	INTEGER	Attachment ID	v.2.1

mon\$transactions.column	Data Type	Description	Vers.
MON\$STATE	SMALLINT	Transaction state 0: Idle (transaction has one or more statements that are prepared and waiting to execute and no statements with open cursors) 1: Active (transaction has one or more statements executing or fetching or with pending inserts, updates or deletes)	v.2.1
MON\$TIMESTAMP	TIMESTAMP	Transaction start date and time	v.2.1
MON\$TOP_TRANSACTION	INTEGER	ID of Top transaction, the upper limit used by the sweeper transaction when advancing the global OIT. All transactions above this threshold are considered active. It is normally equivalent to the MON\$TRANSACTION_ID but COMMIT RETAINING or ROLLBACK RETAINING will cause MON\$TOP_TRANSACTION to remain unchanged (“stuck”) when the transaction ID is incremented.	v.2.1
MON\$OLDEST_TRANSACTION	INTEGER	Local OIT ID (i.e., the OIT as known within the transaction's own isolation context)	v.2.1
MON\$OLDEST_ACTIVE	INTEGER	Local OAT ID (i.e., the OAT as known within the transaction's own isolation context)	v.2.1
MON\$ISOLATION_MODE	SMALLINT	Isolation level 0: consistency 1: concurrency 2: read committed record version 3: read committed no record version	v.2.1
MON\$LOCK_TIMEOUT	SMALLINT	Lock timeout -1: infinite wait 0: no wait N: timeout configured as N seconds	v.2.1
MON\$READ_ONLY	SMALLINT	Read-only flag	v.2.1
MON\$AUTO_COMMIT	SMALLINT	Auto-commit flag	

mon\$transactions.column	Data Type	Description	Vers.
MON\$AUTO_UNDO	SMALLINT	Auto-undo flag, indicates the auto-undo status set for the transaction, i.e., whether a transaction-level savepoint was created. The existence of the transaction-level savepoint allows changes to be undone if ROLLBACK is called and the transaction is then just committed. If this savepoint does not exist, or it does exist but the number of changes is very large, then an actual ROLLBACK is executed and the the transaction is marked in the transaction inventory (TIP) as “dead”.	
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1

MON\$STATEMENTS

Retrieved prepared statements in the connected database

Table V.36 MON\$STATEMENTS

mon\$statements.column	Data Type	Description	Vers.
MON\$STATEMENT_ID	INTEGER	Statement ID	v.2.1
MON\$ATTACHMENT_ID	INTEGER	Attachment ID	v.2.1
MON\$TRANSACTION_ID	INTEGER	Transaction ID—contains valid values for active statements only	v.2.1
MON\$STATE	SMALLINT	Statement state 0: Idle (state after prepare, until execution begins) 1: Active (state during execution and fetch. Idle state (0) returns after cursor is closed) 2: Stalled, i.e., in the interval between client fetches from the open cursor. CPU time is not used during this state. ¹	v.2.1
MON\$TIMESTAMP	TIMESTAMP	Statement start date and time. Values are valid for active statements only.	v.2.1
MON\$SQL_TEXT	BLOB TEXT	Statement text, if applicable. Contains NULL for GDML statements.	v.2.1
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1

1. The *Stalled* state was introduced in Firebird 2.5.1. Priorly, statements in this state were rolled in with *Active*.

Note *The execution plan and the values of parameters are not available.*

MON\$CALL_STACK

Retrieves call stacks of active PSQL requests in the connected database

Table V.37 MON\$CALL_STACK

mon\$call_stack.column	Data Type	Description	Vers.
MON\$CALL_ID	INTEGER	Call ID	v.2.1
MON\$STATEMENT_ID	INTEGER	Top-level DSQL statement ID, groups call stacks by the top-level DSQL statement that initiated the call chain. This ID represents an active statement record in the table MON\$STATEMENTS.	v.2.1
MON\$CALLER_ID	INTEGER	Caller request ID	v.2.1
MON\$OBJECT_NAME	CHAR(93)	PSQL object name (Trigger name, procedure name)	v.2.1
MON\$OBJECT_TYPE	SMALLINT	PSQL object type 2: trigger 5: procedure	v.2.1
MON\$TIMESTAMP	TIMESTAMP	Date and time of start of request.	v.2.1
MON\$SOURCE_LINE	INTEGER	SQL source line number, contains line and column information related to the PSQL statement currently being executed	v.2.1
MON\$SOURCE_COLUMN	INTEGER	SQL source column number, contains line and column information related to the PSQL statement currently being executed	v.2.1
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1

MON\$IO_STATS

Retrieves disk input/output statistics

Table V.38 MON\$IO_STATS

mon\$iostats.column	Data Type	Description	Vers.
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1
MON\$STAT_GROUP	SMALLINT	Statistics group ID 0: database 1: attachment 2: transaction 3: statement 4: call	v.2.1
MON\$PAGE_READS	BIGINT	Number of page reads	v.2.1

mon\$iostats.column	Data Type	Description	Vers.
MON\$PAGE_WRITES	BIGINT	Number of page writes	v.2.1
MON\$PAGE_FETCHES	BIGINT	Number of page fetches	v.2.1
MON\$PAGE_MARKS	BIGINT	Number of pages with changes pending	v.2.1

MON\$MEMORY_USAGE

Retrieves current memory usage on the host of the connected database

Table V.39 MON\$MEMORY_USAGE.

mon\$memory_usage.column	Data Type	Description	Vers.
MON\$STAT_ID	INTEGER	Statistics ID	v.2.5
MON\$STAT_GROUP		Statistics group 0: database 1: attachment 2: transaction 3: statement 4: call	v.2.5
MON\$MEMORY_USED	BIGINT	Memory usage statistics in MON\$STATEMENTS and MON\$STATE represent actual CPU consumption Number of bytes currently in use. High-level memory allocations performed by the engine from its pools. Can be useful for tracing memory leaks and for investigating unusual memory consumption and the attachments, procedures, etc. that might be responsible for it.	v.2.5
MON\$MEMORY_ALLOCATED	BIGINT	Number of bytes currently allocated at the OS level. Low-level memory allocations performed by the Firebird memory manager. These are bytes actually allocated by the operating system, so it enables the physical memory consumption to be monitored. On the whole, only MON\$DATABASE and memory-bound objects point to non-zero “allocated” values. Small allocations are not allocated at this level, being redirected to the database memory pool instead.	v.2.5

mon\$memory_usage.column	Data Type	Description	Vers.
MON\$MAX_MEMORY_USED	BIGINT	Maximum number of bytes used by this object.	v.2.5
MON\$MAX_MEMORY_ALLOCATED	BIGINT	Maximum number of bytes allocated from the operating system by this object	v.2.5

MON\$RECORD_STATS

Retrieves record-level statistics

Table V.40 MON\$RECORD_STATS

mon\$record_state.column	Data Type	Description	Vers.
MON\$STAT_ID	INTEGER	Statistics ID	v.2.1
MON\$STAT_GROUP	SMALLINT	Statistics group ID 0: database 1: attachment 2: transaction 3: statement 4: call	v.2.1
MON\$RECORD_SEQ_READS	BIGINT	Number of records read sequentially	v.2.1
MON\$RECORD_IDX_READS	BIGINT	Number of records read via an index	v.2.1
MON\$RECORD_INSERTS	BIGINT	Number of inserted records	v.2.1
MON\$RECORD_UPDATES	BIGINT	Number of updated records	v.2.1
MON\$RECORD_DELETES	BIGINT	Number of deleted records	v.2.1
MON\$RECORD_BACKOUTS	BIGINT	Number of records where a new primary record version or a change to an existing primary record version is backed out due to rollback or savepoint undo	v.2.1
MON\$RECORD_PURGES	BIGINT	Number of records where the record version chain is being purged of versions no longer needed by OAT or younger transactions	v.2.1
MON\$RECORD_EXPUNGES	BIGINT	Number of records where record version chain is being deleted due to deletions by transactions older than OAT	v.2.1

APPENDIX

VI

CHARACTER SETS AND COLLATIONS

If you have installed a later version and the character set or collation sequence you want is not listed here, read the release notes of your version and any other versions since v.2.5 to see whether it has been added.

Implemented and Activated Character Sets

Table VI.1 lists the character sets and collations available to databases when Firebird 2.5 was released. Some of those listed are not available in earlier Firebird versions.

Table VI.1 Character Sets and Collations, Firebird 2.5

ID	Name	Bytes per Character	Collation	Language	Aliases
2	ASCII	1	ASCII	English	ASCII7, USASCII
56	BIG_5	2	BIG_5	Chinese, Vietnamese, Korean	BIG5, DOS_950, WIN_950
68	CP943C	4	CP943C_UNICODE	Japanese (V.2.1+)	—
50	CYRL	1	CYRL	Russian	—
			DB_RUS	dBase Russian	—
			PDOX_CYRL	Paradox Russian	—
10	DOS437	1	DOS437	English—USA	DOS_437
			DB_DEU437	dBase German	—
			DB_ESP437	dBase Spanish	—
			DB_FRA437	dBase French	—
			DB_FIN437	dBase Finnish	—
			DB_ITA437	dBase Italian	—
			DB_NLD437	dBase Dutch	—

ID	Name	Bytes per Character	Collation	Language	Aliases
10	DOS437	1	DB_SVE437	dBase Swedish	—
			DB_UK437	dBase English—UK	—
			DB_US437	dBase English—USA	—
			PDOX_ASCII	Paradox ASCII code page	—
			PDOX_SWEDFIN	Paradox Swedish/Finnish code pages	—
			PDOX_INTL	Paradox International English code page	—
9	DOS737	1	DOS737	Greek	DOS_737
15	DOS775	1	DOS775	Baltic	DOS_775
11	DOS850	1	DOS850	Latin I (no Euro symbol)	DOS_850
			DB_DEU850	German	—
			DB_ESP850	Spanish	—
			DB_FRA850	French	—
			DB_FRC850	French—Canada	—
			DB_ITA850	Italian	—
			DB_NLD850	Dutch	—
			DB_PTB850	Portuguese—Brazil	—
			DB_SVE850	Swedish	—
			DB_UK850	English—UK	—
			DB_US850	English—USA	—
45	DOS852	1	DOS852	Latin II	DOS_852
			DB_CSY	dBase Czech	—
			DB_PLK	dBase Polish	—
			DB_SLO	dBase Slovakian	—
			PDOX_PLK	Paradox Polish	—
			PDOX_HUN	Paradox Hungarian	—
			PDOX_SLO	Paradox Slovakian	—
			PDOX_CSY	Paradox Czech	—
46	DOS857	1	DOS857	Turkish	DOS_857
			DB_TRK	dBase Turkish	—
16	DOS858	1	DOS858	Latin I + Euro symbol	DOS_858
13	DOS860	1	DOS860	Portuguese	DOS_860
			DB_PTG860	dBase Portuguese	—
47	DOS861	1	DOS861	Icelandic	DOS_861
			PDOX_ISL	Paradox Icelandic	—
17	DOS862	1	DOS862	Hebrew	DOS_862
14	DOS863	1	DOS863	French—Canada	DOS_863
			DB_FRC863	dBase French—Canada	—

ID	Name	Bytes per Character	Collation	Language	Aliases
18	DOS864	1	DOS864	Arabic	DOS_864
12	DOS865	1	DOS865	Nordic	DOS_865
			DB_DAN865	dBase Danish	—
			DB_NOR865	dBase Norwegian	—
12	DOS865	1	PDOX_NORDAN4	Paradox Norwegian and Danish	—
48	DOS866	1	DOS866	Russian	DOS_866
49	DOS869	1	DOS869	Modern Greek	DOS_869
6	EUCJ_0208	2	EUCJ_0208	EUC Japanese	EUCJ
57	GB_2312	2	GB_2312	Simplified Chinese (Hong Kong, PRC)	DOS_936, GB2312, WIN_936
69	GB18030	4		Chinese national standard describing the required language and character support necessary for software in China. Activated from ICU in V.2.5+	—
67	GBK	2		Chinese, sub-set of GB18030 and super-set of GB_2312	—
21	ISO8859_1	1	ISO8859_1	Latin 1	ANSI, ISO88591, LATIN1
			FR_CA	French—Canada	—
			DA_DA	Danish	—
			DE_DE	German	—
			ES_ES	Spanish	—
			ES_ES_CI_AI (v.2.0 +)	Spanish case- and accent-insensitive	—
			FI_FI	Finnish	—
			FR_FR	French	—
			FR_FR_CI_AI	French—case-insensitive and accent-insensitive (V.2.1+)	—
			IS_IS	Icelandic	—
			IT_IT	Italian	—
			NO_NO	Norwegian	—
			DU_NL	Dutch	—
			PT_PT	Portuguese	—
			PT_BR	Portuguese—Brazil (V.2.0+)	—
			SV_SV	Swedish	—

ID	Name	Bytes per Character	Collation	Language	Aliases
21	ISO8859_1	1	EN_UK EN_US	English—UK English—USA	— —
22	ISO8859_2	1	ISO8859_2	Latin 2—Central European (Croatian, Czech, Hungarian, Polish, Romanian, Serbian, Slovakian, Slovenian)	ISO-8859-2, ISO88592, LATIN2
22	ISO8859_2		CS_CZ ISO_HUN ISO_PLK	Czech Hungarian Polish (V.2.0+)	— — —
23	ISO8859_3	1	ISO8859_3	Latin3—Southern European (Maltese, Esperanto)	ISO-8859-3, ISO88593, LATIN3
34	ISO8859_4	1	ISO8859_4	Latin 4—Northern European (Estonian, Latvian, Lithuanian, Greenlandic, Lappish)	ISO-8859-4, ISO88594, LATIN4
35	ISO8859_5	1	ISO8859_5	Cyrillic (Russian)	ISO-8859-5, ISO88595
36	ISO8859_6	1	ISO8859_6	Arabic	ISO-8859-6, ISO88596
37	ISO8859_7	1	ISO8859_7	Greek	ISO-8859-7, ISO88597
38	ISO8859_8	1	ISO8859_8	Hebrew	ISO-8859-8, ISO88598
39	ISO8859_9	1	ISO8859_9	Latin 5	ISO-8859-9, ISO88599, LATIN5
40	ISO8859_13	1	ISO8859_13	Latin 7—Baltic Rim	ISO-8859-13, ISO885913, LATIN7
63	KOI8R		KOI8-RU	Russian character set and dictionary collation (v.2.0+)	—
64	KOI8U		KOI8-UA	Ukrainian character set and dictionary collation (v.2.0+)	—
44	KSC_5601	2	KSC_5601	Korean (Unified Hangeul)	DOS_949, KSC5601, WIN_949
			KSC_DICTIONARY	Korean—dictionary order collation	—
19	NEXT	1	NEXT NXT_US NXT_FRA NXT_ITA	NeXTSTEP encoding English—USA French Italian	— — — —

ID	Name	Bytes per Character	Collation	Language	Aliases
19	NEXT	1	NXT_ESP	Spanish	—
			NXT_DEU	German	—
0	NONE	1	NONE	Code-page neutral; uppercasing limited to ASCII codes 97–122	—
1	OCTETS	1	OCTETS	Binary character	BINARY
5	SJIS_0208	2	SJIS_0208	Japanese	SJIS
66	TIS620	1	TIS620_UNICODE	Thai (V.2.1+)	
3	UNICODE_FSS	3	UNICODE_FSS	Specialised set of UNICODE UTF8	SQL_TEXT, UTF-8 (v.1.X only), UTF8 (v.1.X only), UTF_FSS
4	UTF8	1-4	UCS_BASIC	Universal, UNICODE 4 (V.2.0+)	UTF_8, UTF-8
			UNICODE	UNICODE collation (V.2.0+)	—
			UNICODE_CI	UNICODE case-insensitive (V.2.1+)	—
			UNICODE_CI_AI	UNICODE case-insensitive, accent-insensitive (V.2.5+)	—
51	WIN1250	1	WIN1250	ANSI—Central European	WIN_1250
			BS_BA	Bosnian (V.2.0+)	—
			WIN_CZ	Czech (V.2.0+)	—
			WIN_CZ_AI	Czech—accent-insensitive (V.2.0+)	—
			WIN_CZ_CI_AI	Czech—case-insensitive, accent-insensitive (V.2.0+)	—
			PXW_CSZ	Czech	—
			PXW_PLK	Polish	—
			PXW_HUN	Hungarian	—
			PXW_HUNDC	Hungarian—dictionary sort	—
			PXW_SLOV	Slovakian	—
52	WIN1251	1	WIN1251	ANSI—Cyrillic	WIN_1251
			WIN1251_UA	Ukrainian	—
			PXW_CYRL	Paradox Cyrillic (Russian)	—
53	WIN1252	1	WIN1252	ANSI—Latin I	WIN_1252
			PXW_SWEDFIN	Swedish and Finnish	—
			PXW_NORDAN4	Norwegian and Danish	—
			PXW_INTL	English—International	—

ID	Name	Bytes per Character	Collation	Language	Aliases
53	WIN1252	1	PXW_INTL850	Paradox Multi-lingual Latin I	—
			WIN_PTBR	Portuguese—Brazil (V.2.0+)	—
			PXW_SPAN	Paradox Spanish	—
54	WIN1253	1	WIN1253	ANSI Greek	WIN_1253
			PXW_GREEK	Paradox Greek	—
55	WIN1254	1	WIN1254	ANSI Turkish	WIN_1254
			PXW_TURK	Paradox Turkish	—
58	WIN1255	1	WIN1255	ANSI Hebrew	WIN_1255
59	WIN1256	1	WIN1256	ANSI Arabic	WIN_1256
60	WIN1257	1	WIN1257	ANSI Baltic	WIN_1257
			WIN1257_EE	Estonian dictionary collation (v.2.0+)	—
			WIN1257_LT	Lithuanian dictionary collation (v.2.0+)	—
			WIN1257_LV	Latvian dictionary collation (v.2.0+)	—
65	WIN1258	1	WIN1258	Vietnamese (v.2.0+)	WIN_1258 (V.2.5+)

APPENDIX

VII

FIREBIRD ERROR CODES

The error codes returned to applications or PSQL modules by Firebird 2.1 and 2.5 are listed in Table VII.1.

A few codes are unavailable to lower versions of Firebird and the formatting of messages changed from Firebird 2.0 forward. It is important to ensure that both server and client have the correct version of `firebird.msg` (`interbase.msg` for v.1.0) stored in the Firebird root directory (for servers) and in the application directory for embedded server applications and remote service tools.

It is not mandatory to install the message file on clients but it is certainly recommended when running the command-line tools remotely or for applications that use the Services API. If present, it must be the correct version for the server that the client is attaching to.

Table VII.1 Firebird Error Codes

SQL	ISCCode	Symbol (GDSCODE)	Text
101	335544366	segment	Segment buffer length shorter than expected
100	335544338	from_no_match	No match for first value expression
100	335544354	no_record	Invalid database key
100	335544367	segstr_eof	Attempted retrieval of more segments than exist
100	335544374	stream_eof	Attempt to fetch past the last record in a record stream
0	335741039	gfix_opt_SQL_dialect	-sql_dialectset database dialect n
-84	335544554	nonsql_security_rel	Table/procedure has non-SQL security class defined
-84	335544555	nonsql_security fld	Column has non-SQL security class defined
-84	335544668	dsql_procedure_use_err	Procedure %s does not return any values
-85	335544747	usrname_too_long	The username entered is too long. Maximum length is 31 bytes.
-85	335544748	password_too_long	The password specified is too long. Maximum length is 8 bytes.
-85	335544749	usrname_required	A username is required for this operation.

SQL	ISCCode	Symbol (GDSCODE)	Text
-85	335544750	password_required	A password is required for this operation
-85	335544751	bad_protocol	The network protocol specified is invalid
-85	335544752	dup_username_found	A duplicate user name was found in the security database
-85	335544753	username_not_found	The user name specified was not found in the security database
-85	335544754	error_adding_sec_record	An error occurred while attempting to add the user.
-85	335544755	error_modifying_sec_record	An error occurred while attempting to modify the user record.
-85	335544756	error_deleting_sec_record	An error occurred while attempting to delete the user record.
-85	335544757	error_updating_sec_db	An error occurred while updating the security database.
-103	335544571	dsql_constant_err	Data type for constant unknown
-104	336003075	dsql_transitional_numeric	Precision 10 to 18 changed from DOUBLE PRECISION in SQL dialect 1 to 64-bit scaled integer in SQL dialect 3
-104	336003077	sql_db_dialect_dtype_unsupport	Database SQL dialect %d does not support reference to %s datatype
-104	336003087	dsql_invalid_label	Label %s %s in the current scope
-104	336003088	dsql_datatypes_not_comparable	Datatypes %s are not comparable in expression %s
-104	335544343	invalid_blr	Invalid request BLR at offset %ld
-104	335544390	syntaxerr	BLR syntax error: expected %s at offset %ld, encountered %ld
-104	335544425	ctxinuse	Context already in use (BLR error)
-104	335544426	ctxnotdef	Context not defined (BLR error)
-104	335544429	badparnum	Bad parameter number
-104	335544440	bad_msg_vec	
-104	335544456	invalid_sdl	Invalid slice description language at offset %ld
-104	335544570	dsql_command_err	Invalid command
-104	335544579	dsql_internal_err	Internal error
-104	335544590	dsql_dup_option	Option specified more than once
-104	335544591	dsql_tran_err	Unknown transaction option
-104	335544592	dsql_invalid_array	Invalid array reference
-104	335544608	command_end_err	Unexpected end of command
-104	335544612	token_err	Token unknown
-104	335544634	dsql_token_unk_err	Token unknown - line %ld, char %ld
-104	335544709	dsql_agg_ref_err	Invalid aggregate reference
-104	335544714	invalid_array_id	Invalid blob id
-104	335544730	cse_not_supported	Client/Server Express not supported in this release
-104	335544743	token_too_long	Token size exceeds limit
-104	335544763	invalid_string_constant	A string constant is delimited by double quotes

SQL	ISCCode	Symbol (GDSCODE)	Text
-104	335544764	transitional_date	DATE must be changed to TIMESTAMP
-104	335544796	sql_dialect_datatype_unsupport	Client SQL dialect %d does not support reference to %s datatype
-104	335544798	depend_on_uncommitted_rel	You created an indirect dependency on uncommitted metadata. You must roll back the current transaction.
-104	335544821	dsql_column_pos_err	Invalid column position used in the %s clause
-104	335544822	dsql_agg_where_err	Cannot use an aggregate function in a WHERE clause, use HAVING instead
-104	335544823	dsql_agg_group_err	Cannot use an aggregate function in a GROUP BY clause
-104	335544824	dsql_agg_column_err	Invalid expression in the %s (not contained in either an aggregate function or the GROUP BY clause)
-104	335544825	dsql_agg_having_err	Invalid expression in the %s (neither an aggregate function nor a part of the GROUP BY clause)
-104	335544826	dsql_agg_nested_err	Nested aggregate functions are not allowed
-105	335544702	like_escape_invalid	Invalid ESCAPE sequence
-105	335544789	extract_input_mismatch	Specified EXTRACT part does not exist in input datatype
-150	335544360	read_only_rel	Attempted update of read-only table
-150	335544362	read_only_view	Cannot update read-only view %s
-150	335544446	non_updatable	Not updatable
-150	335544546	constaint_on_view	Cannot define constraints on views
-151	335544359	read_only_field	Attempted update of read-only column
-155	335544658	dsql_base_table	%s is not a valid base table of the specified view
-157	335544598	specify_field_err	Must specify column name for view select expression
-158	335544599	num_field_err	Number of columns does not match select list
-162	335544685	no_dbkey	Dbkey not available for multi-table views
-170	335544512	prcmismat	Parameter mismatch for procedure %s
-170	335544619	extern_func_err	External functions cannot have more than 10 parameters
-171	335544439	funmismat	Function %s could not be matched
-171	335544458	invalid_dimension	Column not array or invalid dimensions (expected %ld, encountered %ld)
-171	335544618	return_mode_err	Return mode by value not allowed for this data type
-172	335544438	funnotdef	Function %s is not defined
-203	335544708	dyn_fld_ambiguous	Ambiguous column reference.
-204	336003085	dsql_ambiguous_field_name	Ambiguous field name between %s and %s
-204	335544463	gennotdef	Generator %s is not defined
-204	335544502	stream_not_defined	Reference to invalid stream number
-204	335544509	charset_not_found	CHARACTER SET %s is not defined
-204	335544511	prcnnotdef	Procedure %s is not defined

SQL	ISCCode	Symbol (GDSCODE)	Text
-204	335544515	codnotdef	Status code %s unknown
-204	335544516	xcpnotdef	Exception %s not defined
-204	335544532	ref_cnstrnt_notfound	Name of Referential Constraint not defined in constraints table.
-204	335544551	grant_obj_notfound	Could not find table/procedure for GRANT
-204	335544568	text_subtype	Implementation of text subtype %d not located.
-204	335544573	dsql_datatype_err	Data type unknown
-204	335544580	dsql_relation_err	Table unknown
-204	335544581	dsql_procedure_err	Procedure unknown
-204	335544588	collation_not_found	COLLATION %s is not defined
-204	335544589	collation_not_for_charset	COLLATION %s is not valid for specified CHARACTER SET
-204	335544595	dsql_trigger_err	Trigger unknown
-204	335544620	alias_conflict_err	Alias %s conflicts with an alias in the same statement
-204	335544621	procedure_conflict_error	Alias %s conflicts with a procedure in the same statement
-204	335544622	relation_conflict_err	Alias %s conflicts with a table in the same statement
-204	335544635	dsql_no_relation_alias	There is no alias or table named %s at this scope level
-204	335544636	indexname	There is no index %s for table %s
-204	335544640	collation_requires_text	Invalid use of CHARACTER SET or COLLATE
-204	335544662	dsql_blob_type_unknown	BLOB SUB_TYPE %s is not defined
-204	335544759	bad_default_value	Can not define a not null column with NULL as default value
-204	335544760	invalid_clause	Invalid clause --- '%s'
-204	335544800	too_many_contexts	Too many Contexts of Relation/Procedure/Views. Maximum allowed is 127
-204	335544817	bad_limit_param	Invalid parameter to FIRST. Only integers >= 0 are allowed.
-204	335544818	bad_skip_param	Invalid parameter to SKIP. Only integers >= 0 are allowed.
-204	335544837	bad_substring_param	Invalid %s parameter to SUBSTRING. Only positive integers are allowed.
-205	335544396	fldnotdef	Column %s is not defined in table %s
-205	335544552	grant_fld_notfound	Could not find column for GRANT
-206	335544578	dsql_field_err	Column unknown
-206	335544587	dsql_blob_err	Column is not a BLOB
-206	335544596	dsql_subselect_err	Subselect illegal in this context
-208	335544617	order_by_err	Invalid ORDER BY clause
-219	335544395	relnotdef	Table %s is not defined
-230	335544487	walw_err	WAL Writer error
-231	335544488	logh_small	Log file header of %s too small

SQL	ISCCode	Symbol (GDSCODE)	Text
-232	335544489	logh_inv_version	Invalid version of log file %s
-233	335544490	logh_open_flag	Log file %s not latest in the chain but open flag still set
-234	335544491	logh_open_flag2	Log file %s not closed properly; database recovery may be required
-235	335544492	logh_diff_dbname	Database name in the log file %s is different
-236	335544493	logf_unexpected_eof	Unexpected end of log file %s at offset %ld
-237	335544494	logr_incomplete	Incomplete log record at offset %ld in log file %s
-238	335544495	logr_header_small	Log record header too small at offset %ld in log file %s
-239	335544496	logb_small	Log block too small at offset %ld in log file %s
-239	335544691	cache_too_small	Insufficient memory to allocate page buffer cache
-239	335544693	log_too_small	Log size too small
-239	335544694	partition_too_small	Log partition size too small
-243	335544500	no_wal	Database does not use Write-ahead Log
-257	335544566	start_cm_for_wal	WAL defined; Cache Manager must be started first
-260	335544690	cache_redef	Cache redefined
-260	335544692	log_redef	Log redefined
-261	335544695	partition_not_supp	Partitions not supported in series of log file specification
-261	335544696	log_length_spec	Total length of a partitioned log must be specified
-281	335544637	no_stream_plan	Table %s is not referenced in plan
-282	335544638	stream_twice	Table %s is referenced more than once in plan; use aliases to distinguish
-282	335544643	dsql_self_join	The table %s is referenced twice; use aliases to differentiate
-282	335544659	duplicate_base_table	Table %s is referenced twice in view; use an alias to distinguish
-282	335544660	view_alias	View %s has more than one base table; use aliases to distinguish
-282	335544710	complex_view	Navigational stream %ld references a view with more than one base table
-283	335544639	stream_not_found	Table %s is referenced in the plan but not the from list
-284	335544642	index_unused	Index %s cannot be used in the specified plan
-291	335544531	primary_key_notnull	Column used in a PRIMARY/UNIQUE constraint must be NOT NULL.
-292	335544534	ref_cnstrnt_update	Cannot update constraints (RDB\$REF_CONSTRAINTS).
-293	335544535	check_cnstrnt_update	Cannot update constraints (RDB\$CHECK_CONSTRAINTS).
-294	335544536	check_cnstrnt_del	Cannot delete CHECK constraint entry (RDB\$CHECK_CONSTRAINTS)
-295	335544545	rel_cnstrnt_update	Cannot update constraints (RDB\$RELATION_CONSTRAINTS).

SQL	ISCCode	Symbol (GDSCODE)	Text
-296	335544547	invald_cnstrnt_type	Internal gds software consistency check (invalid RDB\$CONSTRAINT_TYPE)
-297	335544558	check_constraint	Operation violates CHECK constraint %s on view or table %s
-313	335544669	dsql_count_mismatch	Count of column list and variable list do not match
-314	335544565	transliteration_failed	Cannot transliterate character between character sets
-315	336068815	dyn_dtype_invalid	Cannot change datatype for column %s. Changing datatype is not supported for BLOB or ARRAY columns.
-383	336068814	dyn_dependency_exists	Column %s from table %s is referenced in %s
-401	335544647	invalid_operator	Invalid comparison operator for find operation
-402	335544368	segstr_no_op	Attempted invalid operation on a BLOB
-402	335544414	blobsnotsup	BLOB and array data types are not supported for %s operation
-402	335544427	datnotsup	Data operation not supported
-406	335544457	out_of_bounds	Subscript out of bounds
-407	335544435	nullsegkey	Null segment of UNIQUE KEY
-413	335544334	convert_error	Conversion error from string "%s"
-413	335544454	nofilter	Filter not found to convert type %ld to type %ld
-501	335544327	bad_req_handle	Invalid request handle
-501	335544577	dsql_cursor_close_err	Attempt to reclose a closed cursor
-502	335544574	dsql_decl_err	Declared cursor already exists
-502	335544576	dsql_cursor_open_err	Attempt to reopen an open cursor
-504	335544572	dsql_cursor_err	Cursor %s %s
-508	335544348	no_cur_rec	No current record for fetch operation
-510	335544575	dsql_cursor_update_err	Cursor not updatable
-518	335544582	dsql_request_err	Request unknown
-519	335544688	dsql_open_cursor_request	The prepare statement identifies a prepare statement with an open cursor
-530	335544466	foreign_key	Violation of FOREIGN KEY constraint "%s" on table "%s"
-531	335544597	dsql_crdp_prepare_err	Cannot prepare a CREATE DATABASE/SCHEMA statement
-532	335544469	trans_invalid	Transaction marked invalid by I/O error
-551	335544352	no_priv	No permission for %s access to %s %s
-551	335544790	insufficient_svc_privileges	Service %s requires SYSDBA permissions. Reattach to the Service Manager using the SYSDBA account.
-552	335544550	not_rel_owner	Only the owner of a table may reassign ownership
-552	335544553	grant_nopriv	User does not have GRANT privileges for operation
-552	335544707	grant_nopriv_on_base	User does not have GRANT privileges on base table/view for operation
-553	335544529	existing_priv_mod	Cannot modify an existing user privilege

SQL	ISCCode	Symbol (GDSCODE)	Text
-595	335544645	stream_crack	The current position is on a crack
-596	335544644	stream_bof	Illegal operation when at beginning of stream
-597	335544632	dsql_file_length_err	Preceding file did not specify length, so %s must include starting page number
-598	335544633	dsql_shadow_number_err	Shadow number must be a positive integer
-599	335544607	node_err	Gen.c: node not supported
-599	335544625	node_name_err	A node name is not permitted in a secondary, shadow, cache or log file name
-600	335544680	crp_data_err	Sort error: corruption in data structure
-601	335544646	db_or_file_exists	Database or file exists
-604	335544593	dsql_max_arr_dim_exceeded	Array declared with too many dimensions
-604	335544594	dsql_arr_range_error	Illegal array dimension range
-605	335544682	dsql_field_ref	Inappropriate self-reference of column
-607	336003074	dsql_dbkey_from_non_table	Cannot SELECT RDB\$DB_KEY from a stored procedure.
-607	336003086	dsql_udf_return_pos_err	External function should have return position between 1 and %d
-607	335544351	no_meta_update	Unsuccessful metadata update
-607	335544549	systrig_update	Cannot modify or erase a system trigger
-607	335544657	dsql_no_blob_array	Array/BLOB/DATE data types not allowed in arithmetic
-607	335544746	reftable_requires_pk	"REFERENCES table" without "(column)" requires PRIMARY KEY on referenced table
-607	335544815	generator_name	GENERATOR %s
-607	335544816	udf_name	UDF %s
-612	336068812	dyn_domain_name_exists	Cannot rename domain %s to %s. A domain with that name already exists.
-612	336068813	dyn_field_name_exists	Cannot rename column %s to %s. A column with that name already exists in table %s.
-615	335544475	relation_lock	Lock on table %s conflicts with existing lock
-615	335544476	record_lock	Requested record lock conflicts with existing lock
-615	335544507	range_in_use	Refresh range number %ld already in use
-616	335544530	primary_key_ref	Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.
-616	335544539	integ_index_del	Cannot delete index used by an Integrity Constraint
-616	335544540	integ_index_mod	Cannot modify index used by an Integrity Constraint
-616	335544541	check_trig_del	Cannot delete trigger used by a CHECK Constraint
-616	335544543	cnstrnt fld_del	Cannot delete column being used in an Integrity Constraint.
-616	335544630	dependency	There are %ld dependencies
-616	335544674	del_last_field	Last column in a table cannot be deleted
-616	335544728	integ_index_deactivate	Cannot deactivate index used by an Integrity Constraint

SQL	ISCCode	Symbol (GDSCODE)	Text
-616	335544729	integ_deactivate_primary	Cannot deactivate primary index
-617	335544542	check_trig_update	Cannot update trigger used by a CHECK Constraint
-617	335544544	cnstrnt_fld_rename	Cannot rename column being used in an Integrity Constraint.
-618	335544537	integ_index_seg_del	Cannot delete index segment used by an Integrity Constraint
-618	335544538	integ_index_seg_mod	Cannot update index segment used by an Integrity Constraint
-625	335544347	not_valid	Validation error for column %, value "%s"
-637	335544664	dsql_duplicate_spec	Duplicate specification of %s - not supported
-660	335544533	foreign_key_notfound	Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY.
-660	335544628	idx_create_err	Cannot create index %s
-663	335544624	idx_seg_err	Segment count of 0 defined for index %s
-663	335544631	idx_key_err	Too many keys defined for index %s
-663	335544672	key_field_err	Too few key columns found for index %s (incorrect column name?)
-664	335544434	keytoobig	Key size exceeds implementation restriction for index "%s"
-677	335544445	ext_err	%s extension error
-685	335544465	bad_segstr_type	Invalid BLOB type for operation
-685	335544670	blob_idx_err	Attempt to index BLOB column in index %s
-685	335544671	array_idx_err	Attempt to index array column in index %s
-689	335544403	badpagtyp	Page %ld is of wrong type (expected %ld, found %ld)
-689	335544650	page_type_err	Wrong page type
-690	335544679	no_segments_err	Segments not allowed in expression index %s
-691	335544681	rec_size_err	New record size of %ld bytes is too big
-692	335544477	max_idx	Maximum indexes per table (%d) exceeded
-693	335544663	req_max_clones_exceeded	Too many concurrent executions of the same request
-694	335544684	no_field_access	Cannot access column %s in view %s
-802	335544321	arith_except	Arithmetic exception, numeric overflow, or string truncation
-802	335544836	concat_overflow	Concatenation overflow. Resulting string cannot exceed 32K in length.
-803	335544349	no_dup	Attempt to store duplicate value (visible to active transactions) in unique index "%s"
-803	335544665	unique_key_violation	Violation of PRIMARY or UNIQUE KEY constraint "%s" on table "%s"
-804	335544380	wronumarg	Wrong number of arguments on call
-804	335544583	dsql_sqlda_err	SQLDA missing or incorrect version, or incorrect number/type of variables
-804	335544586	dsql_function_err	Function unknown

SQL	ISCCode	Symbol (GDSCODE)	Text
-804	335544713	dsql_sqlda_value_err	Incorrect values within SQLDA structure
-806	335544600	col_name_err	Only simple column names permitted for VIEW WITH CHECK OPTION
-807	335544601	where_err	No WHERE clause for VIEW WITH CHECK OPTION
-808	335544602	table_view_err	Only one table allowed for VIEW WITH CHECK OPTION
-809	335544603	distinct_err	DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION
-810	335544605	subquery_err	No subqueries permitted for VIEW WITH CHECK OPTION
-811	335544652	sing_select_err	Multiple rows in singleton select
-816	335544651	ext_readonly_err	Cannot insert because the file is readonly or is on a read only medium.
-816	335544715	extfile_uns_op	Operation not supported for EXTERNAL FILE table %s
-817	336003079	isc_sql_dialect_conflict_num	DB dialect %d and client dialect %d conflict with respect to numeric precision %d.
-817	335544361	read_only_trans	Attempted update during read-only transaction
-817	335544371	segstr_no_write	Attempted write to read-only BLOB
-817	335544444	read_only	Operation not supported
-817	335544765	read_only_database	Attempted update on read-only database
-817	335544766	must_be_dialect_2_and_up	SQL dialect %s is not supported in this database
-817	335544793	ddl_not_allowed_by_db_sql_dial	Metadata update statement is not allowed by the current database SQL dialect %d
-820	335544356	obsolete_metadata	Metadata is obsolete
-820	335544379	wrong_ods	Unsupported on-disk structure for file %s; found %ld, support %ld
-820	335544437	wrodynver	Wrong DYN version
-820	335544467	high_minor	Minor version too high found %ld expected %ld
-823	335544473	invalid_bookmark	Invalid bookmark handle
-824	335544474	bad_lock_level	Invalid lock level %d
-825	335544519	bad_lock_handle	Invalid lock handle
-826	335544585	dsql_stmt_handle	Invalid statement handle
-827	335544655	invalid_direction	Invalid direction for find operation
-827	335544718	invalid_key	Invalid key for find operation
-828	335544678	inval_key_posn	Invalid key position
-829	336068816	dyn_char_fld_too_small	New size specified for column %s must be at least %d characters.
-829	336068817	dyn_invalid_dtype_conversion	Cannot change datatype for %s. Conversion from base type %s to %s is not supported.

SQL	ISCCode	Symbol (GDSCODE)	Text
-829	336068818	dyn_dtype_conv_invalid	Cannot change datatype for column %s from a character type to a non-character type.
-829	335544616	field_ref_err	Invalid column reference
-830	335544615	field_aggregate_err	Column used with aggregate
-831	335544548	primary_key_exists	Attempt to define a second PRIMARY KEY for the same table
-832	335544604	key_field_count_err	FOREIGN KEY column count does not match PRIMARY KEY
-833	335544606	expression_eval_err	Expression evaluation not supported
-833	335544810	date_range_exceeded	Value exceeds the range for valid dates
-834	335544508	range_not_found	Refresh range number %ld not found
-835	335544649	bad_checksum	Bad checksum
-836	335544517	except	Exception %d
-837	335544518	cache_restart	Restart shared cache manager
-838	335544560	shutwarn	Database %s shutdown in %d seconds
-841	335544677	version_err	Too many versions
-842	335544697	precision_err	Precision must be from 1 to 18
-842	335544698	scale_nogt	Scale must be between zero and precision
-842	335544699	expec_short	Short integer expected
-842	335544700	expec_long	Long integer expected
-842	335544701	expec_ushort	Unsigned short integer expected
-842	335544712	expec_positive	Positive value expected
-901	336330753	gbak_unknown_switch	Found unknown switch
-901	335740929	gfix_db_name	Data base file name (%s) already given
-901	336920577	gstat_unknown_switch	Found unknown switch
-901	335544322	bad_dbkey	Invalid database key
-901	336330754	gbak_page_size_missing	Page size parameter missing
-901	335740930	gfix_invalid_sw	Invalid switch %s
-901	336920578	gstat_retry	Please retry, giving a database name
-901	336330755	gbak_page_size_toobig	Page size specified (%ld) greater than limit (8192 bytes)
-901	336920579	gstat_wrong_ods	Wrong ODS version, expected %d, encountered %d
-901	336330756	gbak_redir_output_missing	Redirect location for output is not specified
-901	335740932	gfix_incmp_sw	Incompatible switch combination
-901	336920580	gstat_unexpected_eof	Unexpected end of database file.
-901	336330757	gbak_switches_conflict	Conflicting switches for backup/restore
-901	335740933	gfix_replay_req	Replay log pathname required
-901	335544326	bad_dpb_form	Unrecognized database parameter block
-901	336330758	gbak_unknown_device	Device type %s not known
-901	335740934	gfix_pgbuf_req	Number of page buffers for cache required
-901	336330759	gbak_no_protection	Protection is not there yet

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	335740935	gfix_val_req	Numeric value required
-901	335544328	bad_segstr_handle	Invalid BLOB handle
-901	336330760	gbak_page_size_not_allowed	Page size is allowed only on restore or create
-901	335740936	gfix_pval_req	Positive numeric value required
-901	335544329	bad_segstr_id	Invalid BLOB ID
-901	336330761	gbak_multi_source_dest	Multiple sources or destinations specified
-901	335740937	gfix_trn_req	Number of transactions per sweep required
-901	335544330	bad_tpb_content	Invalid parameter in transaction parameter block
-901	336330762	gbak_filename_missing	Requires both input and output filenames
-901	335544331	bad_tpb_form	Invalid format for transaction parameter block
-901	336330763	gbak_dup_inout_names	Input and output have the same name. Disallowed.
-901	335544332	bad_trans_handle	Invalid transaction handle (expecting explicit transaction start)
-901	336330764	gbak_inv_page_size	Expected page size, encountered "%s"
-901	335740940	gfix_full_req	"full" or "reserve" required
-901	336330765	gbak_db_specified	REPLACE specified, but the first file %s is a database
-901	335740941	gfix_username_req	User name required
-901	336330766	gbak_db_exists	Database %s already exists. To replace it, use the -R switch
-901	335740942	gfix_pass_req	Password required
-901	336330767	gbak_unk_device	Device type not specified
-901	335740943	gfix_subs_name	Subsystem name
-901	336723983	gsec_cant_open_db	Unable to open database
-901	336723984	gsec_switches_error	Error in switch specifications
-901	335544337	excess_trans	Attempt to start more than %ld transactions
-901	335740945	gfix_sec_req	Number of seconds required
-901	336723985	gsec_no_op_spec	No operation specified
-901	335740946	gfix_nval_req	Numeric value between 0 and 32767 inclusive required
-901	336723986	gsec_no_usr_name	No user name specified
-901	335544339	infinap	Information type inappropriate for object specified
-901	335740947	gfix_type_shut	Must specify type of shutdown
-901	336723987	gsec_err_add	Add record error
-901	336330772	gbak_blob_info_failed	Gds_\$blob_info failed
-901	335544340	infona	No information of this type available for object specified
-901	335740948	gfix_retry	Please retry, specifying an option
-901	336723988	gsec_err_modify	Modify record error
-901	336330773	gbak_unk_blob_item	Do not understand BLOB INFO item %ld
-901	335544341	infunk	Unknown information item
-901	336723989	gsec_err_find_mod	Find/modify record error

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	336330774	gbak_get_seg_failed	Gds_\$get_segment failed
-901	335544342	integ_fail	Action cancelled by trigger (%ld) to preserve data integrity
-901	336723990	gsec_err_rec_not_found	Record not found for user: %s
-901	336330775	gbak_close_blob_failed	Gds_\$close_blob failed
-901	3357440951	gfix_retry_db	Please retry, giving a database name
-901	336723991	gsec_err_delete	Delete record error
-901	336330776	gbak_open_blob_failed	Gds_\$open_blob failed
-901	336723992	gsec_err_find_del	Find/delete record error
-901	336330777	gbak_put_blr_gen_id_failed	Failed in put_blr_gen_id
-901	335544345	lock_conflict	Lock conflict on no wait transaction
-901	336330778	gbak_unk_type	Data type %ld not understood
-901	336330779	gbak_comp_req_failed	Gds_\$compile_request failed
-901	336330780	gbak_start_req_failed	Gds_\$start_request failed
-901	336723996	gsec_err_find_disp	Find/display record error
-901	336330781	gbak_rec_failed	gds_\$receive failed
-901	336920605	gstat_open_err	Can't open database file %s
-901	336723997	gsec_inv_param	Invalid parameter, no switch defined
-901	336330782	gbak_rel_req_failed	Gds_\$release_request failed
-901	335544350	no_finish	Program attempted to exit without finishing database
-901	336920606	gstat_read_err	Can't read a database page
-901	336723998	gsec_op_specified	Operation already specified
-901	336330783	gbak_db_info_failed	gds_\$database_info failed
-901	336920607	gstat_sysmemex	System memory exhausted
-901	336723999	gsec_pw_specified	Password already specified
-901	336330784	gbak_no_db_desc	Expected database description record
-901	336724000	gsec_uid_specified	Uid already specified
-901	336330785	gbak_db_create_failed	Failed to create database %s
-901	335544353	no_recon	Transaction is not in limbo
-901	336724001	gsec_gid_specified	Gid already specified
-901	336330786	gbak_decomp_len_error	RESTORE: decompression length error
-901	336724002	gsec_proj_specified	Project already specified
-901	336330787	gbak_tbl_missing	Cannot find table %s
-901	335544355	no_segstr_close	BLOB was not closed
-901	336724003	gsec_org_specified	Organization already specified
-901	336330788	gbak_blob_col_missing	Cannot find column for BLOB
-901	336724004	gsec_fname_specified	First name already specified
-901	336330789	gbak_create_blob_failed	Gds_\$create_blob failed
-901	335544357	open_trans	Cannot disconnect database with open transactions (%ld active)

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	336724005	gsec_mname_specified	Middle name already specified
-901	336330790	gbak_put_seg_failed	Gds_\$put_segment failed
-901	335544358	port_len	Message length error (encountered %ld, expected %ld)
-901	336724006	gsec_lname_specified	Last name already specified
-901	336330791	gbak_rec_len_exp	Expected record length
-901	336330792	gbak_inv_rec_len	Wrong length record, expected %ld encountered %ld
-901	336724008	gsec_inv_switch	Invalid switch specified
-901	336330793	gbak_exp_data_type	Expected data attribute
-901	336724009	gsec_amb_switch	Ambiguous switch specified
-901	336330794	gbak_gen_id_failed	Failed in store_blr_gen_id
-901	336724010	gsec_no_op_specified	No operation specified for parameters
-901	336330795	gbak_unk_rec_type	Do not recognize record type %ld
-901	335544363	req_no_trans	No transaction for request
-901	336724011	gsec_params_not_allowed	No parameters allowed for this operation
-901	336330796	gbak_inv_bkup_ver	Expected backup version 1, 2, or 3. Found %ld
-901	335544364	req_sync	Request synchronization error
-901	336724012	gsec_incompat_switch	Incompatible switches specified
-901	336330797	gbak_missing_bkup_desc	Expected backup description record
-901	335544365	req_wrong_db	Request referenced an unavailable database
-901	336330798	gbak_string_trunc	String truncated
-901	336330799	gbak_cant_rest_record	warning -- record could not be restored
-901	336330800	gbak_send_failed	Gds_\$send failed
-901	336330801	gbak_no_tbl_name	No table name for data
-901	335544369	segstr_no_read	Attempted read of a new, open BLOB
-901	336330802	gbak_unexp_eof	Unexpected end of file on backup file
-901	335544370	segstr_no_trans	Attempted action on blob outside transaction
-901	336330803	gbak_db_format_too_old	Database format %ld is too old to restore to
-901	336330804	gbak_inv_array_dim	Array dimension for column %s is invalid
-901	335544372	segstr_wrong_db	Attempted reference to BLOB in unavailable database
-901	336330807	gbak_xdr_len_expected	Expected XDR record length
-901	335544376	unres_rel	Table %s was omitted from the transaction reserving list
-901	335544377	uns_ext	Request includes a DSRI extension not supported in this implementation
-901	335544378	wish_list	Feature is not supported
-901	335544382	random	%s
-901	335544383	fatal_conflict	Unrecoverable conflict with limbo transaction %ld
-901	335740991	gfix_exceed_max	Internal block exceeds maximum size
-901	335740992	gfix_corrupt_pool	Corrupt pool
-901	336330817	gbak_open_bkup_error	Cannot open backup file %s

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	335740993	gfix_mem_exhausted	Virtual memory exhausted
-901	336330818	gbak_open_error	Cannot open status and error output file %s
-901	335740994	gfix_bad_pool	Bad pool id
-901	335740995	gfix_trn_not_valid	Transaction state %d not in valid range.
-901	335544392	bdbincon	Internal error
-901	336724044	gsec_inv_username	Invalid user name (maximum 31 bytes allowed)
-901	336724045	gsec_inv_pw_length	Warning - maximum 8 significant bytes of password used
-901	336724046	gsec_db_specified	Database already specified
-901	336724047	gsec_db_admin_specified	Database administrator name already specified
-901	336724048	gsec_db_admin_pw_specified	Database administrator password already specified
-901	336724049	gsec_sql_role_specified	SQL role name already specified
-901	335741012	gfix_unexp_eoi	Unexpected end of input
-901	335544407	dbbnotzer	Database handle not zero
-901	335544408	tranotzer	Transaction handle not zero
-901	335741018	gfix_recon_fail	Failed to reconnect to a transaction in database %s
-901	335544418	trainlim	Transaction in limbo
-901	335544419	notinlim	Transaction not in limbo
-901	335544420	traoutsta	Transaction outstanding
-901	335544428	badmsgnum	Undefined message number
-901	335741036	gfix_trn_unknown	Transaction description item unknown
-901	335741038	gfix_mode_req	"read_only" or "read_write" required
-901	335544431	blocking_signal	Blocking signal has been received
-901	335544442	noargacc_read	Database system cannot read argument %ld
-901	335544443	noargacc_write	Database system cannot write argument %ld
-901	335544450	misc_interpreted	%s
-901	335544468	tra_state	Transaction %ld is %s
-901	335544485	bad_stmt_handle	Invalid statement handle
-901	336330934	gbak_missing_block_fac	Blocking factor parameter missing
-901	336330935	gbak_inv_block_fac	Expected blocking factor, encountered "%s"
-901	336330936	gbak_block_fac_specified	A blocking factor may not be used in conjunction with device CT
-901	336068796	dyn_role_does_not_exist	SQL role %s does not exist
-901	336330940	gbak_missing_username	User name parameter missing
-901	336068797	dyn_no_grant_admin_opt	User %s has no grant admin option on SQL role %s
-901	336330941	gbak_missing_password	Password parameter missing
-901	335544510	lock_timeout	Lock time-out on wait transaction
-901	336068798	dyn_user_not_role_member	User %s is not a member of SQL role %s
-901	336068799	dyn_delete_role_failed	%s is not the owner of SQL role %s
-901	336068800	dyn_grant_role_to_user	%s is a SQL role and not a user

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	336068801	dyn_inv_sql_role_name	User name %s could not be used for SQL role
-901	336068802	dyn_dup_sql_role	SQL role %s already exists
-901	336068803	dyn_kywd_spec_for_role	Keyword %s can not be used as a SQL role name
-901	336068804	dyn_roles_not_supported	SQL roles are not supported in on older versions of the database. A backup and restore of the database is required.
-901	336330952	gbak_missing_skipped_bytes	missing parameter for the number of bytes to be skipped
-901	336330953	gbak_inv_skipped_bytes	Expected number of bytes to be skipped, encountered "%s"
-901	336068820	dyn_zero_len_id	Zero length identifiers are not allowed
-901	336330965	gbak_err_restore_charset	Bad attribute for RDB\$CHARACTER_SETS
-901	336330967	gbak_err_restore_collation	Bad attribute for RDB\$COLLATIONS
-901	336330972	gbak_read_error	Unexpected I/O error while reading from backup file
-901	336330973	gbak_write_error	Unexpected I/O error while writing to backup file
-901	336330985	gbak_db_in_use	Could not drop database %s (database might be in use)
-901	336330990	gbak_sysmemex	System memory exhausted
-901	335544559	bad_svc_handle	Invalid service handle
-901	335544561	wrospbver	Wrong version of service parameter block
-901	335544562	bad_spb_form	Unrecognized service parameter block
-901	335544563	svcnotdef	Service %s is not defined
-901	336331002	gbak_restore_role_failed	Bad attributes for restoring SQL role
-901	336331005	gbak_role_op_missing	SQL role parameter missing
-901	336331010	gbak_page_buffers_missing	Page buffers parameter missing
-901	336331011	gbak_page_buffers_wrong_param	Expected page buffers, encountered "%s"
-901	336331012	gbak_page_buffers_restore	Page buffers is allowed only on restore or create
-901	336331014	gbak_inv_size	Size specification either missing or incorrect for file %s
-901	336331015	gbak_file_outof_sequence	File %s out of sequence
-901	336331016	gbak_join_file_missing	Can't join -- one of the files missing
-901	336331017	gbak_stdin_not_supptd	standard input is not supported when using join operation
-901	336331018	gbak_stdout_not_supptd	Standard output is not supported when using split operation
-901	336331019	gbak_bkup_corrupt	Backup file %s might be corrupt
-901	336331020	gbak_unk_db_file_spec	Database file specification missing
-901	336331021	gbak_hdr_write_failed	Can't write a header record to file %s
-901	336331022	gbak_disk_space_ex	Free disk space exhausted
-901	336331023	gbak_size_lt_min	File size given (%d) is less than minimum allowed (%d)
-901	336331025	gbak_svc_name_missing	Service name parameter missing
-901	336331026	gbak_not_ownr	Cannot restore over current database, must be SYSDBA or owner of the existing database.

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	336331031	gbak_mode_req	"read_only" or "read_write" required
-901	336331033	gbak_just_data	Just data ignore all constraints etc.
-901	336331034	gbak_data_only	Restoring data only ignoring foreign key, unique, not null & other constraints
-901	335544609	index_name	INDEX %s
-901	335544610	exception_name	EXCEPTION %s
-901	335544611	field_name	COLUMN %s
-901	335544613	union_err	Union not supported
-901	335544614	dsql_construct_err	Unsupported DSQL construct
-901	335544623	dsql_domain_err	Illegal use of keyword VALUE
-901	335544626	table_name	TABLE %s
-901	335544627	proc_name	PROCEDURE %s
-901	335544641	dsql_domain_not_found	Specified domain or source column %s does not exist
-901	335544656	dsql_var_conflict	Variable %s conflicts with parameter in same procedure
-901	335544666	svr_version_too_old	Server version too old to support all CREATE DATABASE options
-901	335544673	no_delete	Cannot delete
-901	335544675	sort_err	Sort error
-901	335544703	svcnoexe	Service %s does not have an associated executable
-901	335544704	net_lookup_err	Failed to locate host machine.
-901	335544705	service_unknown	Undefined service %s/%s.
-901	335544706	host_unknown	The specified name was not found in the hosts file or Domain Name Services.
-901	335544711	unprepared_stmt	Attempt to execute an unprepared dynamic SQL statement.
-901	335544716	svc_in_use	Service is currently busy: %s
-901	335544731	tra_must_sweep	
-901	335544740	udf_exception	A fatal exception occurred during the execution of a user defined function.
-901	335544741	lost_db_connection	Connection lost to database
-901	335544742	no_write_user_priv	User cannot write to RDB\$USER_PRIVILEGES
-901	335544767	blob_filter_exception	A fatal exception occurred during the execution of a blob filter.
-901	335544768	exception_access_violation	Access violation. The code attempted to access a virtual address without privilege to do so.
-901	335544769	exception_datatype_missalignment	Datatype misalignment. The attempted to read or write a value that was not stored on a memory boundary.
-901	335544770	exception_array_bounds_exceeded	Array bounds exceeded. The code attempted to access an array element that is out of bounds.
-901	335544771	exception_float_denormal_operand	Float denormal operand. One of the floating-point operands is too small to represent a standard float value.

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	335544772	exception_float_divide_by_zero	Floating-point divide by zero. The code attempted to divide a floating-point value by zero.
-901	335544773	exception_float_inexact_result	Floating-point inexact result. The result of a floating-point operation cannot be represented as a decimal fraction.
-901	335544774	exception_float_invalid_operand	Floating-point invalid operand. An indeterminate error occurred during a floating-point operation.
-901	335544775	exception_float_overflow	Floating-point overflow. The exponent of a floating-point operation is greater than the magnitude allowed.
-901	335544776	exception_float_stack_check	Floating-point stack check. The stack overflowed or underflowed as the result of a floating-point operation.
-901	335544777	exception_float_underflow	Floating-point underflow. The exponent of a floating-point operation is less than the magnitude allowed.
-901	335544778	exception_integer_divide_by_zero	Integer divide by zero. The code attempted to divide an integer value by an integer divisor of zero.
-901	335544779	exception_integer_overflow	Integer overflow. The result of an integer operation caused the most significant bit of the result to carry.
-901	335544780	exception_unknown	An exception occurred that does not have a description. Exception number %X.
-901	335544781	exception_stack_overflow	Stack overflow. The resource requirements of the runtime stack have exceeded the memory available to it.
-901	335544782	exception_sigsegv	Segmentation Fault. The code attempted to access memory without privileges.
-901	335544783	exception_sigill	Illegal Instruction. The Code attempted to perform an illegal operation.
-901	335544784	exception_sigbus	Bus Error. The Code caused a system bus error.
-901	335544785	exception_sigfpe	Floating Point Error. The Code caused an Arithmetic Exception or a floating point exception.
-901	335544786	ext_file_delete	Cannot delete rows from external files.
-901	335544787	ext_file_modify	Cannot update rows in external files.
-901	335544788	adm_task_denied	Unable to perform operation. You must be either SYSDBA or owner of the database
-901	335544794	cancelled	Operation was cancelled
-901	335544797	svcouser	User name and password are required while attaching to the services manager
-901	335544801	datatype_notsup	Data type not supported for arithmetic
-901	335544803	dialect_not_changed	Database dialect not changed.
-901	335544804	database_create_failed	Unable to create database %s
-901	335544805	inv_dialect_specified	Database dialect %d is not a valid dialect.
-901	335544806	valid_db_dialects	Valid database dialects are %s.
-901	335544811	inv_client_dialect_specified	Passed client dialect %d is not a valid dialect.
-901	335544812	valid_client_dialects	Valid client dialects are %s.

SQL	ISCCode	Symbol (GDSCODE)	Text
-901	335544814	service_not_supported	Services functionality will be supported in a later version of the product
-901	335544835	bad_shutdown_mode	Target shutdown mode is invalid for database "%s"
-902	335544333	bug_check	Internal gds software consistency check (%s)
-902	335544335	db_corrupt	Database file appears corrupt (%s)
-902	335544344	io_error	I/O error for file %.0s"%s"
-902	335544346	metadata_corrupt	Corrupt system table
-902	335544373	sys_request	Operating system directive %s failed
-902	335544384	badblk	Internal error
-902	335544385	invpoolcl	Internal error
-902	335544387	relbadblk	Internal error
-902	335544388	blktoobig	Block size exceeds implementation restriction
-902	335544394	badodsver	Incompatible version of on-disk structure
-902	335544397	dirtypage	Internal error
-902	335544398	waifortra	Internal error
-902	335544399	doubleloc	Internal error
-902	335544400	nodnotfnd	Internal error
-902	335544401	dupnodfnd	Internal error
-902	335544402	locnotmar	Internal error
-902	335544404	corrupt	Database corrupted
-902	335544405	badpage	Checksum error on database page %ld
-902	335544406	badindex	Index is broken
-902	335544409	trareqmis	Transaction--request mismatch (synchronization error)
-902	335544410	badhndcnt	Bad handle count
-902	335544411	wrotpbver	Wrong version of transaction parameter block
-902	335544412	wroblrver	Unsupported BLR version (expected %ld, encountered %ld)
-902	335544413	wrodpbver	Wrong version of database parameter block
-902	335544415	badrelation	Database corrupted
-902	335544416	nodetach	Internal error
-902	335544417	notremote	Internal error
-902	335544422	dbfile	Internal error
-902	335544423	orphan	Internal error
-902	335544432	lockmanerr	Lock manager error
-902	335544436	sqlerr	SQL error code = %ld
-902	335544448	bad_sec_info	
-902	335544449	invalid_sec_info	
-902	335544470	buf_invalid	Cache buffer for page %ld invalid
-902	335544471	indexnotdefined	There is no index in table %s with id %d

SQL	ISCCode	Symbol (GDSCODE)	Text
-902	335544472	login	Your user name and password are not defined. Ask your database administrator to set up a Firebird login.
-902	335544506	shutinprog	Database %s shutdown in progress
-902	335544528	shutdown	Database %s shutdown
-902	335544557	shutfail	Database shutdown unsuccessful
-902	335544569	dsql_error	Dynamic SQL Error
-902	335544653	psw_attach	Cannot attach to password database
-902	335544654	psw_start_trans	Cannot start transaction for password database
-902	335544717	err_stack_limit	Stack size insufficient to execute current request
-902	335544721	network_error	Unable to complete network request to host "%s".
-902	335544722	net_connect_err	Failed to establish a connection.
-902	335544723	net_connect_listen_err	Error while listening for an incoming connection.
-902	335544724	net_event_connect_err	Failed to establish a secondary connection for event processing.
-902	335544725	net_event_listen_err	Error while listening for an incoming event connection request.
-902	335544726	net_read_err	Error reading data from the connection.
-902	335544727	net_write_err	Error writing data to the connection.
-902	335544732	unsupported_network_drive	Access to databases on file servers is not supported.
-902	335544733	io_create_err	Error while trying to create file
-902	335544734	io_open_err	Error while trying to open file
-902	335544735	io_close_err	Error while trying to close file
-902	335544736	io_read_err	Error while trying to read from file
-902	335544737	io_write_err	Error while trying to write to file
-902	335544738	io_delete_err	Error while trying to delete file
-902	335544739	io_access_err	Error while trying to access file
-902	335544745	login_same_as_role_name	Your login %s is same as one of the SQL role name. Ask your database administrator to set up a valid Firebird login.
-902	335544791	file_in_use	The file %s is currently in use by another process. Try again later.
-902	335544795	unexp_spb_form	Unexpected item in service parameter block, expected %s
-902	335544809	extern_func_dir_error	Function %s is in %s, which is not in a permitted directory for external functions.
-902	335544819	io_32bit_exceeded_err	File exceeded maximum size of 2GB. Add another database file or use a 64 bit I/O version of Firebird.
-902	335544820	invalid_savepoint	Unable to find savepoint with name %s in transaction context
-902	335544831	conf_access_denied	Access to %s "%s" is denied by server administrator
-902	335544834	invalid_cursor_state	Invalid cursor state: %s
-904	335544324	bad_db_handle	Invalid database handle (no active connection)

SQL	ISCCode	Symbol (GDSCODE)	Text
-904	335544375	unavailable	Unavailable database
-904	335544381	imp_exc	Implementation limit exceeded
-904	335544386	nopoolids	Too many requests
-904	335544389	bufexh	Buffer exhausted
-904	335544391	bufinuse	Buffer in use
-904	335544393	reqinuse	Request in use
-904	335544424	no_lock_mgr	No lock manager available
-904	335544430	virtmemexh	Unable to allocate memory from operating system
-904	335544451	update_conflict	Update conflicts with concurrent update
-904	335544453	obj_in_use	Object %s is in use
-904	335544455	shadow_accessed	Cannot attach active shadow file
-904	335544460	shadow_missing	A file in manual shadow %ld is unavailable
-904	335544661	index_root_page_full	Cannot add index, index root page is full.
-904	335544676	sort_mem_err	Sort error: not enough memory
-904	335544683	req_depth_exceeded	Request depth exceeded. (Recursive definition?)
-904	335544758	sort_rec_size_err	Sort record size of %ld bytes is too big
-904	335544761	too_many_handles	Too many open handles to database
-904	335544792	service_att_err	Cannot attach to services manager
-904	335544799	svc_name_missing	The service name was not specified.
-904	335544813	optimizer_between_err	Unsupported field type specified in BETWEEN predicate.
-904	335544827	exec_sql_invalid_arg	Invalid argument in EXECUTE STATEMENT - cannot convert to string
-904	335544828	exec_sql_invalid_req	Wrong request type in EXECUTE STATEMENT '%s'
-904	335544829	exec_sql_invalid_var	Variable type (position %d) in EXECUTE STATEMENT '%s' INTO does not match returned column type
-904	335544830	exec_sql_max_call_exceeded	Too many recursion levels of EXECUTE STATEMENT
-904	335544832	wrong_backup_state	Cannot change difference file name while database is in backup mode
-906	335544744	max_att_exceeded	Maximum user count exceeded. Contact your database administrator.
-909	335544667	drdb_completed_with_errs	Drop database completed with errors
-911	335544459	rec_in_limbo	Record from transaction %ld is stuck in limbo
-913	335544336	deadlock	Deadlock
-922	335544323	bad_db_format	File %s is not a valid database
-923	335544421	connect_reject	Connection rejected by remote interface
-923	335544461	cant_validate	Secondary server attachments cannot validate databases
-923	335544464	cant_start_logging	Secondary server attachments cannot start logging
-924	335544325	bad_dpb_content	Bad parameters on attach or create database

SQL	ISCCode	Symbol (GDSCODE)	Text
-924	335544441	bad_detach	Database detach completed with errors
-924	335544648	conn_lost	Connection lost to pipe server
-926	335544447	no_rollback	No rollback performed
-999	335544689	ib_error	Firebird error

A P P E N D I X

XI

SQLSTATE CODES

From Firebird 2.5 forward, the ISO-standard SQLSTATE codes are supported. Table VIII.1 shows all of the SQLSTATE codes implemented as at Firebird 2.5.1.

The 5-character SQLSTATE code returned by the status array consists of SQL CLASS (2 characters) and SQL SUBCLASS (3 characters).

Where 1:1 mappings exist between the ISO-standard SQLSTATE code and an older SQLCODE, the SQLCODE is deprecated.



The SQL Standards Committee’s apparent failure to provide 1:1 mappings between SQLSTATE and SQLCODE was intentional. Their objective, for many years, has been to deprecate the SQLCODEs entirely.

Table VIII.1 SQLState Codes and Mappings

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
00–Success	00000	Success	
01–Warning	01000	General Warning	
	01001	Cursor operation conflict	
	01002	Disconnect error	
	01003	NULL value eliminated in set function	
	01004	String data right-truncated	
	01005	Insufficient item descriptor areas	
	01006	Privilege not revoked	
	01007	Privilege not granted	
	01008	Implicit zero-bit padding	
	01100	Statement reset to unprepared	
	01101	Ongoing transaction has been committed	

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
	01102	Ongoing transaction has been rolled back	
02—No Data	02000	No data found or no rows affected	
07—Dynamic SQL error	07000	Dynamic SQL error	
	07002	Wrong number of output parameters	
	07003	Cursor specification cannot be executed	
	07004	USING clause required for dynamic parameters	
	07005	Prepared statement not a cursor-specification	
	07006	Restricted data type attribute violation	
	07007	USING clause required for result fields	
	07008	Invalid descriptor coun	
	07009	Invalid descriptor index	
08—Connection Exception	08001	Client unable to establish connection	
	08002	Connection name in use	
	08003	Connection does not exist	
	08004	Server rejected the connection	
	08006	Connection failure	
	08007	Transaction resolution unknown	
0A—Feature Not Supported	0A000	Feature not supported	
0B—Invalid Transaction Initiation	0B000	Invalid transaction initiation	
0L—Invalid Grantor	0L000	Invalid grantor	
0P—Invalid Role Specification	0P000	Invalid role specification	
0V—Attempt to Assign to Ordering Column	0V000	Attempt to assign to ordering column	
20—Case Not Found For Case Statement	20000	Case not found for CASE statement	
21 (Cardinality Violation	21000	Cardinality violation	

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
21–Cardinality Violation	21S01	Insert value list does not match column list	
	21S02	Degree of derived table does not match column list	
22–Data Exception	22000	Data exception	
	22001	String data, right truncation	
	22002	Null value, no indicator parameter	
	22003	Numeric value out of range	
	22004	Null value not allowed	
	22005	Error in assignment	
	22006	Null value in field reference	
	22007	Invalid datetime format	
	22008	Datetime field overflow	
	22009	Invalid time zone displacement value	
	2200A	Null value in reference target	
	2200B	Escape character conflict	
	2200C	Invalid use of escape character	
	2200D	Invalid escape octet	
	2200E	Null value in array target	
	2200F	Zero-length character string	
	22010	Invalid indicator parameter value	
	22011	Substring error	
	22012	Division by zero	
	22014	Invalid update value	
	22015	Interval field overflow	
	22018	Invalid character value for cast	
	22019	Invalid escape character	
	2201B	Invalid regular expression	
	2201C	Null row not permitted in table	
	22020	Invalid limit value	
	22021	Character not in repertoire	
	22022	Indicator overflow	
	22023	Invalid parameter value	
	22024	Character string not properly terminated	
	22025	Invalid escape sequence	
	22026	String data, length mismatch	
	22027	Trim error	
	22028	Row already exists	
	2202D	Null instance used in mutator function	

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
22–Data Exception	2202E	Array element error	
	2202F	Array data, right truncation	
23–Integrity Constraint Violation	23000	Integrity constraint violation	
24–Invalid Cursor State	24000	Invalid cursor state	
	24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row	
25–Invalid Transaction State	25000	Invalid transaction state	
	25S01	Transaction state	
	25S02	Transaction is still active	
	25S03	Transaction is rolled back	
26–Invalid SQL Statement Name	26000	Invalid SQL statement name	
27–Triggered Data Change Violation	27000	Triggered data change violation	
28–Invalid Authorization Specification	28000	Invalid authorization specification	
2B–Dependent Privilege Descriptors Still Exist	2B000	Dependent privilege descriptors still exist	
2C–Invalid Character Set Name	2C000	Invalid character set name	
2D–Invalid Transaction Termination	2D000	Invalid transaction termination	
2E–Invalid Connection Name	2E000	Invalid connection name	
2F–SQL Routine Exception	2F000	SQL routine exception	
	2F002	Modifying SQL-data not permitted	

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
2F–SQL Routine Exception	2F003	Prohibited SQL-statement attempted	
	2F004	Reading SQL-data not permitted	
	2F005	Function executed no return statement	
33–Invalid SQL Descriptor Name	33000	Invalid SQL descriptor name	
34–Invalid Cursor Name	34000	Invalid cursor name	
35–Invalid Condition Number	35000	Invalid condition number	
36–Cursor Sensitivity Exception	36001	Request rejected	
	36002	Request failed	
37–Invalid Identifier	37000	Invalid identifier	
	37001	Identifier too long	
38–External Routine Exception	38000	External routine exception	
39–External Routine Invocation Exception	39000	External routine invocation exception	
3B–Invalid Save Point	3B000	Invalid save point	
3C–Ambiguous Cursor Name	3C000	Ambiguous cursor name	
3D–Invalid Catalog Name	3D000	Invalid catalog name	
	3D001	Catalog name not found	
3F–Invalid Schema Name	3F000	Invalid schema name	
40–Transaction Rollback	40000	Ongoing transaction has been rolled back	
	40001	Serialization failure	

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
40—Transaction Rollback	40002	Transaction integrity constraint violation	
	40003	Statement completion unknown	
42—Syntax Error or Access Violation	42000	Syntax error or access violation	
	42702	Ambiguous column reference	
	42725	Ambiguous function reference	
	42818	The operands of an operator or function are not compatible	
	42S01	Base table or view already exists	
	42S02	Base table or view not found	
	42S11	Index already exists	
	42S12	Index not found	
	42S21	Column already exists	
	42S22	Column not found	
44—With Check Option Violation	44000	WITH CHECK OPTION violation	
45—Unhandled User-defined Exception	45000	Unhandled user-defined exception	
54—Program Limit Exceeded	54000	Program limit exceeded	
	54001	Statement too complex	
	54011	Too many columns	
	54023	Too many arguments	
HY—CLI-specific Condition	HY000	CLI-specific condition	
	HY001	Memory allocation error	
	HY003	Invalid data type in application descriptor	
	HY004	Invalid data type	
	HY007	Associated statement is not prepared	
	HY008	Operation canceled	
	HY009	Invalid use of null pointer	
	HY010	Function sequence error	
	HY011	Attribute cannot be set now	
	HY012	Invalid transaction operation code	
	HY013	Memory management error	

SQLCLASS	SQLSTATE	Mapped Message	SQLCODE Equiv.
	HY014	Limit on the number of handles exceeded	
	HY015	No cursor name available	
	HY016	Cannot modify an implementation row descriptor	
	HY017	Invalid use of an automatically allocated descriptor handle	
	HY018	Server declined the cancellation request	
	HY019	Non-string data cannot be sent in pieces	
	HY020	Attempt to concatenate a null value	
	HY021	Inconsistent descriptor information	
	HY024	Invalid attribute value	
	HY055	Non-string data cannot be used with string routine	
	HY090	Invalid string length or buffer length	
	HY091	Invalid descriptor field identifier	
	HY092	Invalid attribute identifier	
	HY095	Invalid FunctionId specified	
	HY096	Invalid information type	
	HY097	Column type out of range	
	HY098	Scope out of range	
	HY099	Nullable type out of range	
	HY100	Uniqueness option type out of range	
	HY101	Accuracy option type out of range	
	HY103	Invalid retrieval code	
	HY104	Invalid length precision value	
	HY105	Invalid parameter type	
	HY106	Invalid fetch orientation	
	HY107	Row value out of range	
	HY109	Invalid cursor position	
	HY110	Invalid driver completion	
	HY111	Invalid bookmark value	
	HYC00	Optional feature not implemented	
	HYT00	Timeout expired	
	HYT01	Connection timeout expired	
XX—Internal Error	XX000	Internal error	
	XX001	Data corrupted	
	XX002	Index corrupted	

APPENDIX

XI

DATABASE REPAIR HOW-TO

This Appendix describes the steps you need to take, primarily using the command-line tools *gfix* and *gbak*, to try to identify and attempt to repair some kinds of corruption that might occur in a Firebird database.

Indications of Possible Corruption

Corruption in a Firebird database is far from being an everyday event since, unlike many other data management systems, its structure does not rely on dependencies between physical files. Untoward events in the operating environment can interfere with the normal flow of input and output in unpredictable ways that are beyond the control of the Firebird engine, however.

You will know a database has been corrupted if you cannot connect to it or back it up and a message in the Firebird log file, or from *gbak -b[ackup]*, tells you that something is corrupt or some other message reports a checksum error during an operation. You might also suspect corruption if a hard disk is throwing errors or if you are having trouble upgrading the on-disk structure in preparation for a version migration.

The evidence might appear during the database validation that you do as part of your routine housekeeping to detect minor anomalies and recycle misallocated space. You might have run a validation to check whether some misbehaviour could be associated with structural damage, perhaps prompted by one of the following circumstances:

- A “corrupt database” or “consistency check” error has appeared in the *firebird.log* or in a client application
- A backup has ended abnormally
- Power failure or brownout occurred without UPS protection or with suspected UPS failure, while “Forced Writes” was disabled
- Hard disk, network, or memory faults are suspected or have been reported by the operating system
- A database shadow has taken over from a dead database after a hard disk crash

- A production database is about to be moved to another platform or storage system
- A compromise of the network or database by malicious attack is suspected



Because the tools operate on the structures that are managed by the Firebird engine, there are certain kinds of corruption that this procedure cannot fix. For example, a deteriorating hard disk might have developed unreadable sectors and have physically destroyed blocks of disk surface that were storing pieces of the database. The Firebird tools might be able to make enough sense of what is left to report a message of doom, but physical recovery and reconstruction will be beyond their capabilities.

Preparing for Analysis and Repair

If you suspect you have a corrupt database, it is important to follow a proper sequence of recovery steps in order to avoid further corruption.

Because some of the steps make irreversible changes to structures within the database, it is essential to isolate the original database file from these operations and work with a file copy.

Take the database off-line as soon as evidence of a possible corruption appears and isolate the file to prevent any further connections. If it is not practicable to move the file then rename it.



Never attempt a recovery task by operating directly on the production database file.

Step 1—Get the database off-line

- Skip this step if the server is not running or is unable to accept connections.
- If the server is running and is still accepting connections then ask all users to cancel their work and log out. If necessary, use `gfix -force` with a sufficient timeout to let them try to get off gracefully.
- If logged-in users are unable to log out, the last resort will be to stop the server (Superserver or Superclassic) or kill the processes (Classic).

If the Superserver or Superclassic server is serving other databases, be sure to get those databases off-line before you stop the server.

Step 2—Make a working copy of the database

Once the database is off-line, take a file copy of it using a reliable filesystem utility and store the copy to disk in the location where you intend to work with it.

INSURANCE!

- Do not locate the copy in the same directory as the suspect database.
- If you suspect hard disk damage, make at least two copies and place them on a separate physical drive. Work on one and keep the other as a fall-back.
- Make sure enough space is available to accommodate the copied file.
- Do not start the analysis and repair steps until you are certain the copying operation has finished.
- Make sure you have sufficient disk space available to both create a backup and restore the backup to a new version of the database.

Step 3—Make an alias for the working database in `aliases.conf`. For example:

```
tempdb.fdb = /var/databases/copy1.fdb
```

Step 4—Set the environment variables `ISC_USER` and `ISC_PASSWORD`

This step is optional but, because the procedure involves the command-line utilities *gfix* and *gbak*, requiring repeated logins as SYSDBA, it will save some tedium.

How they are set depends on platform. For example, on Windows you can use the SET command in the command shell or the specialised applet under ControlPanel>Administration Tools>Advanced.

The arguments are:

```
ISC_USER=SYSDBA
```

```
ISC_PASSWORD=masterkey
```

but substituting the correct SYSDBA password, of course.



Having these variables set can create a security vulnerability. If you have other databases that need to be kept running while you are repairing this one, either

- *Take the long road: Skip this step and be prepared to do the typing; or*
- *Use Superuser or Administrator privileges to masquerade as SYSDBA. This is possible on Linux and configurable on Windows if the server is Firebird 2.1 or higher.*

Now you should be ready to begin the analysis and repair procedure.

Steps for Recovery Using Command-line Tools

Assumptions:

- you are set up so that you don't have to supply login credentials
- you have aliased your working copy (in `aliases.conf`) as `tempdb.fdb`

The steps described below involve use of privileged options of the command-line tools *gfix* and *gbak*. They will not work if you do not have SYSDBA or equivalent privileges. This procedure can mend some forms of corruption and return the database to a state wherein it becomes usable. It usually results in some data loss, since its strategy is to disable and remove troublesome records and, in some situations, even whole pages.

Step 1

Check for database corruption

The *gfix* switches `-v[alidate]` and `-f[ull]` are used first, to check record and page structures. This checking process reports corrupt structures and releases unassigned record fragments or orphan pages (i.e., pages that are allocated but unassigned to any data structures).

```
gfix -v -full tempdb.fdb {-user SYSDBA -password yourpwd}
```

The switch `-n[o_update]` can be used with `-v`, to validate and report on corrupt or misallocated structures without attempting to fix them:

```
gfix -v -n tempdb.fdb {-user SYSDBA -password yourpwd}
```

If persistent checksum errors are getting in the way of validation, include the `-i[gnore]` switch to have the validation ignore them:

```
gfix -v -n -i tempdb.fdb {-user SYSDBA -password yourpwd}
```

Outcome

- If errors were reported, the next step is for you.
- If the outcome of all this is that no errors were reported, SKIP the next two steps: attempting to mend databases without broken structures can actually create breakages.

Step 2

Mend corrupted pages

If `gfix` validation reports damage to data, the next step is to mend (or repair) the database by putting those structures out of the way. Along with the `-m[end]` switch, include these switches:

- a `-f[ull]` switch to request mend for all corrupt structures
 - an `-i[gnore]` switch to bypass checksum errors during the mend
- ```
gfix -mend -full -ignore tempdb.fdb {-user SYSDBA -password yourpwd}
```

or, briefly

```
gfix -m -f -i tempdb.fdb {-user SYSDBA -password yourpwd}
```

## Outcome

The `-m[end]` switch marks corrupt records as unavailable, so they will be skipped during a subsequent backup. It is likely, therefore, that some data will be lost.

## Step 3

## Validate after `-mend`

After the `-mend` command completes its work, again do a full validation to check whether any corrupt structures remain:

```
gfix -v -full tempdb.fdb {-user SYSDBA -password yourpwd}
```

## Step 4

## Clean and recover the database

Whether you skipped steps 2 and 3 or not, now do a full backup and restore using *gbak*, even if errors are still being reported. Include the `-v[erify]` switch to watch progress. In its simplest form, the backup command would be (all in one command):

```
gbak -b -v -i tempdb.fdb {path}tempdb.fbk
```

The following should help to resolve some of the complications that might show up during *gbak*:

- 1 Garbage collection problems might cause *gbak* to fail.

If this happens, run it again, adding the `-g` switch to signify “no garbage collection”:

```
gbak -b -v -i -g tempdb.fdb {path}tempdb.fbk
```

- 2 Corruption in limbo transactions

If there is corruption in record versions associated with a limbo transaction, you may need to include the `-l[imbo]` switch:

```
gbak -b -v -i -g -l {path}tempdb.fbk
```

## Step 5 Restore the cleaned backup as a new database

---

Now create a new database from the backup, using the `-v[erify]` switch to watch what is being restored:

```
gbak -create -v {path}tempdb.fbk {path}reborn.fdb
```

If the restore throws up further errors, you may need to consider further attempts using `gbak -c` with other switches to eliminate the sources of these problems. For example:

- 1 Restore with inactive indexes. The `-i[nactive]` switch will eliminate problems with damaged indexes, by restoring without activating any indexes. Afterwards, you can activate the indexes manually, one at a time, until the problem index is found.
- 2 Restore tables one at a time. The `-o[ne_at_a_time]` switch will restore and commit each table, one by one, allowing you restore good tables and bypass the problem ones.

## Step 6 Validate the restored database

---

Verify that restoring the database fixed the problems by validating the restored database with the `-n[o_update]` switch:

```
gfix -v -full {path}reborn.fdb {-user SYSDBA -password yourpassword}
```

### Outcome

If identifiable damage to page and record structures were found and fully eliminated by this procedure, you should now be able to log in to the “reborn” database.

All that remains now is to store the good backup in a safe place, move the repaired copy back to its proper directory and rename it.

## Failed Repair

---

If the preceding steps do not work, but you are still able to access the copy of the corrupt database, you may still be able to transfer table structures and data from the damaged database to a new one, a technique known as data pumping. Several free and commercial tools are available. Look in the Downloads section of the website <http://www.ibphoenix.com> for candidates, including free ones.

Data pumping can also be done using the *qli* (Query Language Interpreter) command line tool that can be found in the `/bin` directory of a Firebird 1.5 installation kit (from the Firebird Downloads pages at <http://firebirdsql.org>). The IBPhoenix website has a How-To article for this task at [http://www.ibphoenix.com/resources/documents/how\\_to/doc\\_42](http://www.ibphoenix.com/resources/documents/how_to/doc_42).

Commercial tools for rescuing data from broken Firebird databases are available from IBSurgeon: <http://www.ib-aid.com>.





A P P E N D I X



# DEFAULT DISK LOCATIONS

The tables in this section describe the default disk locations for the components on Windows, POSIX and Mac OSX/Darwin, along with a few of the distro-specific packages.

Tables X.1, X.2 and X.3 show where to look for the components of the standard server installations after running an installer. Table X.4 shows the typical locations for client libraries on workstations. The exact locations may change from release to release.

In Table X.5 the distro-specific packaging of the Firebird 2.5 components for the Fedora/Mageia extended family is shown, with some notes about the Debian family packaging.

## Linux and Some UNIX

**Table X.1** Firebird Installation on Linux and Some UNIX Platforms

| Component                                                                  | File Name                        | Default Location  |
|----------------------------------------------------------------------------|----------------------------------|-------------------|
| Classic server                                                             | fb_inet_server                   | /opt/firebird/bin |
| Superclassic server <sup>1</sup>                                           | fb_smp_server                    | /opt/firebird/bin |
| Lock Manager program (Classic and Superclassic servers)                    | fb_lock_mgr                      | /opt/firebird/bin |
| Embedded client for Classic server                                         | libfbembed.so (symlink)          | /opt/firebird/lib |
|                                                                            | libfbembed.so.2 (symlink)        | /opt/firebird/lib |
|                                                                            | libfbembed.so.2.5.n <sup>2</sup> |                   |
| Guardian (Superserver only)                                                | fbguard                          | /opt/firebird/bin |
| Superserver                                                                | fbserver                         | /opt/firebird/bin |
| Thread-safe remote client for Superserver, Superclassic and Classic server | libfbclient.so (symlink)         | /opt/firebird/lib |
|                                                                            | libfbclient.so.2 (symlink)       | /opt/firebird/lib |

| Component                                                               | File Name                                                                                                                                    | Default Location              |
|-------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
|                                                                         | libfbclient.so.2.5.n                                                                                                                         | /opt/firebird/lib             |
| Configuration file                                                      | firebird.conf                                                                                                                                | /opt/firebird                 |
| Database alias file                                                     | aliases.conf                                                                                                                                 | /opt/firebird                 |
| Message lookup file                                                     | firebird.msg, fr_FR.msg,<br>de_DE.msg, et al.                                                                                                | /opt/firebird                 |
| Generated password file                                                 | SYSDBA.password                                                                                                                              | /opt/firebird                 |
| Security database                                                       | security2.fdb <sup>3</sup>                                                                                                                   | /opt/firebird                 |
| Backup of security database                                             | security.fbk <sup>4</sup>                                                                                                                    | /opt/firebird                 |
| Command-line tools                                                      | isql, gbak, gfix, gstat, gsec,<br>gdef, gpre, qli,<br>fb_lock_print, fbvcmgr <sup>5</sup> ,<br>fbtracmgr <sup>6</sup> , nbackup <sup>7</sup> | /opt/firebird/bin             |
| Trace and audit plug-in                                                 | libfbtrace.so                                                                                                                                | /opt/firebird/plugins         |
| Server tool (Superserver only)                                          | fbmgr                                                                                                                                        | /opt/firebird/bin             |
| Template init script for Firebird (Classic<br>server only) <sup>8</sup> | firebird.xinetd                                                                                                                              | opt/firebird/misc             |
| External function libraries (UDF libraries)                             | ib_udf.so, fbudf.so                                                                                                                          | opt/firebird/UDF <sup>9</sup> |
| DDL scripts for external function<br>libraries                          | ib_udf.sql, ub_udf2.sql <sup>10</sup> ,<br>fbudf.sql                                                                                         | opt/firebird/UDF              |
| Memory utility library                                                  | libib_util.so                                                                                                                                | /opt/firebird/lib             |
| International language support plug-in                                  | fbintl                                                                                                                                       | opt/firebird/intl             |
| Configuration template file for character<br>sets and collations        | fbintl.conf                                                                                                                                  | opt/firebird/intl             |
| <b>ICU 3.0 international language libraries—v.2.1 + only</b>            |                                                                                                                                              |                               |
| ICU character sets data library                                         | libicudata.so (symlink)                                                                                                                      | /opt/firebird./lib            |
|                                                                         | libicudata.so.30 (symlink)                                                                                                                   | /opt/firebird./lib            |
|                                                                         | licicudata.so.30.0                                                                                                                           | /opt/firebird./lib            |
| ICU 18N library                                                         | libicu18n.so (symlink)                                                                                                                       | /opt/firebird./lib            |
|                                                                         | libicu18n.s0.30 (symlink)                                                                                                                    | /opt/firebird./lib            |
|                                                                         | libicu18n.so.30.0                                                                                                                            | /opt/firebird./lib            |
| <b>Shell scripts</b>                                                    |                                                                                                                                              |                               |
| changeDBAPassword.sh                                                    | Simple interactive script for<br>changing the SYSDBA<br>password                                                                             | /opt/firebird/bin             |
| changeGDSLibraryCompatibleLin.sh                                        |                                                                                                                                              | ditto                         |
| changeRunUser.sh                                                        |                                                                                                                                              | ditto                         |
| createAliasDB.sh                                                        |                                                                                                                                              | /opt/firebird/bin             |
| FirebirdUninstall.sh                                                    |                                                                                                                                              | ditto                         |
| restoreRootRunUser.sh                                                   |                                                                                                                                              | ditto                         |

| Component          | File Name                                                                                                                      | Default Location                   |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------|------------------------------------|
| User documentation | Release notes <sup>11</sup> , Language Update and Quick Start Guide (all PDF); and README files on various topics (text files) | opt/firebird/doc                   |
| Sample database    | employee.fdb                                                                                                                   | opt/firebird/example<br>s/empbuild |
| C header files     | ibase.h, iberror.h, and others                                                                                                 | opt/firebird/include               |

1. V.2.5 and higher

2. n = sub-release number

3. security.fdb in v.1.5

4. In v.1.5 only

5. v.2.5 and higher

6. ditto

7. v.2.0+

8. Dropped in later versions

9. Check UdfAccess parameter in firebird.conf: by default it is NONE.

10. Declarations for null-enabled UDFs in v.2.0 +

11. In v.2.1.X, Bug fixes and Installation and Migration topics are distributed in PDF files separately from the release notes, in the same directory. In older releases, release notes were often distributed in /opt/firebird.

## Microsoft Windows

**Table X.2** Firebird Installation on Windows Platforms (32-bit and 64-bit)

| Component                                                      | File Name                                 | Default Location                                                       |
|----------------------------------------------------------------|-------------------------------------------|------------------------------------------------------------------------|
| Classic/Superclassic server                                    | fb_inet_server.exe                        | C:\Program<br>Files\Firebird\Firebird_n_n\bin <sup>1</sup>             |
| Lock Manager program<br>(Classic/Superclassic servers<br>only) | fb_lock_mgr.exe                           | C:\Program<br>Files\Firebird\Firebird_n_n\bin                          |
| Guardian (Superserver only)                                    | fbguard.exe                               | C:\Program<br>Files\Firebird\Firebird_n_n\bin                          |
| Superserver                                                    | fbserver.exe                              | C:\Program<br>Files\Firebird\Firebird_n_n\bin                          |
| Embedded                                                       | fbembed.dll                               | Install to application root <sup>2</sup> and rename to<br>fbclient.dll |
| Client library for all server<br>models except embedded        | fbclient.dll or<br>gds32.dll <sup>3</sup> | C:\Program<br>Files\Firebird\Firebird_n_n\bin <sup>4</sup>             |
| Configuration file                                             | firebird.conf                             | C:\Program Files\Firebird\Firebird_n_n                                 |
| Database alias file                                            | aliases.conf                              | C:\Program Files\Firebird\Firebird_n_n                                 |
| Message lookup file                                            | firebird.msg                              | C:\Program Files\Firebird\Firebird_n_n                                 |
| Security database                                              | security2.fdb <sup>5</sup>                | C:\Program Files\Firebird\Firebird_n_n                                 |
| Backup of security database <sup>6</sup>                       | security.fbk                              | C:\Program Files\Firebird\Firebird_1_5                                 |

| Component                                                                       | File Name                                                                                                       | Default Location                                          |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| Command-line tools                                                              | EXEs: isql, gbak, gfix, gstat, gsec, gdef, gpre, qli, fb_lock_print, fbsvcmgr, fbtracemgr                       | C:\Program Files\Firebird\Firebird_n_n\bin                |
| Services and registration tools, including client library installation utility  | instsvc.exe <sup>7</sup> , instreg.exe, instclient.exe                                                          | C:\Program Files\Firebird\Firebird_n_n\bin                |
| External function libraries (UDF libraries)                                     | ib_udf.dll, fbudf.dll                                                                                           | C:\Program Files\Firebird\Firebird_n_n\UDF <sup>8</sup>   |
| Memory utility library (used by ib_udf.dll and some linked component libraries) | ib_util.dll                                                                                                     | C:\Program Files\Firebird\Firebird_n_n\bin                |
| DDL scripts for external function libraries                                     | ib_udf.sql, ub_udf2.sql <sup>9</sup> , fbudf.sql                                                                | C:\Program Files\Firebird\Firebird_n_n\UDF                |
| International language support library                                          | fbintl.dll                                                                                                      | C:\Program Files\Firebird\Firebird_n_n\intl               |
| Configuration file for character sets and collations                            | fbintl.conf                                                                                                     | C:\Program Files\Firebird\Firebird_n_n\intl               |
| ICU support library                                                             | icuin30.dll                                                                                                     | C:\Program Files\Firebird\Firebird_n_n\bin                |
| User documentation                                                              | Release notes, Language Update and Quick Start Guide (all PDF); and README files on various topics (text files) | C:\Program Files\Firebird\Firebird_n_n\doc                |
| Sample database                                                                 | employee.fdb                                                                                                    | C:\Program Files\Firebird\Firebird_n_n\examples\emp build |
| C header files                                                                  | ibase.h, iberror.h, and others                                                                                  | C:\Program Files\Firebird\Firebird_n_n\include            |
| Utility library related to GPRE support for RM Cobol.                           | fbrmclib.dll                                                                                                    | C:\Program Files\Firebird\Firebird_n_n\bin                |

1. n\_n represents major and minor release numbers, e.g. 2\_5 for v.2.5. Sub-release numbers are not used in folder names.
2. Application root = folder where the user application resides. If installing Firebird utilities such as gbak.exe and isql.exe, install them to this directory as well.
3. For compatibility with old Borland components, fbclient.dll may be optionally generated as gds32.dll through the Windows installer kit or by running instclient.exe. For 64-bit issues, refer to Note 4.
4. Using the same routines as in the previous note, either client library can be copied to the system folder. Be aware that, on 64-bit installations, this will be the 64-bit library, not accessible to 32-bit applications.
5. Security.fdb in v.1.5.
6. In v.1.5 only
7. In v.2.5+, use this utility to convert a Classic installation to Superclassic, or vice versa.
8. Check UdfAccess parameter in firebird.conf: by default it is NONE.
9. Declarations for null-enabled UDFs in v.2.0 +

## Mac OSX and Darwin

For the server installations, everything is stored in a Mac Framework located at `/Library/Frameworks/Firebird.framework/`. Mac Frameworks are described in detail at the [Mac OSX Developer web site](#).

Components inside `/Library/Frameworks/Firebird.framework`:

**Table X.3** Firebird 2.5 installation on Mac OSX and Darwin

| Component                                                                           | File Name                                                                                                  | Default Location        |
|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|-------------------------|
| Classic server                                                                      | fb_inet_server                                                                                             | /Resources/bin          |
| Superclassic server <sup>1</sup>                                                    | fb_smp_server                                                                                              | /Resources/bin          |
| Lock Manager program<br>(Classic and Superclassic<br>servers)                       | fb_lock_mgr                                                                                                | /Resources/bin          |
| Superclassic enabling script                                                        | changeMultiConnect<br>Mode.sh                                                                              | /Resources/bin          |
| <b>OR</b>                                                                           |                                                                                                            |                         |
| Guardian (Superserver<br>only)                                                      | fbguard                                                                                                    | /Resources/bin          |
| Superserver                                                                         | fbserver                                                                                                   | /Resources/bin          |
| <hr/>                                                                               |                                                                                                            |                         |
| Command-line tools                                                                  | isql, gbak, gfix, gstat,<br>gsec, gdef, gpre, qli,<br>fb_lock_print,<br>fbsvcmgr, fbtracemgr               | /Resources/bin          |
| “Root” files                                                                        | aliases.conf,<br>firebird.conf,<br>fbtrace.conf,<br>security2.fdb and also<br>a link to<br>./Resources/bin | /English.lproj/var      |
| External function libraries<br>(UDF libraries)                                      | ib_udf.dylib,<br>fbudf.dylib                                                                               | /English.lproj/var/UDF  |
| International language<br>support plug-in                                           | fbintl                                                                                                     | /English.lproj/var/intl |
| Configuration template file<br>for character sets and<br>collations                 | fbintl.conf                                                                                                | /English.lproj/var/intl |
| Embedded client for<br>Classic server                                               | libfbembed.dylib <sup>2</sup>                                                                              | /Libraries              |
| Thread-safe remote client<br>for Superserver,<br>Superclassic and Classic<br>server | libfbclient.dylib <sup>3</sup>                                                                             | /Libraries              |

| Component                               | File Name                                                                                                       | Default Location    |
|-----------------------------------------|-----------------------------------------------------------------------------------------------------------------|---------------------|
| User documentation                      | Release notes, Language Update and Quick Start Guide (all PDF); and README files on various topics (text files) | /Resources/doc      |
| Sample database and other example files | employee.fdb                                                                                                    | /Resources/examples |
| C header files                          | ibase.h, iberror.h, and others                                                                                  | /Headers            |

1. V.2.5 and higher
2. Linked as Firebird.framework/Firebird which is a link to Firebird.framework/Versions/Current/Firebird. See also Note 4 in Table X.4, Names and default locations of Firebird clients.
3. Ditto

Client Libraries

Table X.4 Names and default locations of Firebird clients

| Client OS           | Connecting to                                                                              | Library name                                                                               | Default Location of Client |
|---------------------|--------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|----------------------------|
| Linux and some UNIX |                                                                                            | Firebird “2” series                                                                        |                            |
| Embedded client     | Embedded Superclassic (local only) in a thread                                             | libfbembed.so (symlink)<br>→libfbembed.so.2 (symlink)<br>→libfbembed.so.2.5.n <sup>1</sup> | /opt/firebird/lib          |
| Remote client       | Any remote Superserver, Superclassic or Classic server, including on local loopback server | libfbclient.so (symlink)<br>→libfbclient.so.2 (symlink)<br>→libfbclient.so.2.5.n           | /opt/firebird/lib          |
| Linux and some UNIX |                                                                                            | Firebird 1.5                                                                               |                            |
| Embedded client     | Embedded Classic, non-threaded, direct to database                                         | libfbembed.so (symlink)<br>→libfbembed.so.0                                                | /usr/lib                   |
| Remote client       | Any remote Classic or Superserver                                                          | libfbclient.so (symlink)<br>→libfbclient.so.0,                                             | /usr/lib                   |
| Linux and some UNIX |                                                                                            | Firebird 1.0                                                                               |                            |
| Embedded client     | Embedded v.1.0 Classic, non-threaded, direct to database                                   | libgds.so (symlink)<br>→libgds.so.0                                                        | /usr/lib                   |
| Remote client       | Any remote Classic or Superserver                                                          | libgds.so (symlink)<br>→libgds.so.0 <sup>2</sup>                                           | /usr/lib                   |

| Client OS                                     | Connecting to                                                                                | Library name                                                                                          | Default Location of Client               |
|-----------------------------------------------|----------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|------------------------------------------|
| <b>Any Windows from XP/Server 2003 onward</b> |                                                                                              | <b>Firebird “2” series</b>                                                                            |                                          |
| Remote client                                 | Any server of matching or lower version on any platform                                      | Native: fbclient.dll                                                                                  | \$firebird\$\bin                         |
|                                               |                                                                                              | Compatibility: fbclient.dll or gds32.dll built by instclient.exe                                      | c:\windows\system32                      |
| <b>Windows NT/2K<sup>3</sup></b>              | ditto                                                                                        | Native: fbclient.dll                                                                                  | \$firebird\$\bin                         |
|                                               |                                                                                              | Compatibility: fbclient.dll or gds32.dll built by instclient.exe                                      | c:\winnt\system32                        |
| <b>Any Windows from XP/Server 2003 onward</b> |                                                                                              | <b>Firebird 1.5</b>                                                                                   |                                          |
| Embedded client                               | Embedded Superserver; but will behave as fbclient.dll if used as client to a remote database | fbembed.dll, renamed as required to fbclient.dll or gds32.dll                                         | Root directory of application executable |
| <b>Windows XP/Server 2003</b>                 |                                                                                              | Native: fbclient.dll                                                                                  | \$firebird\$\bin                         |
|                                               |                                                                                              | Compatibility: fbclient.dll or gds32.dll built by instclient.exe                                      | c:\windows\system32                      |
| <b>Windows NT/2K</b>                          | Any 1.5 server                                                                               | Native: fbclient.dll                                                                                  | \$firebird\$\bin                         |
|                                               | Any 1.5 server                                                                               | Compatibility: fbclient.dll or gds32.dll built by instclient.exe                                      | c:\winnt\system32                        |
| <b>Windows 9x/ME</b>                          | Any 1.5 server                                                                               | Native: fbclient.dll                                                                                  | \$firebird\$\bin                         |
|                                               | Any 1.5 server                                                                               | Compatibility: fbclient.dll or gds32.dll built by instclient.exe                                      | c:\windows or c:\windows\system          |
| <b>Windows XP/Server 2003</b>                 |                                                                                              | <b>Firebird 1.0</b>                                                                                   |                                          |
|                                               | Any v.1.0 server                                                                             | gds32.dll                                                                                             | c:\windows\system32                      |
| <b>Windows NT/2K</b>                          | ditto                                                                                        | gds32.dll                                                                                             | c:\winnt\system32                        |
| <b>Windows 9x/ME</b>                          | ditto                                                                                        | gds32.dll                                                                                             | c:\Windows                               |
| <b>Mac OSX</b>                                | Any version-compatible Firebird server                                                       | libfbclient.dylib for remote clients, libfbembed.dylib for direct local, Classic only, i.e., embedded | See note <sup>4</sup>                    |

1. n = sub-release number

2. This library is different to the libgds.so client in Classic

3. Windows NT 4, 2000, ME and 9X are not officially supported in the “2” series

4. If you look for *libfbembedded.dylib* (Classic) and *libfbclient.dylib* (SuperServer and Superclassic) by those names you won't find them. They are enclosed within the `/Library/Frameworks/Firebird.framework` in `/Libraries/`. They become linked as `Firebird.framework/Firebird` which is a link to `Firebird.framework/Versions/Current/Firebird`.

Fedora/Mageia, Debian and related distributions

Table X.5 shows how the components are distributed in the Firebird 2.5 packages under Mageia, Mandriva, Fedora, Red Hat Enterprise Linux (RHEL) and derivatives.

**Table X.5**    Distribution-specific packaging of Firebird 2.5 (Fedora, Mageia, etc., also Debian family)

| Component                                                                                                                                                  | Location                                                                        | Notes                                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Firebird root directory                                                                                                                                    | <code>/usr/lib/firebird</code> or<br><code>/usr/lib64/firebird</code>           | Debian family doesn't have<br><code>/usr/lib64/</code> for 64 bits. It always uses<br><code>/usr/lib/</code>                                                                                       |
| Main binaries: <code>fbguard</code> ,<br><code>fb_inet_server</code> ,<br><code>fb_smp_server</code> ,<br><code>fbserver</code> <code>fb_lock_print</code> | <code>/usr/sbin</code>                                                          |                                                                                                                                                                                                    |
| Other binaries                                                                                                                                             | <code>/usr/bin</code>                                                           | <i>isql</i> is named <i>isql-fb</i> and <i>gstat</i> is <i>gstat-fb</i><br>due to name conflicts with other<br>software<br><br>Under Debian and derivatives <i>gstat</i> is<br>named <i>fbstat</i> |
| Client libraries                                                                                                                                           | <code>/usr/lib</code> or <code>/usr/lib64</code>                                | Debian family doesn't have<br><code>/usr/lib64/</code> for 64 bits. It always uses<br><code>/usr/lib/</code>                                                                                       |
| Configuration (.conf) files                                                                                                                                | <code>/etc/firebird</code>                                                      | Debian uses the version for some files<br>so the conf files are in<br><code>/etc/firebird2.5/</code> , for example                                                                                 |
| External function (UDF)<br>libraries                                                                                                                       | <code>/usr/lib/firebird/UDF</code> or<br><code>/usr/lib64/firebird/UDF</code>   | Debian family doesn't have<br><code>/usr/lib64/</code> for 64 bits. It always uses<br><code>/usr/lib/</code>                                                                                       |
| International language<br>support plug-in (fbintl)                                                                                                         | <code>/usr/lib/firebird/intl</code> or<br><code>/usr/lib64/firebird/intl</code> | Debian family doesn't have<br><code>/usr/lib64/</code> for 64 bits. It always uses<br><code>/usr/lib/</code>                                                                                       |
| <code>security2.fdb</code> , <code>firebird.msg</code>                                                                                                     | <code>/var/lib/firebird/system</code>                                           |                                                                                                                                                                                                    |
| <code>firebird.log</code>                                                                                                                                  | <code>/var/log/firebird</code>                                                  |                                                                                                                                                                                                    |
| Lock files                                                                                                                                                 | <code>/var/run/firebird</code>                                                  |                                                                                                                                                                                                    |
| C headers                                                                                                                                                  | <code>/usr/include</code>                                                       |                                                                                                                                                                                                    |
| Documentation                                                                                                                                              | <code>/usr/share/doc/firebird</code>                                            |                                                                                                                                                                                                    |
| Sample database<br><code>employee.fdb</code> and other<br>examples                                                                                         | <code>/var/lib/firebird/data</code>                                             |                                                                                                                                                                                                    |



## APPENDIX

## XI

# HEALTHCARE FOR DATABASES

Those who have turned to Firebird after having been involved in caring for databases under other database engines are often dumbfounded by how much they *don't* have to do to keep their databases healthy and well-performing. However, Firebird's admin-friendliness can be a temptation to neglect just the very tasks that enable it to keep being so friendly.

In this article, we look at the essentials you need to consider when defining the administration procedures for your production servers.

## Considerations for a Maintenance Regime

---

A maintenance regime will be very particular to the conditions under which the databases are deployed. The five main areas for attention are backups, garbage collection, index statistics, page size and disk usage.

### 1 Backups

---

It goes without saying that databases must be backed up. Firebird has two utilities that have backup capabilities: *gbak* and *nbackup*. One of these utilities must be used for backing up databases. Do not plan to include databases in any backups that are performed using other tools unless there is absolute certainty that the databases are fully shut-down, the files all closed and there is no chance that any user could log on.

#### **gbak**

The *gbak* utility is run from a command shell, either directly from the command line or from a user-written shell script. The same operations are available to applications through the Services API. Because *gbak -backup* works by first saving the metadata of both user-created and system-created objects and then extracting, compressing and saving all the data, it literally touches everything in the database. Unless you specify otherwise, it cleans

up garbage as it goes. Because of this, a regime of regular *gbak* backups is not just failure insurance: it can play an important role also in maintaining database health and good performance.

## nbackup

The *nbackup* utility works quite differently to *gbak*. It takes page-level snapshots of the physical image on disk. It enables incremental backups for very large systems where frequent *gbak* backups are impracticable. It does not tidy up nor reorganise anything at record level: any garbage awaiting collection is backed up along with active and other “interesting” data.

## 2 Garbage Collection

---

Firebird uses a “multi-generational architecture” (MGA) to manage multi-user concurrency and isolate each transaction’s view of database state. The MGA mechanism is discussed in detail elsewhere. Its effect, in short, is that older versions of updated records and the stubs of deleted records remain physically in place until every transaction that had an interest in the old version has finished. After that, when the engine performs “garbage collection” to clear out obsolete records and stubs that are no longer “interesting”, the physical space they occupied on pages becomes available for storing new records and record versions.

### Garbage Collection Policy

Firebird takes care of garbage in several ways. The process known as “garbage collection” (GC) goes on continually. On Superserver, it runs as a worker thread, in the background. On other models, GC occurs as transactions touch records that have been committed by other transactions and are no longer interesting to any transactions. Because transactions are cleaning up after other transactions, the mechanism is known as “cooperative GC”.

Superserver can be set up to do GC in background, cooperatively, or a mix of both, by setting the *GcPolicy* parameter in *firebird.conf*.



*GcPolicy was introduced with Firebird v.2.0. In older versions, Background was the only GC mechanism available to Superserver.*

*The default GcPolicy setting was Combined for Superserver versions lower than 2.1.3 and 2.0.5, respectively. From versions 2.1.3 and 2.0.5, the default policy became Background.*

### Index Cleanups

Similar cleanups occur on the database pages where index data are stored. Index data relating to obsolete records that have not yet been garbage-collected cannot be cleared.

### Database Header Statistics

The Firebird engine keeps transaction summary statistics on the database header page. Statistics read from the header of an active database can tell you much about the state of garbage following the most recent garbage collection. The command *gstat -h[header]* reports numbers that can help you to determine whether very old (possibly abandoned) transactions are causing pages to be clogged with garbage.

For the application programmer, monitoring these header statistics can be a useful way to test whether program code is processing transactions in the way you intended it to.

## “Garbage buildup”

Well-written applications with well-behaved users do not accumulate huge volumes of garbage, relative to the throughput of transactions. The inbuilt mechanisms for disposing of garbage and freeing space on pages just work. However, not all applications manage transactions well and users are not always as careful with their database connections as designers would like. Frequent large batch updates or deletes also have the potential to over-stretch a system’s capacity to deal with garbage in a timely way. Such conditions can lead to large build-ups of garbage that will progressively slow down performance and cause database files to grow unreasonably.

Where the inbuilt mechanisms are not coping with the volume of garbage, some intervention mechanisms are available to reduce a garbage stockpile. One is sweeping, which can be performed manually or automatically; another is backing up with *gbak*; yet another is a directed cooperative garbage collection, useful for clearing large amounts of garbage left behind by batch deletes or updates.

## Sweeping

Sweeping is a supplementary garbage collection mechanism that is built into the Firebird engine. Under any server model, sweeps can be performed manually using *gfix -sweep*. It is also an alternative—and the default action—to have it performed automatically.

## Automatic sweeping

By default, databases are created with a sweep interval of 20,000 transactions. The sweep interval represents the difference between the number of the “oldest snapshot” (OST, the newest transaction that was active last time garbage collection was performed) and the oldest transaction that is still “interesting” (OIT). An automatic sweep will happen soon after that threshold is reached.

The sweep interval can be set to a different threshold if needed, using *gfix -b n*, where *n* is an integer. Automatic sweeping can be disabled altogether by using an *n* value of zero. When auto-sweeping is disabled, it is up to the administrator to ensure that manual sweeps are conducted when required.



*In Classic versions before Firebird 2.5, auto-sweep runs in the context of the user attachment. An “unlucky” transaction that happens to start as the sweep threshold is reached is landed with the the sweep and must wait until the sweep is done. From v.2.5-onward, sweep runs in a separate worker thread for all models.*

## Manual sweeps

Manual sweeping is indicated for conditions where the normal background or cooperative garbage collection is not keeping up with the throughput of transactions in a database. A manual sweep whilst the database is off-line may be helpful sometimes, or even on a regular schedule, if the database is prone to extraordinary buildups of garbage.

If *nbackup* is the mechanism used for regular backups, scheduling regular sweeps with users off-line may be the only way to keep on top of garbage disposal, especially in systems where there are application programs that do not take good care to commit transactions regularly or explicitly.



*You can run a manual sweep at any time. It is not a requirement to have users off-line. However, if your need to sweep seems to arise from poor transaction management, sweeping is likely to inhibit performance for very little gain. Catch-22!*

When formulating a maintenance plan it is essential to monitor the garbage situation over a realistic period of usage to assess whether manual sweeps should be factored in.

## Using *gbak* for cleanup

Where regular *gbak* backups are practicable, they are the best mechanism for getting stubborn garbage out of the way. Running *gbak -backup*—without the *-g* switch—serves the dual purpose of backing up everything and clearing out all of the old record versions and delete stubs it finds on its way through every table.

## Using the *-g* switch

The *-g* switch is the abbreviated form of *-[no\_]g[arbage\_collection]*. In databases where a lot of garbage accumulates, administrators sometime prefer to separate the role of *gbak* as a backup tool from its role as a garbage collector. It is fairly common practice under these conditions to schedule a no-garbage backup and a manual sweep at different times.

## Cooperative garbage collection

“Cooperative” garbage collection gets its name from the fact that, after one transaction has committed updates or deletes, the first transaction after it that touches the same records gets to do the clean-up. The transaction that gets hit with this work may be from a different connection, i.e., it is not necessarily the next transaction started by the garbage-creating connection nor, under Classic, even the same process, that lands the job of cleaning up.

After a large batch of updates or deletes, it is possible to “force” a cooperative cleanup, simply by following the final COMMIT with a new transaction that submits a request to read every record in the table. It need not be a bandwidth-eater: a simple stored procedure that opens a cursor on the table and reads a column in each record will suffice.



*Be aware that effective GC is dependent on managing transactions well. No cleanup mechanism can collect garbage that is interesting to any transaction.*

## 3 Index Statistics

---

Making good indexes available to the engine is the key to good query performance. However, even good indexes can throttle performance under adverse conditions. The *gstat* command-line tool provides reports on all indexes, which can help in identifying and improving poorly performing ones.



*The *gstat* program can generate other statistics from a database as well. Refer to Collecting Database Statistics—*gstat* in Chapter 38 for details.*

To get a report covering all the indexes in the database, in alphabetical order by table, and direct the output to a file, use the command

```
gstat ff25 -index > ff25.stat1
```

“ff25” here is a database alias. You can use any name and suffix for your output file.

Here is an excerpt from the output, concerning one table in this database:

```
FF_TRANSACTION (132)
```

```
Index RDB$PRIMARY4 (0)
```

```
Depth: 2, leaf buckets: 8, nodes: 4824
```

Average data length: 7.03, total dup: 0, max dup: 0

Fill distribution:

0 - 19% = 0  
 20 - 39% = 0  
 40 - 59% = 0  
 60 - 79% = 1  
 80 - 99% = 7

Index TRANSAC\_DATE\_XA (2)

Depth: 2, leaf buckets: 6, nodes: 4824

Average data length: 0.34, total dup: 3226, max dup: 23

Fill distribution:

0 - 19% = 0  
 20 - 39% = 0  
 40 - 59% = 4  
 60 - 79% = 1  
 80 - 99% = 1

Index TRANSAC\_DATE\_XD (1)

Depth: 2, leaf buckets: 7, nodes: 4824

Average data length: 0.34, total dup: 3226, max dup: 23

Fill distribution:

0 - 19% = 1  
 20 - 39% = 0  
 40 - 59% = 5  
 60 - 79% = 0  
 80 - 99% = 1

The table FF\_TRANSACTION (numbered 132 in the metadata table RDB\$RELATIONS) has just three indexes: one for the primary key and an ascending and a descending index on a TIMESTAMP column, numbered 0, 2 and 1, respectively.

All indexes for this table have a Depth of 2, meaning they are just two levels deep. The first level is the root page for the index and it is not counted in the Depth statistic. It points to the pages where the first level of values is stored. If there is enough space, the actual record addresses (“leaf” values) are stored there and Depth will be 1. If not, indirection pointers are stored at that level to pages in a second level.

The index-specific branches at the next level might consist of more pointers, i.e., another level of indirection that points to a third level. However, for this table, the indexes are reasonably shallow: the second level in all cases has the actual “leaf” values.

In *gstat* reports, index pages are called “buckets”. “Nodes” represents the number of indexed records. In our example, the leaf nodes are distributed across eight pages in the case of the primary index, six pages in the case of the ascending TIMESTAMP index and seven pages for the descending index.

We can see from the fill distribution breakdown that most of the pages are fairly well filled.



*If garbage collection was done recently and it was able to clear all the old record versions, the total of pages in the breakdown should be the same as the figure calculated in Leaf buckets. However, gstat does not connect to a database as a client; rather, it reads the data for its calculations directly from the file. Thus, it is not “transaction-aware”. It could*

*be simply a matter of timing if some index leaf pages appear not to have been synchronised with table cleanup.*

## Performance and Index Depth

As index depth increases, efficiency decreases. Depths of three or more on busy indexes will make performance slower than it could be. This database has just a 4 Kb page size and the table is quite small. It will take quite a lot of additional nodes to trigger a third level of indirection. At that point we should think about increasing the page size of our database in order to knock back the depths (indirection levels) of our busiest indexes.



*If you are using `gstat -i` regularly to monitor index depths, the database header information—including the page size—is always output at the start of the report.*

## Duplicates and Selectivity

“Total dup” shows the number of nodes that have non-unique key values. “Max dup” is the largest count of duplicate key values. You can figure out the number of nodes in the index that are unique by subtracting “Total dup” from “Nodes”.

Selectivity (not displayed in this report) is the degree to which a targeted record is likely to be found. The Firebird engine recalculates the selectivity of indexes—including the individual selectivity of key fields in compound indexes from v.2.0-forward—each time the database is brought online.

Such is the contrariness of life that the most selective index or index column has the lowest selectivity value. You can query the system tables to examine the latest calculated selectivities of your indexes and keys:

- The column `RDB$STATISTICS` in `RDB$INDICES` stores the selectivities for whole indexes
- From the “2” series onward, segment-level statistics for multi-segment indexes are also maintained, in `RDB$INDEX_SEGMENTS`.

To some extent, the “dup” results from `gstat -i` are broad indicators of the selectivity of the index.

At one extreme, the most selective index is a unique one, such as the primary key index in our example. Unique indexes have zero in both “dup” results.

At the other extreme would be a table that had “total dup” close to the “nodes” value and “max dup” also up there in the high numbers. Indexes like that are best dropped, as a rule, although foreign key indexes exhibiting such statistics cannot be dropped. Certainly, do not try to force the engine, via a supplied `PLAN`, to use an index with poor selectivity.

During monitoring of the index statistics, do keep an eye out for indexes that start throwing up statistics like that. They are a common cause of poor query performance.

## Rebuilding Indexes

Indexes are implemented as binary trees. When a new record is added to the table, a new node (“branch”) is added above the existing nodes. As new nodes proliferate, the tree tends to become “top-heavy” and the index statistics go out of date. The effect is that some indexes that should be useful in searches and sorts start to cause slow-downs instead of improving search performance. Rebuilding the index can return it to its former optimal condition.



The “2” series versions of Firebird do better than their predecessors at balancing indexes.

## Using ALTER INDEX to rebuild an index

Indexes can, if necessary, be rebuilt using DDL statements. You need to be SYSDBA, a privileged user or the owner of the indexed table and the index itself. Exclusive access to the database is required.

```
ALTER INDEX <index-name> ACTIVE
```

clears the index tree and rebuilds it.

```
ALTER INDEX <index-name> INACTIVE
```

is a valid way to clear the tree for non-constraint indexes. Commit it; then run the ALTER INDEX ACTIVE command to do the rebuild. Constraint indexes cannot be deactivated.



*It is not useful to rebuild an index that has only a few nodes: it will not help if the index has fewer leaf nodes than could fill a page.*

*The larger the table, the longer it takes to rebuild an index. Very large rebuilds could take the database out of action for a long period. If you have identified indexes that are prone to “losing their edge” frequently, factor the out-time into your off-peak maintenance activity.*

## Using DROP INDEX and CREATE INDEX

If you have an index that frequently loses its edge, you might consider just dropping it and recreating it on an “as required” basis. In evaluating the mechanisms for housekeeping indexes, it is strongly recommended that you test each approach under similar conditions with typical sample data, to arrive at the most effective strategy for that particular database.



*If you choose to spring-clean by this method, keep a script that can be run every time the rebuild is required. Not only does it simplify the task, especially if you have more than one index to clean up, but it ensures that the index redefinition will be identical every time.*

## Using *gbak* as the rebuilding mechanism

Restoring a database from a *gbak* backup recreates every index from scratch, not just those you have identified as troublesome. You get completely “clean”, well-balanced trees for all indexes. If your system has applications that regularly run large batch operations on several tables, it is worth considering a full *gbak* backup and restore as good medicine for the health of the entire database.

## Recalculating Index Statistics

The Firebird engine maintains the selectivities of indexes in a system table called RDB\$STATISTICS. The query optimizer refers to these statistics when it constructs a retrieval plan for SELECT queries. It does not recalculate the statistics constantly but just whenever the database file is first opened.

Statistics recalculation does not change the state of an index tree. Its purpose is to analyse the state of the table and its indexes and (in latter versions) of the individual keys of compound indexes, to inform the optimizer of the current states.

The value stored is a number between zero and 1. The more unique nodes in an index tree, the higher the selectivity and the LOWER the number calculated. Unique indexes have a value of zero (or near-zero). Any indexes with a selectivity value above about 0.3 have poor selectivity and doubtful value for optimizing a search.

When the number of nodes in an index increases or decreases dramatically, as after a large bulk operation, for example, recalculating the statistics may improve the usefulness of the index somewhat, until it is possible to run an index rebuild.

## Using SET STATISTICS

Statistics recalculation can be performed at any time using the SET STATISTICS command, either with isql (interactively or via a command-line script) or as a DSQL command from a host application. It requires SYSDBA or owner credentials.

```
SET STATISTICS INDEX <name>;
```

where <name> is the name of the index.



*If you need to recalculate statistics for several indexes regularly, consider making a script for the entire task.*

## 4 Page Size

---

Page size is the size of a block of disk space that the Firebird engine requests from the operating system. The required page size can be specified as an optional parameter in the CREATE DATABASE statement; otherwise, databases created by Firebird servers from v.2 onward have a default page size of 4 Kb. In earlier versions, the default page size was 1 Kb. Databases of less than 4 Kb cannot be created under Firebird 2.1 or higher.



*It is possible to have page sizes smaller than 4 KB on databases that have been upgraded from older versions. Backup and restore with *gbak* retains the existing page unless a new page size is specified on restore (see below).*

As explained elsewhere, page size in databases version 2 and higher can be configured to be as large as 16 Kb. Somewhere within this range will be optimal for the conditions under which your system is deployed.

Through studying the index statistics using *gstat* or another tool, you might detect an indication that a larger or smaller page size would improve the performance of a database. You can change the page size by doing a full *gbak* backup and restoring it with the new page size in the command, using the optional *-page\_size* switch.

## 5 Disk Usage

---

With the large capacities of today's disks, some could regard watching database growth to guard against running out of disk space as unnecessary. Yet some sites ignore disk usage at their peril. Administrators are prone to estimating disk space requirements according to the anticipated volume of data to be stored, without considering factors like uncollected garbage, space needed for temporary files or the sizes of external files that may be passing through the system. A disk that looked adequate for two lifetimes when the system went in a couple of years ago could approach capacity if usage monitoring is neglected.

Databases that share hosts with other applications are high risk candidates for serious down-time and possibly corruption on disks that are storing data, willy-nilly, for many applications. It is all too common, for example, to find sites where mail servers and users' home directories are using the same disk for databases, memory cache and even backups!



The Firebird engine has no way to assess the disk usage status: the first it knows it has run out of disk is either when the operating system refuses an allocation request or when a temporary file cannot be created or, if it exists, cannot be written to.

In v.2.1, some measures were introduced pre-write fresh blocks to disk in anticipation of space that is required to flush “dirty” pages from memory to disk. The purpose is to cushion databases against a sudden out-of-disk condition occurring when the flush is in progress.

V.2.1 also introduced a strategy to reduce fragmentation: pre-allocating disk space in relatively larger chunks. At the time of this writing, this pre-allocation is available only on Windows. Besides improving the overall contiguity of the data pages for a table, it also helps the “cushioning” measures when disk space is getting tight. The maximum size of the pre-allocated chunks is configurable: refer to the parameter *DatabaseGrowthIncrement* in Chapter 34.

It helps, insofar as it gives the DBA some room to take action before disaster strikes. It is not a magic bullet that suddenly creates more space on disk! Nobody has invented that yet.

## Other Considerations

---

### Usage Monitoring

---

The health of a database is very much conditioned by the quality of the applications that manipulate the data and the humans who deploy and use the applications. A lot can be learnt by developing a regime of monitoring the behaviour of applications and users, using the MON\$ database monitoring facilities and, if you have already commissioned Firebird 2.5, the Trace and Audit services. Both systems are discussed in detail in Part VIII, *Administering and Securing Firebird*.

If programming utilities is not your game, look out for some excellent third-party tools that have been built over these facilities as they have emerged in releases. Visit the Downloads section of the IBPhoenix website for links to vendor sites and trial versions.

### Data Access and Security

---

It is very important to maintain a very orderly system for regulating who can access your servers and which users may access which data. The following points of good practice are intended to help in the task of defining and maintaining your system security.

#### Users

Create enough users to satisfy the particular needs of your site but avoid creating too many.

On large sites, consider creating users that match jobs, rather than users that match one-to-one with the humans performing the jobs. When people leave, it is much safer to change the password for the job than to rely on remembering to remove a user from the security database.

Be careful about humans who have escalated privileges as system users but for whom administrative privileges in the databases might not be appropriate.

Ensure that server access passwords are changed regularly and coach users in the art of choosing strong passwords.

## **SQL Access Privileges**

Design the data access scheme BEFORE any ordinary user access is granted and put a procedure in place to ensure that the scheme is followed rigorously.

Do not grant permissions directly to users. Use SQL roles.

## **Database Access**

Use database aliases and insist that user applications support them.

Do not place databases in shared locations.

Pay attention to the configuration in firebird.conf of directories that are accessed by the Firebird server (UdfAccess, TempDirectories, ExternalFileAccess, et al.).

## **Backups**

Do not store backups on the same host as the database.

## APPENDIX

## XI

## UPGRADE SCRIPTS

This Appendix contains copies of the upgrade scripts for upgrading legacy security databases and metadata when upgrading from Firebird 2.0.x and lower versions to Firebird 2.1 and higher.



If upgrading metadata and data to Firebird 2.5, restore the databases using *gbak* with the *-fix\** switches instead of running the metadata upgrade script.

## Security Database

---

```
/* security_database.sql */
/*
 * The contents of this file are subject to the Initial
 * Developer's Public License Version 1.0 (the "License");
 * you may not use this file except in compliance with the
 * License. You may obtain a copy of the License at
 * http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_idpl.
 *
 * Software distributed under the License is distributed AS IS,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied.
 * See the license for the specific language governing rights
 * and limitations under the License.
 *
 * The Original Code was created by Alex Peshkov on 16-Nov-2004
 * for the Firebird Open Source RDBMS project.
 *
 * Copyright (c) 2004 Alex Peshkov
 * and all contributors signed below.
 *
 * All Rights Reserved.
```

```

* Contributor(s): _____.
*
*/

-- 1. temporary table to alter domains correctly.
CREATE TABLE UTMP (
 USER_NAME VARCHAR(128) CHARACTER SET ASCII,
 SYS_USER_NAME VARCHAR(128) CHARACTER SET ASCII,
 GROUP_NAME VARCHAR(128) CHARACTER SET ASCII,
 UID INTEGER,
 GID INTEGER,
 PASSWD VARCHAR(64) CHARACTER SET BINARY,

 PRIVILEGE INTEGER,

 COMMENT BLOB SUB_TYPE TEXT SEGMENT SIZE 80 CHARACTER SET UNICODE_FSS,
 FIRST_NAME VARCHAR(32) CHARACTER SET UNICODE_FSS DEFAULT _UNICODE_FSS '',
 MIDDLE_NAME VARCHAR(32) CHARACTER SET UNICODE_FSS DEFAULT _UNICODE_FSS '',
 LAST_NAME VARCHAR(32) CHARACTER SET UNICODE_FSS DEFAULT _UNICODE_FSS ''
);
COMMIT;

-- 2. save users data
INSERT INTO UTMP(USER_NAME, SYS_USER_NAME, GROUP_NAME, UID, GID, PRIVILEGE,
 COMMENT, FIRST_NAME, MIDDLE_NAME, LAST_NAME, PASSWD)
SELECT USER_NAME, SYS_USER_NAME, GROUP_NAME, UID, GID, PRIVILEGE,
 COMMENT, FIRST_NAME, MIDDLE_NAME, LAST_NAME, PASSWD
FROM USERS;
COMMIT;

-- 3. drop old tables and domains
DROP TABLE USERS;
DROP TABLE HOST_INFO;
COMMIT;

DROP DOMAIN COMMENT;
DROP DOMAIN NAME_PART;
DROP DOMAIN GID;
DROP DOMAIN HOST_KEY;
DROP DOMAIN HOST_NAME;
DROP DOMAIN PASSWD;
DROP DOMAIN UID;
DROP DOMAIN USER_NAME;
DROP DOMAIN PRIVILEGE;
COMMIT;

-- 4. create new objects in database

```

```

CREATE DOMAIN RDB$COMMENT AS BLOB SUB_TYPE TEXT SEGMENT SIZE 80 CHARACTER SET
UNICODE_FSS;
CREATE DOMAIN RDB$NAME_PART AS VARCHAR(32) CHARACTER SET UNICODE_FSS DEFAULT _UNICODE_FSS
'';
CREATE DOMAIN RDB$GID AS INTEGER;
CREATE DOMAIN RDB$PASSWD AS VARCHAR(64) CHARACTER SET BINARY;
CREATE DOMAIN RDB$UID AS INTEGER;
CREATE DOMAIN RDB$USER_NAME AS VARCHAR(128) CHARACTER SET UNICODE_FSS;
CREATE DOMAIN RDB$USER_PRIVILEGE AS INTEGER;
COMMIT;

```

```

CREATE TABLE RDB$USERS (
 RDB$USER_NAME RDB$USER_NAME NOT NULL PRIMARY KEY,
 /* local system user name for setuid for file permissions */
 RDB$SYS_USER_NAME RDB$USER_NAME,
 RDB$GROUP_NAME RDB$USER_NAME,
 RDB$UID RDB$UID,
 RDB$GID RDB$GID,
 RDB$PASSWD RDB$PASSWD,

 /* Privilege level of user - mark a user as having DBA privilege */
 RDB$PRIVILEGE RDB$USER_PRIVILEGE,

 RDB$COMMENT RDB$COMMENT,
 RDB$FIRST_NAME RDB$NAME_PART,
 RDB$MIDDLE_NAME RDB$NAME_PART,
 RDB$LAST_NAME RDB$NAME_PART);
COMMIT;

```

```

CREATE VIEW USERS (USER_NAME, SYS_USER_NAME, GROUP_NAME, UID, GID, PASSWD,
 PRIVILEGE, COMMENT, FIRST_NAME, MIDDLE_NAME, LAST_NAME, FULL_NAME) AS
 SELECT RDB$USER_NAME, RDB$SYS_USER_NAME, RDB$GROUP_NAME, RDB$UID, RDB$GID,
 RDB$PASSWD,
 RDB$PRIVILEGE, RDB$COMMENT, RDB$FIRST_NAME, RDB$MIDDLE_NAME, RDB$LAST_NAME,
 COALESCE (RDB$first_name || _UNICODE_FSS ' ', '') ||
 COALESCE (RDB$middle_name || _UNICODE_FSS ' ', '') ||
 COALESCE (RDB$last_name, '')
 FROM RDB$USERS
 WHERE CURRENT_USER = 'SYSDBA'
 OR CURRENT_USER = RDB$USERS.RDB$USER_NAME;
COMMIT;

```

```

GRANT ALL ON RDB$USERS to VIEW USERS;
GRANT SELECT ON USERS to PUBLIC;
GRANT UPDATE(PASSWD, GROUP_NAME, UID, GID, FIRST_NAME, MIDDLE_NAME, LAST_NAME)
 ON USERS TO PUBLIC;
COMMIT;

```

```
-- 5. move data from temporary table and drop it
INSERT INTO RDB$USERS(RDB$USER_NAME, RDBSYS_USER_NAME, RDBGROUP_NAME, RDBUID, RDBGID,
RDB$PRIVILEGE,
 RDB$COMMENT, RDB$FIRST_NAME, RDB$MIDDLE_NAME, RDB$LAST_NAME,
RDB$PASSWD)
 SELECT USER_NAME, SYS_USER_NAME, GROUP_NAME, UID, GID, PRIVILEGE,
 COMMENT, FIRST_NAME, MIDDLE_NAME, LAST_NAME, PASSWD
 FROM UTMP;
COMMIT;

DROP TABLE UTMP;
COMMIT;
```

## Metadata Repair

---

For upgrading databases to ODS 11.1 only.

For updating ODS 11.0 and lower to ODS 11.2, use the metadata repair switches on the *gbak* restore—see [The gbak method](#) under [Metadata Repair in Chapter 5, Migration Notes](#).

Script `metadata_charset_create.sql`, which creates the procedures `rdbs$fix_metadata` and `rdbs$check_metadata`.

For instructions, see [The script method](#) under [Metadata Repair in Chapter 5, Migration Notes](#).

```
/*
 * The contents of this file are subject to the Initial
 * Developer's Public License Version 1.0 (the "License");
 * you may not use this file except in compliance with the
 * License. You may obtain a copy of the License at
 * http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_idpl.
 *
 * Software distributed under the License is distributed AS IS,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied.
 * See the License for the specific language governing rights
 * and limitations under the License.
 *
 * The Original Code was created by Adriano dos Santos Fernandes
 * for the Firebird Open Source RDBMS project.
 *
 * Copyright (c) 2007 Adriano dos Santos Fernandes <adrianosf@uol.com.br>
 * and all contributors signed below.
 *
 * All Rights Reserved.
 * Contributor(s): _____.
```

```

set term !;

create or alter procedure rdb$fix_metadata
(charset varchar(31) character set ascii)
returns
(table_name char(31) character set unicode_fss,
 field_name char(31) character set unicode_fss,
 name1 char(31) character set unicode_fss,
 name2 char(31) character set unicode_fss)
as
declare variable system integer;
declare variable field1 char(31) character set unicode_fss;
declare variable field2 char(31) character set unicode_fss;
declare variable has_records integer;
begin
 for select rf.rdb$relation_name, rf.rdb$field_name,
 (select 1 from rdb$relation_fields
 where rdb$relation_name = rf.rdb$relation_name and
 rdb$field_name = 'RDB$SYSTEM_FLAG'),
 case rdb$relation_name
 when 'RDB$CHARACTER_SETS' then 'RDB$CHARACTER_SET_NAME'
 when 'RDB$COLLATIONS' then 'RDB$COLLATION_NAME'
 when 'RDB$EXCEPTIONS' then 'RDB$EXCEPTION_NAME'
 when 'RDB$FIELDS' then 'RDB$FIELD_NAME'
 when 'RDB$FILTERS' then 'RDB$INPUT_SUB_TYPE'
 when 'RDB$FUNCTIONS' then 'RDB$FUNCTION_NAME'
 when 'RDB$GENERATORS' then 'RDB$GENERATOR_NAME'
 when 'RDB$INDICES' then 'RDB$INDEX_NAME'
 when 'RDB$PROCEDURES' then 'RDB$PROCEDURE_NAME'
 when 'RDB$PROCEDURE_PARAMETERS' then 'RDB$PROCEDURE_NAME'
 when 'RDB$RELATIONS' then 'RDB$RELATION_NAME'
 when 'RDB$RELATION_FIELDS' then 'RDB$RELATION_NAME'
 when 'RDB$ROLES' then 'RDB$ROLE_NAME'
 when 'RDB$TRIGGERS' then 'RDB$TRIGGER_NAME'
 else NULL
 end,
 case rdb$relation_name
 when 'RDB$FILTERS' then 'RDB$OUTPUT_SUB_TYPE'
 when 'RDB$PROCEDURE_PARAMETERS' then 'RDB$PARAMETER_NAME'
 when 'RDB$RELATION_FIELDS' then 'RDB$FIELD_NAME'
 else NULL
 end
 from rdb$relation_fields rf
 join rdb$fields f
 on (rf.rdb$field_source = f.rdb$field_name)
 where f.rdb$field_type = 261 and f.rdb$field_sub_type = 1 and
 f.rdb$field_name <> 'RDB$SPECIFIC_ATTRIBUTES' and

```

```

 rf.rdb$relation_name starting with 'RDB$'
 order by rf.rdb$relation_name
into :table_name, :field_name, :system, :field1, :field2
do
begin
 name1 = null;
 name2 = null;

 if (field1 is null and field2 is null) then
 begin
 has_records = null;

 execute statement
 'select first 1 1 from ' || table_name ||
 ' where ' || field_name || ' is not null' ||
 iif(system = 1, ' and coalesce(rdb$system_flag, 0) in (0, 3)', '')
 into :has_records;

 if (has_records = 1) then
 begin
 suspend;

 execute statement
 'update ' || table_name || ' set ' || field_name || ' = ' ||
 ' cast(cast(' || field_name || ' as blob sub_type text character
set none) as ' ||
 ' blob sub_type text character set ' || charset || ') ' ||
 iif(system = 1, 'where coalesce(rdb$system_flag, 0) in (0, 3)', '');
 end
 end
 else
 begin
 for execute statement
 'select ' || field1 || ', ' || coalesce(field2, ' null') || ' from '
|| table_name ||
 ' where ' || field_name || ' is not null' ||
 iif(system = 1, ' and coalesce(rdb$system_flag, 0) in (0, 3)', '')
 into :name1, :name2
 do
 begin
 suspend;

 execute statement
 'update ' || table_name || ' set ' || field_name || ' = ' ||
 ' cast(cast(' || field_name || ' as blob sub_type text character
set none) as ' ||
 ' blob sub_type text character set ' || charset || ') ' ||
 ' where ' || field1 || ' = ''' || name1 || '''' ||

```



```

 iif(name2 is null, '', ' and ' || field2 || ' = '' || name2 ||
 ''');
 end
end
end
end!

commit!

create or alter procedure rdb$check_metadata
returns
 (table_name char(31) character set unicode_fss,
 field_name char(31) character set unicode_fss,
 name1 char(31) character set unicode_fss,
 name2 char(31) character set unicode_fss)
as
begin
 for select table_name, field_name, name1, name2
 from rdb$fix_metadata ('UTF8')
 into :table_name, :field_name, :name1, :name2
 do
 begin
 suspend;
 end
 end
end!

commit!

set term ;!

```

Script metadata\_charset\_drop.sql, dropping the metadata update procedures when completed.

```

/*
 * The contents of this file are subject to the Initial
 * Developer's Public License Version 1.0 (the "License");
 * you may not use this file except in compliance with the
 * License. You may obtain a copy of the License at
 * http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_idpl.
 *
 * Software distributed under the License is distributed AS IS,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied.
 * See the License for the specific language governing rights
 * and limitations under the License.
 *
 * The Original Code was created by Adriano dos Santos Fernandes
 * for the Firebird Open Source RDBMS project.
 *
 * Copyright (c) 2007 Adriano dos Santos Fernandes <adrianosf@uol.com.br>

```

```
* and all contributors signed below.
*
* All Rights Reserved.
* Contributor(s): _____.
*/
```

```
drop procedure rdb$check_metadata;
drop procedure rdb$fix_metadata;
commit;
```

## APPENDIX

## XI

# APPLICATION INTERFACES

Once you create and populate a database, its content can be accessed through a client application. Some client applications—such as the Firebird *isql* tool and a range of excellent commercial and open source database administration tools—provide the capability to query data interactively and to create new metadata..

## Application Development

---

Any application developed as a user interface to one or more Firebird databases will use the SQL query language, both to define the sets of data that can be stored and to pass SQL statements to the server requesting operations on data and metadata.

Firebird implements a set of SQL syntaxes which have a high degree of compliance with the recognized SQL-92 standards. The Firebird API provides complete structures for packaging SQL statements and the associated parameters and for receiving the results back into applications.

### Dynamic client/server applications

---

Applications often need to cater for run-time SQL statements that are created and modified by applications, or entered by users on an ad hoc basis. Applications typically provide user selection lists, retrieved from database tables, from which users choose search criteria for the data they wish to retrieve and the operations they wish to perform. The program constructs queries from the user's choices and manages the data retrieved.

Client applications use dynamic SQL (DSQL) for submitting queries in run-time. The Firebird client exposes the API as a library of functions that pass the complex record structures that form the data-level protocol for communication between the application and the server.



*API programming is a big topic, beyond the scope of this guide. However, because dynamic SQL does not surface certain functions in the language, this guide does refer to some API functions to help the reader understand how driver and component interfaces make them visible to their design environments.*

## The Firebird Core API

---

Programming directly with the API is necessary when writing drivers for scripting languages such as PHP and Python and for developing object-oriented data access classes for object-oriented languages like Java, C++ and ObjectPascal. Applications may also be written to call the API functions directly, without the mediation of a driver. These “direct-to-API” applications can be powerful and flexible, with benefits in execution speed, small footprint and fine control over memory allocation.

### Core API function categories

API functions, all having names starting with *isc\_*, fall into eight categories:

- Database attach and detach, e.g. *isc\_attach\_database()*
- Transaction start, prepare, commit, and rollback, e.g. *isc\_start\_transaction()*
- Statement execution calls, e.g. *isc\_dsql\_describe()*
- Services API calls, e.g., *isc\_service\_attach()*
- BLOB calls, e.g. *isc\_blob\_info()*
- Array calls, e.g. *isc\_array\_get\_slice()*
- Database security, e.g., *isc\_attach\_database()*
- Information calls, e.g. *isc\_database\_info()*
- Date and integer conversions, e.g. *isc\_encode\_date()*

For more information about direct-to-API programming, look for the **API Guide** volume of the beta InterBase 6 documentation set, published by Inprise/Borland—back in the day!

## Application interfaces using the API

---

Applications that use drivers for generic interfaces like ODBC or JDBC rely on DSQL statements beneath user interfaces such as query builders and other tools.

With the rise of rapid application development (RAD) tools in the past two decades, the encapsulation of the API functions in suites of classes and components presents a variety of attractive application development options for Firebird developers.

### Object-oriented classes

Object-oriented data access classes and components encapsulate the function calls and structures of the API. All have properties and methods that analyze and parse request statements and manage the results passed back. The richer classes include methods and properties that support Firebird's special capabilities, such as multi-database transactions, array handling and parameterized statements. Most component suites implement at least one class of container component for buffering one or more rows returned to the client as a result set. Some implement advanced techniques such as scrolling cursors, “live datasets”, callbacks and transaction management.

The Jaybird Type 4 (“native”) JDBC driver provides a dedicated interface for platform-independent development with Firebird in Java. Several component suites have become established as the interface of choice for developers using Delphi®, FreePascal and C++Builder® to write front-ends for Firebird databases. The two best-developed are *IB Objects* and *FIBPlus*. Several other component suites are available with lighter support for Firebird features.

## Embedded Firebird applications

Firebird provides for two distinct embedding models: embedded SQL applications and embedded servers.

### Embedded SQL applications

In this model, the application program incorporates both the client/server interface and the end-user application layers in a single executable. SQL statements are entered directly in the source code of a program written in C, C++, or another programming language. The application source code is then passed through *gpre*, the pre-processor, which searches the code for blocks of SQL statements. It substitutes them with source code macro function calls that are functionally equivalent to the dynamic API library functions.

When this pre-processed source code is compiled, all of the plumbing for the SQL conversations is compiled directly into the application. These precompiled statements are known as static SQL.

A special, extra subset of SQL-like source commands is available for this style of application. Known as Embedded SQL (ESQL), it provides a simpler, high-level language “black box” for the programmer, whilst *gpre* does the grunt work of packaging the more complex language structures of the equivalent API calls. These precompiled statements give a slight speed advantage over dynamic SQL, because they avoid the overhead of parsing and interpreting the SQL syntax at run time.

ESQL language and techniques are little used today and are not discussed in this book. The InterBase ***Embedded SQL Guide*** (available from sources on the Web, but no longer from the proprietors of InterBase) provides reference documentation and guidelines for writing embedded Firebird applications.

### Embedded server applications

In the Embedded Server model, there is no pre-compilation of SQL conversations. Client and the server are merged into one compact dynamic library for deployment with a stand-alone application. The application loads the library when it starts up, just as a regular Firebird application would load the client library, and the API functions are called directly at run-time. However, no external server need be installed because this client communicates internally with its own instance of a Firebird server process. When the application terminates, it unloads the embedded server along with the client and no server process remains.

Although it does not use or emulate a network connection, the merged client-server accesses the database in the same way as any other dynamic Firebird client application does. Existing application code written for use on a normal client server network works unmodified with the embedded server.

#### ***Embedded server on Windows***

The embedded server library *fbembed.dll* available in a compressed (.zip) binary kit for versions from v.1.5 up, is a variant of Firebird Superserver in all versions prior to v.2.5. From v.2.5 onward, it is a variant of Superclassic.

If you plan to install it and give it a try, take care to read the special instructions for placement of Firebird's libraries and executables. Updated readme files and other notes are usually located in the \doc\ directory of a regular server installation.

You can install and run embedded server applications on a Windows machine that is also hosting a Firebird Superserver or Classic server. Remote clients cannot attach to a database whilst it is being accessed by an embedded server application. Until v.2.5, multiple local clients could not attach simultaneously, either. From v.2.5, the embedded server is threadable and can accept multiple local attachments.

**v.1.0.x** Firebird 1.0.x does not have an embedded server option on Windows.

### ***Embedded server on Linux/UNIX***

Embedded server is the "native" mode of access for a local client to the Firebird Classic server on Linux/UNIX, including Firebird 1.0.x. The embedded server library for local access is libfbembed.so in Firebird 1.5 and higher, libgds.so in Firebird 1.0.x.

Like the Windows version, the Linux/UNIX embedded client can do double duty as a remote client to another Firebird Classic server. However, this client is not certain to be thread-safe until v.2.5. For multi-threaded applications on the lower versions, it should be avoided in favor of the thread-capable remote client, libfbclient.so.

**v.1.0.x** In Firebird 1.0.x, the full remote client is distributed in the Linux Superserver kit, and is confusingly named libgds.so, like the embedded Classic client.

## **Designing Databases for Client/Server Systems**

---

It is a 'given' that client/server systems need to be designed to be used across networks. It often comes as a shock to newcomers to discover that the "lightning-fast" task that they used to run through an Access, Paradox or MySQL application on a desktop takes all day after they convert the database to a client/server RDBMS.

"It's gotta be something wrong with Firebird," they say. "It can't be my application code—because I didn't change a thing! It can't be my database design—because the same design has been perfect for years!"

The focus of application design for clients with desktop-based back-ends is vastly different to that for remote clients and client/server. The obligatory desktop browsing interface that illusorily displays "200,000 records at a glance" has generated a major industry in RAD data-aware grid components. The developer never needed to ponder on how many human beings might be capable of browsing 200,000 records in a day, let alone at a glance!

Those RAD grid components, that did such a perfect job of presenting unlimited volumes of desktop data in small containers for random browsing, are not a friendly interface for remote clients. Behind the screen, the characteristic client looping operation—"start at record 1 and for every record DO repeat"—that seemed so perfect for processing data that were in-memory images of local tables has the remote client users now demanding the developer's head on a plate.

It is very common, indeed, for the design of the database itself to have been influenced more strongly by the perception of the client interface—"I want a table just like this spreadsheet!"—than by the elegant wisdom of a sound abstract data model.

When the interface is physically separated from the data by transaction isolation and a network of busy wire, much pondering is required. There is much more to migrating from

the desktop than mere data conversion. The ultimate benefits from a critical design review to users, database integrity and performance will amply justify the effort.

## Abstraction of stored data

Even in established client/server environments, all too many poorly-performing, corruption-prone systems are found to have been “designed” using reports and spreadsheets as the blueprint for both database and user interface design. In summary, it is all too common for desktop database conversions to come through to Firebird with many of the following client-server-unfriendly “features”:

- widespread redundancy—structures that graduated from spreadsheets to databases with the same data items repeated across many tables
- hierarchical primary key structures—needed in many desktop database systems to implement dependencies—which interfere with the refined model of foreign key constraints in mature relational database systems
- large, composite character keys composed of meaningful data
- lack of normalization, resulting in very large row structures containing many repeating groups and infrequently-accessed information
- large numbers of unnecessary and often overlapping indexes

That is not to say that the old desktop system was no good. If it worked well in its environment, it was good for its purpose. Client/server is simply a very different technology to “desktop databasing”. It changes the scale of information management from file-and-retrieve to store, manage and manipulate. It shifts the client application away from its desktop role as principal actor to that of message-bearer. Effective client interfaces are lightweight and very elegant in the ways they capture what users want and deliver what they need.

## Shedding the spreadsheet mindset

A common characteristic of desktop database applications is that they provide an interface of grids: data arranged in rows and columns, with scrollbars and other navigation devices for browsing from the first row in a table to the last. Often, the grids deliver a visual structure that exactly mimics the metadata structure of the source tables. It is a common trap to import such tables into a client/server system and consider that the migration task is done.

Moving these legacy databases to client/server usually takes more than a data conversion program. Do your conversion and be prepared to treat your new database objects as holding places. Plan to re-analyze and re-engineer to abstract this style of database into structures that work well in the new environment. In Firebird it is very easy to create new tables and move data into them. For storage, think thin—simple keys, abstracting large table structures into families of linked, normalized relations, massaging calendar-style groups of repeating columns into separate tables, eliminating key structures that compound down dependency levels, eliminating duplicated data, and so on.

If you are all at sea about normalization and abstraction, study some of the excellent literature available in books and websites. Get yourself started using a small data model—a subset of five or six of your primary tables is ideal—rather than hitting a 200-table database as if it were a single problem that you have to solve in a day. This way, the conversion becomes a proactive exercise in self-instruction and solving the tough bits quickly becomes more intuitive. For example, learn about stored procedures and triggers and test what you know by writing a data conversion module!

## Output-driven tables

It is essential to part with the notion that your starting point in designing a relational database is to represent everybody's favourite reports, spreadsheets and most-used reference displays as tables in the database. These things are output and output is retrieved by queries and stored procedures.

## The human interface

Client applications in a system where enterprise information services have a back-end that is a full-blooded DBMS with powerful data processing capabilities do not transform user input, beyond parsing it at source and packaging it into prepared containers—the structures for the API transport functions. FOR loops through hundreds or thousands of rows in a client dataset buffer do not have a place in the front-ends of client/server systems.

The application designer needs to think constantly about the cost of redundant throughput. Pulling huge sets across the wire for browsing bottles up the network and makes the user's experience frustrating. The human face of the system has to concentrate on efficient ways to show users what they need to see and to collect input from them—their instructions and any new data that humans want to add. User interface design should focus on quick and intuitive techniques to capture inputs raw and pass them quickly to the server for any cooking that is required.

Client/server developers can learn a lot from studying successful web form interfaces, even if the applications are not being designed for the Internet, because a web browser is the extreme thin client.

Short, quick queries keep users in touch with the state of the database and dilute the load on the network. Effective database clients present drill-down, searching interfaces, rather than table browsers, and limit sets to no more than 200 rows.

## The relational data storage model

Relational databases depend on robust, highly abstract structures to perform effectively and produce predictable, correct results from operations. Complete analysis of the entities and processes in your system is essential, so that you arrive at a logical model that is free of redundancy and represents every relationship.

## The primary key

During logical analysis, a primary key is established for each grouping of data. The logical primary key helps to determine which item or items are capable of identifying a group of related data uniquely. The physical design of tables will reflect the logical groupings and uniqueness characteristics of the data model, although the table structures and key columns actually draughted are often not exactly like the model. For example, in an Employee table, the unique key involves first and last name fields, along with others. Because the composite unique key of the data model involves several large items that are vulnerable to human error, a special column would be added to the table to act as a surrogate primary key.

A RDBMS relies on each row having a unique column in each table, in order to target and locate a specific row unambiguously, for matching search conditions and for linking data items or streams.

## Relationships

Relationships in the model map to keys in the tables. Theoretically, every relationship in the model would be realized as keys. When keys are linked to one another by foreign key



constraints, the tables become bound into a network of dependencies that reflect how groups of data should interact, regardless of context. Underlying rules in server logic refer to these dependencies in order to protect the referential integrity of the database. The SQL standards formulate rules describing how referential dependencies should work. It is up to the vendor of an individual RDBMS to decide how to implement and enforce them internally.

In individual server engine implementations, there can be technical reasons why certain key constraints should be left without a formal declaration and implemented in an alternative way. For example, most relational systems require mandatory, non-unique indexes on the column elements of foreign keys. Under some data distribution conditions, it may be undesirable to index such columns, if another way can be used to protect the integrity of the relationship concerned.

A RDBMS can also realize relationships that do not involve keys. For example, it can extract sets of data on the basis of comparing values, or expressions involving values, in different columns of a single table or between columns in different tables.

The SQL query language, the structures of stored data sets and the logical skills of the application developer interact to ensure that client/server traffic is kept down and user requests return results that are precise and appropriate.

### **“Hands-off” data access**

Relational systems that are designed for client/server use do not make data directly accessible to users. When a user application wants an operation performed on a set of data, it tells the client module what it wants and the client negotiates with the server to satisfy the request. If the request is denied for some reason, it is the client that bears the bad news back to the application.

If the application requests to read a set of data, it is the client that fetches the output from the server's operation and carries it back to the application. The data that are seen by the application are images of the state of the original data in the database at the moment the conversation between the client and the server began. The images that users see are disconnected—or isolated—from the database. The “moment of isolation” may not be the same moment that the request was received by the server. In a client/server environment, where it is assumed that more than one user is reading and writing data, every request has a context.

### **Multiple users and concurrency**

For a data management system that is designed to allow multiple users to work with images of stored data and to request changes that may impact the work that others are doing, something is needed to manage *concurrency*. Concurrency is the set of conditions that is anticipated when two or more users request to change the same row in a table at the same time, i.e. concurrently. Mature data management systems like Firebird implement some kind of scheme whereby each request is made in a concurrency context. The standard SQL term for this concurrency context is transaction—not to be confused with the “business transactions” that database applications frequently implement!

### **Transactions**

For the ex-user of desktop databases, the transaction one of the most confusing abstractions in the client/server and RDBMS environment. With desktop databases and spreadsheet programs, it is taken for granted that, once the user hits the Save button and the drive light goes out, the operation is done, for better or for worse. It is also a fact that, once the penny drops about transactions, developers tend to spiral rapidly away from the

“spreadsheet mindset” that has preoccupied them during those years when the old database model seemed perfect.

In Firebird, all communications between the client and the server occur during transactions. Even reading a few rows from a table can not occur if a transaction has not started. A transaction starts when the application asks the client to start it. From the point when the transaction begins until it ends—again, at the request of the application—the client/server conversation is open and the application can ask the client to make requests. During this period, operations to change database state are executed and written to disk. However, they do not change database state and they are reversible.

Transactions end when the application asks the client to request the server to commit all of the work since the transaction began (even if the work was nothing but reads) or, in the event of errors, to roll back the work. The rules of atomicity apply: if one pending change of database state fails, requiring it to be rolled back because it can not be committed, then all pending changes in that transaction are rolled back too. The rollback includes any work that was executed by triggers and stored procedures during the course of the transaction.



*For the application designer, it is very useful to visualize each unit of database “work” as a task or group of tasks that are accomplished within a transaction context. Transactions are configured in various ways to condition behavior. For example, one isolation level will return a different kind of conflict message than another.*

*The most effective application programs are aware of these variations and are responsive to them, to the extent that the context of each transaction “extends” in its logical scope to the application workspace surrounding the actual physical transaction.*

Transactions matter so much in a client/server system that this book has devoted three chapters to them in Part V—beginning at chapter 25, Overview of Firebird Transactions.

# Index

## Symbols

- \_ introducer marker for character set 153
- \_ wildcard for LIKE comparisons 355
- ? parameter placeholder 323
- 'Index is in use' error 276
- 'Object is in use' error 291, 580, 614, 639
- "Warm backups" 852
- % wildcard for LIKE comparisons 355
- || double-pipe concatenation operator 144, 351

## Numerics

- 2-phase commit 714
- 32-bit 26
- 64-bit 26

## A

- ACID properties 492
  - Atomicity 492
  - Consistency 492
  - Durability 493
  - Isolation 492
- Adding character sets 162
- Administering databases 76
- Affinity mask value, calculating 694
- Aggregating expressions 414
- Aggregating function 378, 421
  - AVG() 421
  - LIST() 423
  - MAX() 422
  - MIN() 422
  - SUM() 421
  - using COUNT() as 422
- Aggregation
  - effects of NULL in 414
- Aliases
  - database 73
    - connecting with 75
    - on non-default port 678
- aliases.conf 73, 74
- Aliasing
  - and DatabaseAccess parameter 74

- Aliasing (continued)
  - relations 388
  - tables 388
- ALL privilege 750
- ALL() predicate 359
- ALTER DATABASE statement 845
- ALTER DOMAIN statement 189
- ALTER EXCEPTION statement 650
- ALTER INDEX statement 275
- ALTER PROCEDURE statement 612
- ALTER ROLE statement 755
  - and Authentication parameter 755
- ALTER SEQUENCE statement 203
- ALTER TABLE statement 258
- ALTER TRIGGER statement 637, 638
- ALTER USER statement 215, 732
- ALTER VIEW statement 435
- Altering table structure 258
- ANY predicate 359
- API
  - application interfaces using 1020
  - see* Application Programming Interface 22
- Application development 1019
  - core function categories 1020
  - dynamic applications 1019
  - embedded 1021
  - Firebird Core API 1020
  - relational data storage model 1024
- Application Programming Interface 1020
- Architectures
  - comparison 9
- Arithmetic operators 351
  - precedence of 351
- Array columns
  - accessing data 178
  - array descriptor (API) 178
  - defining 176
  - eligible element types 176
  - inserts to 177
  - multi-dimensional 176
  - specifying 177
  - storage of 177

- Array columns (continued)
  - updates to **177**
  - view via stored procedure **605**
- Array types **175–179**
  - and SQL **175**
  - documentation about **178**
  - when to use **176**
- Assertions **347**
- AuditTraceConfigFile parameter **775**
- Authentication parameter
  - and ALTER ROLE statement **755**
- Auto-incrementing keys **635–637**
- Automatic fields **313**
  - Before Insert triggers **314**
  - columns with defaults **313**
  - computed columns **313**
- Autonomous transactions **643**
  - introduction **5**

## B

- Backward compatibility **84**
- Batch DML operations **324**
  - batch inserts **324**
- BEGIN BACKUP statement **845**
- Binary strings **155**
- BLOB columns
  - character set conversion **174**
  - collation sequence **174**
  - external functions for **174**
  - operations on **173**
  - string functions with **174**
  - string input to **173**
  - UDFs for **174**
- BLOB filters **170**
- BLOB page **237**
- BLOB subtypes
  - BINARY **170**
  - custom **170**
  - TEXT **170**
  - with negative numbers **170**
- BLOB types **169–175**
  - and subtypes **169**
    - supported subtypes **170**
  - declaring **171**
  - security **173**
  - segment size **171**
  - segments **171**
  - when to use **172**

- BLR **331**
- BOOLEAN
  - defining as a domain **189**

## C

- Calling selectable procedures **598**
- Cancel a running query **768**
- Cardinality of a set **296**
- CASE() constructs **365**
- CAST() expressions **374**
- Character metadata
  - update script for **1014**
- Character sets **148**
  - adding aliases for **161**
  - adding to a database **162**
  - aliases
    - adding **161**
  - aliases of **151**
  - catalogue of **151**
  - character set NONE **150**
  - client character set **150**
  - collation sequence **157**
  - custom **162**
  - default **149**
  - effects on storage boundaries **152**
  - Firebird character sets **151**
  - for Windows **156**
  - in XSQLVAR **154**
  - introducer marker syntax **153**
  - ISO8859\_1 **156**
  - naming of **151**
  - overrides **149**
    - field-level **153**
    - statement-level **153**
  - RDB\$CHARACTER\_SETS **151**
  - special **154**
  - transliteration **156**
  - well-formedness **152**
- Character sets and collations list **951**
- Character types **143–168**
  - binary strings **155**
  - CHAR alias CHARACTER **147**
  - escape characters **144**
  - fixed length **147**
  - impact of size on memory usage **147**
  - index restrictions **146**
  - limitations **146**

- Character types **143–168** (continued)
  - NCHAR alias NATIONAL CHARACTER **147**
  - NCHAR VARYING and aliases **148**
  - non-printable characters **144**
  - string concatenation **144**
  - string delimiter **144**
  - string functions **145**
  - VARCHAR alias CHARACTER VARYING **148**
  - variable length **147**
- CHECK constraint
  - in table definition **251**
  - restrictions **252**
  - syntax **252**
- CHECK expressions **372**
- Circular reference (relationship) **288**
- Citrix **20**
- Classic
  - MacOSX **37**
  - on POSIX **68**
  - [x]inetd daemon **68**
- Client character set **150**
- Client libraries **22**
- Client/server
  - designing databases for **1022**
  - introduction **13**
- Clients **20**
- COALESCE() **367**
- Collation sequence **148, 157**
  - affecting index size **161**
  - and index size **160**
  - CREATE COLLATION **164**
  - custom **162**
  - default collation sequence **158**
  - DROP COLLATION **167**
  - for a column **159**
  - for a string comparison **159**
  - in a GROUP BY clause **160**
  - in sorting criteria **160**
  - listing of **159**
  - naming of **159**
  - unregistering **167**
- Columns
  - adding **261**
  - altering **258**
  - based on domains **239**
- Columns (continued)
  - changing definition **107**
  - CHARACTER SET **241**
  - COLLATE clause **241**
  - computed **241**
  - DEFAULT value **240**
  - defining **238**
  - dropping **259**
  - introduction **105**
  - restrictions on data type changes **259**
- Command-line tools, Summary **79**
- COMMIT statement **526**
- COMMIT WITH RETAIN statement **526**
- Common table expressions **441**
  - non-recursive **443**
  - recursive **444**
  - restrictions on recursing set **444**
  - syntax pattern **442**
- Comparison operators **351**
- Comparisons
  - and indexing **270**
- Computed columns **241**
- Concatenation of strings **144**
- Concurrency isolation **509**
- Configuration parameters **681–699**
  - AuditTraceConfigFile **681, 775**
  - Authentication **682**
  - BugCheckAbort (Psx) **696**
  - CompleteBooleanEvaluation **682**
  - ConnectionTimeout **683**
  - CpuAffinityMask (Win) **693**
  - DatabaseAccess **683**
  - DatabaseGrowthIncrement **684**
  - DeadlockTimeout **684**
  - DefaultDbCachePages **684**
  - Deprecated
    - CreateInternalWindow (Win) **698**
    - DeadThreadsCollection **699**
    - OldParameterOrdering **698**
  - DummyPacketInterval **684**
  - ExternalFileAccess **697**
  - FileSystemCacheSize **685**
  - Windows permissions issue **686**
  - FileSystemCacheThreshold **685**
  - GCPolicy **686**
  - GuardianOption (Win) **694**
  - IpcName (Win) **694**

- Configuration parameters **681–699**
  - (continued)
  - LegacyHash **686**
  - LockAcquireSpins **686, 815**
  - LockGrantOrder **687, 815**
  - LockHashSlots **686, 815**
  - LockMemSize **687, 815**
  - LockSemCount **688, 815**
  - LockSignal **688, 815**
  - MaxFileSystemCache **685**
  - MaxUnflushedWrites (Win) **694**
  - MaxUnflushedWriteTime (Win) **695**
  - MaxUserTraceLogSize **688**
  - OldColumnNaming **689**
  - ProcessPriorityLevel (Win) **695**
  - Redirection **689**
  - RelaxedAliasChecking **689**
  - RemoteAuxPort **690**
  - RemoteBindAddress **690**
  - RemoteFileOpenAbility **690**
  - RemotePipeName (Win) **695**
  - RemoteServiceName **691**
  - RemoteServicePort **691**
  - RootDirectory **691**
  - SortMemBlockSize **692**
  - SortMemUpperLimit **692**
  - TcpNoNagle **691**
  - TcpRemoteBufferSize **691**
  - TempBlockSize **692**
  - TempCacheLimit **692**
  - TempDirectories **692**
  - UdfAccess **696**
  - UsePriorityScheduler (Win) **695**
- Configuring database **702**
  - access mode **707**
  - default cache size **705**
  - dialect **709**
  - disable auto sweep **709**
  - forced writes **706**
  - getting exclusive access **702**
  - make read-only **707**
  - page fill capacity **708**
  - page size **710**
  - reserve space **708**
  - shutdown **702**
  - SQL dialect **709**
  - sweep interval **708**
  - use all space **708**
- Configuring Firebird **667–680**
  - configuration file **672**
  - customising the TCP/IP port service **675**
  - embedded server **679**
  - Environment variables **668**
    - EDITOR **670**
    - FIREBIRD **670**
    - FIREBIRD\_LOCK **670**
    - FIREBIRD\_MSG **670**
    - FIREBIRD\_TMP **671**
    - HOME **671**
    - ISC\_INET\_SERVER\_HOME **671**
    - ISC\_MSGS **671**
    - ISC\_PATH **671**
    - ISC\_USER and ISC\_PASSWORD **671**
    - LC\_MESSAGES **671**
    - setting on POSIX **669**
    - setting on Windows **668**
    - TMP or TEMP **671**
    - VISUAL **672**
  - find its root directory **667**
  - FIREBIRD environment variable **667**
  - firebird.conf **672**
  - modifying its root directory **668**
  - RootDirectory parameter **668**
  - Windows Registry key **667**
- Connection string syntax **56**
  - inconsistency on Windows **57**
- Consistency isolation **509**
- Constraints
  - dropping **260**
  - FOREIGN KEY **280**
  - interaction of **283**
- Context namespaces **379**
- Context variables **103**
  - (list) **911**
- Correlated subqueries **399**
- Correlated subquery **300**
- Corruption
  - See Database corruption **987**
- CREATE COLLATION statement **164**
- CREATE DATABASE statement **222**
  - database path and name **222**
  - default character set **224**
  - DIFFERENCE file **224**

CREATE DATABASE statement **222**  
     (continued)  
         OWNER attribute **223**  
         page size (PAGE\_SIZE) **223**  
 CREATE DOMAIN statement **182**  
 CREATE EXCEPTION statement **650**  
 CREATE GENERATOR statement **203**  
 CREATE INDEX statement **271**  
 CREATE OR ALTER PROCEDURE  
     statement **613**  
 CREATE OR ALTER TRIGGER  
     statement **637, 639**  
 CREATE OR ALTER VIEW statement  
     **435**  
 CREATE PROCEDURE statement **584**  
 CREATE ROLE statement **750**  
 CREATE SEQUENCE statement **203**  
 CREATE SHADOW statement **849**  
 CREATE TABLE statement **238**  
 CREATE TRIGGER  
     alternative syntax **623**  
 CREATE TRIGGER statement **622**  
 CREATE USER statement **215, 732**  
 CREATE VIEW statement **427**  
 Creating tables **238**  
 Cross joins **393**  
 CTEs **441**  
     syntax **442**  
 CURRENT\_CONNECTION **103**  
 CURRENT\_TIME **103**  
 CURRENT\_TIMESTAMP **103**  
 CURRENT\_TRANSACTION **103**  
     accessing **522**  
 CURRENT\_USER **103**  
 Cursor sets **298**  
 Cursors  
     in PSQL modules **571**  
 Custom character sets **162**  
 Custom collations **162**  
 Customising the TCP/IP port service **675**  
     configuring Services file **679**  
     copy firebird.conf to clients **678**  
     database alias with non-default port **678**  
     in TCP/IP connection string **678**  
     in WNET connection string **678**  
     reconfiguring [x]inetd (Classic, POSIX)

**676**

Customising the TCP/IP port service **675**  
     (continued)  
         redirection on WNET (Windows  
         networking) **676**  
         RemoteServiceName parameter **675,**  
         **677**  
         RemoteServicePort parameter **675, 677**  
         setting up the client **677**  
         using the -p switch **676**

## D

Data Definition Language (DDL) **211–217**  
     ALTER statements **212**  
     CREATE statements **211**  
     DECLARE statements **213**  
     DROP statements **213**  
     object dependencies **215**  
     RECREATE statements **212**  
     reference material **217**  
     scripts **457**  
     storing descriptive text **213**  
     the COMMENT statement **214**  
     using DDL to manage user accounts  
         **215**  
 Data Manipulation Language (DML)  
     scripts **480**  
         what's in them? **481**  
 Data Manipulation Language—DML **295–**  
     **300**  
 Data model **196**  
 Data page **237**  
 Data types **101–111**  
     BLOB types **169**  
     BOOLEAN **189**  
     character types **143**  
     converting **106**  
         valid conversions **108**  
     date and time types **125**  
     DECIMAL **118**  
     floating point **122**  
     keywords for specifying **108**  
     number **113–124**  
         integer **114**  
     NUMERIC **118**  
     supported types **101**

- Database
  - page size
    - and performance **831**
- Database administration **76**
  - healthcare **1001–1010**
- Database aliasing **73**
- Database attributes
  - sweep interval **233**
- Database Backups **817–852**
- Database corruption
  - analysis and repair **988**
  - failed repair **991**
  - indications **987**
  - steps for recovery **989**
- Database design **195–210**
  - criteria for **195**
  - description and analysis **196**
  - file-naming conventions for databases **206**
  - generators **203**
  - indexes **202**
  - keys **198**
    - making atomic **199**
    - surrogate **200**
  - object naming conventions **205**
  - optional delimited identifiers **205**
  - referential integrity **201**
  - schemas and scripts **210**
  - sequences **203**
  - stored procedures **202**
  - tables **197**
  - the physical objects **197**
  - triggers **202**
  - views **202**
- Database file-naming conventions **206**
- Database repair how-to **987–991**
- Database replication **852**
- Database security **741–764**
  - a trick to beat Bad Guys **764**
  - and access (default) **741**
  - attaching to database under a role **751**
  - introduction **7**
  - see* SQL privileges **741**
- Database shadow **236, 847–852**
- Database Statisticsmgstat **780–791**
- Database triggers **620, 639**
  - “gotchas” **640**
  - about **544**
- Database triggers **620, 639** (continued)
  - changing **641**
  - creating **641**
  - events **639**
  - introduction **6**
  - session-level **640**
  - transaction-level **640**
- DatabaseAccess parameter and aliasing **74**
- Databases
  - attributes **225**
    - forced writes **225**
    - single and multi-file **225**
    - sweep interval **225**
  - background garbage collection **233**
  - backing up **236**
  - cache size **227**
    - estimating **227**
  - cache usage **228**
    - RAM requirements **229**
  - creating **221**
    - multi-file database **226**
  - creating and maintaining **219–236**
  - dialect **221**
  - dropping **236**
  - getting information about **224**
  - how to corrupt **235**
  - ISC\_USER and ISC\_PASSWORD **221**
  - keeping clean **232**
  - metadata **206**
  - multi-generational architecture (MGA) **232**
  - objects and counters **234**
  - physical storage for **219**
  - read-only **231**
  - security access **220**
  - stand-by (shadow) **236**
  - sweeping **233**
  - system tables **206**
  - validation and repair **234**
- Date and time types **125**
  - DATE **125**
  - interval of time **126**
  - operations using **133**
  - other date/time functions **140**
  - quick date/time casts **138**
  - sliding century window **129**
  - the EXTRACT() function **138**
  - TIME **126**



- Date and time types **125** (continued)
  - TIMESTAMP **126**
  - uses for casting **137**
  - using CAST() with **135**
- Date literals, pre-defined **104, 130**
- Date/time context variables **132**
  - CURRENT\_DATE **132**
  - CURRENT\_TIME **132**
  - CURRENT\_TIMESTAMP **132**
  - specifying sub-seconds precision **132**
- Date/time literals **127**
  - formats **127**
  - separators in non-US dates **129**
  - type-casting **131**
- DDL triggers **620**
- Deadlocks **794**
- DECLARE EXTERNAL FUNCTION statement **383**
- DECLARE...AS CURSOR FOR... statement **573**
- Default
  - user name and password **48**
- Default character set **149**
- Default disk locations **993–1000**
  - client libraries **998**
  - Fedora, Mageia, Debian et al. **1000**
  - Linux and some UNIX **993**
  - Mac OSX and Darwin **997**
  - Windows **995**
- default\_cache\_pages (v.1.0.x) **231**
- DefaultDbCachePages parameter **230**
- Degree of a set **296**
- DELETE statement **322**
- Delimited identifiers **105**
- Delimited identifiers (SQL-92) **205**
  - for pervasive case-sensitivity **205**
- Deployment topologies **17**
- Derived tables **402, 439**
  - rules for **440**
  - when to use **441**
- Descriptive text for database objects **213**
  - commenting in scripts **214**
  - using COMMENT **214**
- Dialect **87, 208**
  - changing to dialect 3 **88**
  - how to determine **87**
- DISTINCT FROM **354**
- Distributed environments **19**
- DML
  - concept of sets **295**
  - the optimizer
    - introduction **298**
- DML queries **301–328**
  - and custom triggers **327**
  - and referential integrity action clauses **327**
  - batch operations **324**
  - DELETE statement **322**
  - EXECUTE statements **322**
  - executing procedures **325**
  - INSERT INTO statement **311**
  - inserting from an in-line query **311**
  - MERGE statement **320**
  - positioned vs searched operations **315**
  - SELECT statement **301**
  - selecting from procedures **326**
  - that execute server-side code **325**
  - UPDATE OR INSERT statement **319**
  - UPDATE statement **315**
  - use of indexes **330**
  - using parameters **323**
- Domain definition
  - changing **189**
    - constraints on **190**
    - valid changes **190**
- Domains **181–191**
  - benefits of using **181**
  - changing definition **107**
  - character sets and collations in **186**
  - CHECK conditions **184**
    - dependencies in **185**
  - creating **182**
  - DEFAULT attribute **183**
  - defining for use as BOOLEAN **189**
  - dropping **191**
  - in column definitions **186**
  - in PSQL variable definitions **188**
  - introduction **106**
  - NOT NULL attribute **183**
  - overriding attributes **187**
  - using **186**
    - using with CAST expressions **188**
- DROP COLLATION **167**
- DROP DATABASE statement **236**

DROP DOMAIN statement **191**  
 DROP EXCEPTION statement **650**  
 DROP INDEX statement **276**  
 DROP PROCEDURE statement **614**  
 DROP ROLE statement **751**  
 DROP SHADOW statement **851**  
 DROP TABLE statement **263**  
 DROP TRIGGER statement **641**  
 DROP USER statement **215, 732**  
 DROP VIEW statement **436**

## E

Embedded server **53**  
     introduction **13**  
     Windows **45**  
 employee.fdb database **76**  
 Enabling multi-hop **689**  
 END BACKUP statement **845**  
 ASC **273**  
 DESC **273**  
 Environment variables  
     EDITOR **670**  
     FIREBIRD **29, 667, 670**  
     FIREBIRD\_LOCK **670**  
     FIREBIRD\_MSG **670**  
     FIREBIRD\_TMP **671**  
     HOME **671**  
     ISC\_INET\_SERVER\_HOME **671**  
     ISC\_MSGS **671**  
     ISC\_PATH **671**  
     ISC\_USER and ISC\_PASSWORD **221, 671**  
     LC\_MESSAGES **671**  
     TMP or TEMP **671**  
     VISUAL **672**  
 Equi-joins **396**  
 Error  
     ‘Index is in use’ **276**  
     ‘Object is in use’ **291, 614**  
 Error codes list **957**  
 Escape character for apostrophes **144**  
 Escape sequences for literals in expressions **355**  
 Events  
     network problems with **59**  
 Events port  
     introduction **13**

Exact numerics  
     DECIMAL **118**  
     NUMERIC **118**  
     SQL dialect 1 **118**  
 Example database employee.fdb **76**  
 Exceptions  
     receiving **529**  
 Executable blocks **615**  
     about **544**  
     coding **616**  
     in applications **618**  
 Executable procedures  
     introduction **325**  
 EXECUTE BLOCK statement **615**  
     reference **322**  
 EXECUTE privilege **748**  
 EXECUTE PROCEDURE **325**  
 EXECUTE PROCEDURE statement **591**  
     reference **322**  
 EXECUTE STATEMENT Extensions  
     **643–647**  
     AS <user> PASSWORD <password>  
         **645**  
     authentication **647**  
     exceptions **647**  
     external query **646**  
     log-in arguments **645**  
     ON EXTERNAL DATA SOURCE  
         **644, 646**  
     optional clauses **644**  
     ROLE clause **645**  
     to query another database **644, 646**  
     WITH AUTONOMOUS  
         TRANSACTION **644, 646**  
     WITH CALLER PRIVILEGES **645**  
     WITH COMMON TRANSACTION  
         **644, 645**  
 EXECUTE STATEMENT statement  
     reference **322**  
     to install SQL privileges **762**  
 Existential predicates **357**  
     ALL() **359**  
     ANY **359**  
     EXISTS() **358**  
     IN() **359**  
     SINGULAR **359**  
     SOME **359**  
 EXISTS() predicate **358**

Explicit row locking **534**  
 Exporting data via an external table **257**  
 Expression indexes **273, 372**  
 Expressions **345**  
   elements used in **348**  
   escape sequences for literals in **355**  
   expression indexes **372**  
   for computed columns **363**  
   function calls **373**  
   in CHECK conditions **372**  
   in ordering and grouping conditions **371**  
   in PSQL **373**  
   in search conditions **369**  
   NULL in **360**  
   using aggregating functions **378**  
   using CASE() constructs **365**  
     COALESCE() **367**  
     IIF() **369**  
     NULLIF() **368**  
   using CAST() **374**  
   using contextual functions **379**  
   using EXTRACT() **375**  
   using GEN\_ID() **377**  
   using LOWER() **377**  
   using NEXT VALUE FOR **378**  
   using SUBSTRING() **376**  
   using UPPER() **376**  
   with GROUP BY **371**  
   with ORDER BY **371**  
 External files as tables **253**  
 External tables **253**  
   converting to internal table **257**  
   exporting data **257**  
   importing data **255**  
   operations on **255**  
 external\_function\_directory (v.1.0.x) **6**  
 ExternalFileAccess parameter **254**  
 EXTRACT() expressions **375**

## F

fbtrace.conf file **771**  
 fbtracemgr utility **778**  
   action switches for **778**  
   to submit a trace service request **779**  
 fbudf.sql script **384**

Firebird  
   backward compatibility **84**  
   downgrading **84**  
   major releases **85**  
   SQL language **207**  
     and ISO standards **208**  
     DDL **208**  
     DML **209**  
       dynamic vs static **209**  
     interactive SQL **210**  
     PSQL **209**  
     SQL “dialect” **208**  
   sub-releases **86**  
   version lineage **81**  
 Firebird API **22, 1020**  
 Firebird client **20, 21**  
 Firebird configuration file **60**  
 Firebird events  
   introduction **12**  
 Firebird server models  
   introduction **7**  
 FIREBIRD variable **29, 667**  
 firebird.conf **60, 672**  
   editing parameters **674**  
   parameter syntax **673**  
     comments **673**  
     values (Boolean) **673**  
     values (integer) **673**  
     values (string) **673**  
   version differences **674**  
 FIRST...SKIP **302**  
 Fixed decimal types **117**  
   behaviour in operations  
     addition and subtraction **121**  
     division **119**  
     multiplication **121**  
 DECIMAL **118**  
 input and exponents **121**  
 NUMERIC **118**  
 storage **118**  
 FlameRobin **50**  
 Floating point types **122**  
   DOUBLE PRECISION **123**  
   FLOAT **123**  
   SQL dialect 1 **123**  
 FOR UPDATE clause **310, 536**  
 Forced writes **706**

FOREIGN KEY constraint **280**  
     action triggers **282**  
     cascading actions **282**  
     defining **281**  
     parent-child relationship **280**  
     the REFERENCES clause **281**

Full joins **395**

Function calls **373**

Functions

    external (UDF) **382, 875–894**  
         altering a declaration **384**  
         BLOB **889**  
         conditional logic **875**  
         configuration and security issues **383**  
         date and time **881**  
         declaring to a database **383**  
         function identifier (name) **385**  
         library declaration scripts **384**  
         mathematical **876**  
         path names in declarations **386**  
         pre-built libraries **382**  
         sizes of string arguments **385**  
         stability of **383**  
         string and character **884**  
         trigonometrical **890**  
     internal **373, 861–874**  
         binary **873**  
         BLOB **869**  
         CAST() **374**  
         conditional logic **861**  
         conversion **374**  
         date and time **862**  
         EXTRACT() **375**  
         GEN\_ID() **377**  
         LOWER() **377**  
         mathematical **870**  
         miscellaneous **874**  
         string **376**  
         string and character **864**  
         SUBSTRING() **376**  
         trigonometrical **872**  
         UPPER() **376**

## G

Garbage collection **233, 499**  
     performed during backup **233**

gbak

    backup

        arguments **821**  
         cross-version **823**  
         logging progress **825**  
         metadata-only **825**  
         multi-file db to multiple files **824**  
         remote and localhost **825**  
         return codes and feedback **825**  
         running **821**  
         security considerations **825**  
         single-file db to multiple target files **824**  
         switches **821**  
         to a single file **824**  
         transportable **823**  
         with Services Manager **832**  
     backup and restore rights with **820**  
     backup files **818**  
     error messages **833**  
     other talents **819**  
         change default cache size **831**  
         change page size **831**  
         changing database ownership **820**  
         upgrading the ODS **819**

    restore

        arguments **826**  
         intended behaviour **826**  
         logging **830**  
         multiple-file **829**  
         restore or create? **827**  
         return codes and feedback **830**  
         switches **827**  
         syntax **825**  
         to a single file **829**  
         user-defined objects **829**  
         with Services Manager **832**  
     suppressing database triggers **821**  
     user authentication **820**  
     user name and password **820**  
     with Services Manager **831**  
         backup **832**  
         restore **832**

gbak Backup and Restore utility **818–836**

GEN\_ID() expressions **377**

GEN\_ID() function **203**

Generators

    caveats about resetting **204**

- Generators (continued)
    - current value of **204**
    - GEN\_ID() function **203**
    - getting next value for **203**
    - negative stepping **204**
    - setting a new start value for **203**
    - using to populate a variable **204**
  - gfix tool set **701–718**
    - error messages **718**
    - summary of switches and options **716**
  - Global temporary tables **264**
  - GRANT ADMIN ROLE statement **733**
  - GRANT ROLE statement **751**
    - WITH ADMIN OPTION **751**
  - GRANT statement **746**
  - GRANTED BY clause **754**
  - Graphical tools **65**
    - FlameRobin **50, 65**
  - GROUP BY
    - and indexing **269**
  - GROUP BY clause **413**
    - COLLATE sub-clause **418**
    - groupable field list **413**
    - grouping by degree **416**
    - grouping items **415**
    - introduction **308**
    - the HAVING sub-clause **417**
    - with non-aggregating expressions **416**
  - Grouped sets **413**
    - advanced grouping conditions **418**
    - aggregating expressions **414**
    - filtering **417**
    - groupable field list **413**
    - using ORDER BY with **418**
  - gsec utility **734**
    - command-line usage **737**
    - error messages and suggested remedies **739**
    - interactive commands **735**
      - add **736**
      - delete **737**
      - display **735**
      - help **736**
      - modify **736**
      - quit **736**
    - starting an interactive session **735**
      - from a remote station **735**
  - gstat command-line tool **780**
    - analysing statistics **784**
    - index depth statistics **783**
    - on POSIX **780**
    - switches **781**
      - mdata **788**
      - mheader **786**
      - mindex **782**
      - mrecords **789**
    - syntax **780**
  - GTT *see* Global temporary tables **264**
- ## H
- Handling exceptions in PSQL **652**
  - HAVING clause **308, 417**
  - Hexadecimal input **155**
  - Higher-level triggers
    - changing **641**
    - creating **641**
    - session-level **640**
    - transaction-level **640**
  - Hyperthreading support **25**
- ## I
- ib\_udf.sql script **384**
  - ibase.h **855**
  - iberror.h header file **530**
  - ICU character sets usage **163–164**
  - ICU language support libraries **162**
  - IIF() **369**
  - Implicit join vs explicit JOIN **391**
  - Importing data via an external table **255**
  - IN() predicate **359**
  - Index depth **783**
  - Indexes **267–277**
    - altering the structure of **276**
    - and comparisons **270**
    - and JOINS **270**
    - ascending **268, 273**
    - automatic vs user-defined **268**
    - compound **274**
    - COMPUTED BY **273**
    - creating **271**
    - descending **268, 273**
    - directional **268**
    - dropping **276**
    - expression **273**

- Indexes **267–277** (continued)
  - for searches **275**
  - for WHERE criteria **275**
  - housekeeping **276**
  - how they can help **269**
  - importing—DON'T! **268**
  - inspecting **275**
  - maximum per table **267**
  - multi-column **274**
  - query plans **269**
  - setting active/inactive **275**
  - size limit **267**
  - sorting and grouping **269**
  - unique **272**
  - what to index **270**
  - when to index **271**
  - with OR predicates **274**
- inetd daemon **68**
  - inetd.conf **676**
- Initial configuration **60–63**
  - changing configuration parameters **60**
  - configuration files in v.1.0.x **60**
  - environment variables (link) **62**
  - file access parameters **61**
  - Firebird configuration file **60**
  - Firebird root directory **60**
  - parameters of interest **62**
- Inner joins **390**
- Input sets **297**
- INSERT INTO statement **311**
- Inserting rows **311**
  - inserting into ARRAY columns **312**
  - inserting into BLOB columns **312**
  - RETURNING clause **311**
  - RETURNING...INTO **312**
  - with automatic fields **313**
- Installation **23–50**
  - client-only install **41**
    - Linux/UNIX **42**
    - Windows **42**
  - Darwin **36**
  - drives **24**
  - getting a kit **27**
  - Linux **33**
    - Superclassic **35**
    - SYSDBA.password file **34**
  - MacOSX **36**
    - “lipo” packages **37**
  - Installation **23–50** (continued)
    - on old Linux versions **34**
    - on Windows **30**
      - from a zip kit **31**
    - POSIX **33, 36**
      - NPTL **36**
      - old pthreads library **36**
    - RPM (Linux) **33**
    - tarballs **34**
    - uninstalling Firebird **46**
    - Windows
      - client libraries **41**
      - embedded server **45**
      - instclient.exe **44**
    - Windows client
      - fbclient.dll **44**
      - gds32.dll.dll **44**
- Installing a server **29**
- instsvc.exe **70**
- Integer types **114**
  - 64-bit **116**
  - BIGINT **116**
  - division operation **116**
  - INTEGER **116**
  - notation **114**
  - SMALLINT **115**
- Internal functions
  - aggregating **378**
- Intersection table **289**
- IpcName parameter **52**
- IPServer transport channel **52**
- IPServer transport channel **52**
- isql
  - authentication **452**
  - autocommitting DDL **485**
  - AUTODDL **454**
  - BLOBDUMP command **457**
  - BLOBVIEW command **457**
  - command mode **476**
    - command-line switches **477**
    - quick help **477**
  - connecting to a database **451**
  - continuation prompt CON> **452**
  - DDL scripts **457**
  - DDL statements **454**
  - default text editor **449**
  - EDIT command **458**
  - exceptions in **454**

## isql (continued)

EXIT command **458**  
 extracting metadata **478**  
 HELP command **458**  
 host and path names **452**  
 how to exit from **476**  
 how to start **450**  
 INPUT command **459**  
 interactive commands **456**  
 interactive mode **449**  
 OUTPUT command **459**  
 output from SELECT statements **456**  
 quick-start tips **76**  
 QUIT command **459**  
 retrieving the line buffer **454**  
 SET AUTO command **469**  
 SET AUTODDL command **469**  
 SET AUTODDL command (isql) **454**  
 SET BAIL command **469**  
 SET BLOB command **470**  
 SET BLOBDISPLAY command **470**  
 SET COUNT command **470**  
 SET ECHO command **471**  
 SET HEAD[ING] command **471**  
 SET NAMES command **471**  
 SET PLAN command **472**  
 SET PLANONLY command **472**  
 SET ROWCOUNT command **472**  
 SET SQL DIALECT command **455, 473**  
 SET SQLDA\_DISPLAY command **473**  
 SET STATS command **473**  
 SET TERM command **474**  
 SET TIME command **475**  
 SET TRANSACTION command **453, 475**  
 SET WARNING command **454**  
 SET WARNINGS command **476**  
 SET WNG command **476**  
 SHELL command **460**  
 SHOW CHECK command **460**  
 SHOW COLLATIONS command **461**  
 SHOW DATABASE command **461**  
 SHOW DOMAIN[S] command **461**  
 SHOW EXCEPTION[S] command

**462**

## isql (continued)

SHOW FUNCTION[S] command **462**  
 SHOW GENERATOR[S] command **463**  
 SHOW GRANT command **463**  
 SHOW INDEX command **463**  
 SHOW INDICES command **463**  
 SHOW PROCEDURE[S] command **464**  
 SHOW ROLE[S] command **465**  
 SHOW SQL DIALECT command **465**  
 SHOW SYSTEM command **465**  
 SHOW TABLE[S] command **466**  
 SHOW TRIGGER[S] command **467**  
 SHOW VERSION command **468**  
 SHOW VIEW[S] command **468**  
 SQL dialect in **455**  
 stopping query output **456**  
 terminator character **453**  
 transactions in **453**  
 using or disabling warnings **454**

isql SQL utility **449–487****J**

## JOIN clause

SQL-89 implicit inner join **306**  
 use of table aliases **306**

Joining relations **390**

CROSS JOIN **393**  
 equi-joins **396**  
 INNER JOIN **390**  
 named column joins **396**  
 NATURAL joins **397**  
 OUTER joins **393**  
 re-entrant joins **397**

## JOINS

and indexing **270**

joins **394, 395****K**Keeping the OIT and OAT moving **500**Keywords (list) **899–910****L**Language support plug-in **162**

- Limbo transaction **531**
  - resolving **531, 713**
- Linux
  - installation on **33**
  - Red Hat Package Manager (RPM) **33**
- Linux runtimes **34**
- Local Access **52**
- Local access method
  - introduction **11**
- localhost loopback server **38**
- Lock management
  - introduction **10**
- Lock Manager module **792**
  - configuration settings **814**
- Lock Print utility **794**
  - History block **812**
  - interactive reports **796**
  - interactive sampling **813**
  - Lock blocks **804**
  - Lock\_header block **797**
  - Owner blocks **801**
  - Owner flag states **802**
  - reporting to a file **797**
  - Request blocks **811**
  - resource blocks **804**
  - static reports **796**
- Locks
  - table-level **496**
- Log-in
  - under a role **751**
- Lookup tables
  - and referential issues **284**
- LOWER() expressions **377**
- Lowercasing strings **377**
- M**
- Machine specifications **24**
- MacOSX
  - fat client (lipo) packages **37**
  - installation **36**
- Malformed string errors **167**
- Managing database
  - analysing and repairing corruption **712**
  - garbage collection **711**
  - managing shadows **715**
  - resolving limbo transaction **713**
  - sweeping **711**
  - transaction recovery **713**
- Managing database (continued)
  - validating **713**
- Mandatory relationships **291**
  - enforcing with triggers **628**
- Many
  - many relationship **287**
- MATCHED clause **321**
- MATCHING condition in UPDATE OR INSERT **320**
- MERGE statement **320**
  - syntax elements **321**
- Metadata
  - extracting with isql **478**
- Metadata repair **92**
  - gbak method **94**
  - script method **93**
- Metadata text conversion **167**
- MGA
  - optimistic locking **496**
  - post vs commit **494**
  - rollback **495**
  - row locking **495**
  - see* multi-generational architecture **494**
  - successful commits **496**
- Microsoft C++ runtimes **31**
- Migration
  - connection parameters **95**
  - SQL language changes **96**
  - tools for **97**
- Migration tasks **90**
  - metadata repair **92**
  - security database **91**
- Mixed platforms **53, 73**
- MON\$ tables **766**
  - character set of strings **767**
- Monitoring
  - cancelling a running query **768**
  - excluding "Me" **766**
  - how it works **765**
  - mutiple connections **766**
  - security and scope **766**
  - using MON\$ **766**
- Monitoring database activity **765–769**
- Monitoring Locks **791–815**
  - Lock Manager module **792**
  - Lock Print utility **794**



- Monitoring Locks **791–815** (continued)
  - lock table **793**
    - “free” lists **793**
    - deadlocks **794**
    - NULL locks **793**
  - locking in Firebird **791**
- Multi-database applications
  - introduction **6**
- Multi-database transactions **530**
- Multi-generational architecture (MGA) **494**
- Multi-hop
  - vulnerability **727**
- Multi-hop, enabling **689**
- Multiple processors (SMP)
  - introduction **10**

## N

- Named constraints **244**
- Named Pipes **52**
- Named-column joins **396**
- Namespaces for context functions **379**
- NAS (network-attached storage) **55, 73**
- Natural joins **397**
- nBackup
  - “delta” file **839**
  - about **837**
  - backup
    - “difference” backups **840**
    - backup sets **841**
    - stdout **840**
    - user name and password **840**
  - command-line summary **846**
  - file names and locations **839**
  - Freeze utility **843**
    - ending a Freeze **844**
    - not a “fix” at all **844**
    - reset an “unfrozen” database **844**
    - starting a Freeze **843**
    - syntax for **845**
  - limitations **837**
  - making backups **838**
    - command line syntax **839**
  - restoring from nBackup files **842**
    - bad file chains **843**
  - running **838**
- nBackup (continued)
  - SQL support for **845**
    - ALTER DATABASE statement **845**
    - BEGIN BACKUP statement **845**
    - END BACKUP statement **845**
  - suppressing database triggers **839**
  - switch order **840**
- nBackup Incremental Backup tool **837–847**
- Nested sets **299**
- NetBEUI **52**
- Network protocol **38, 51**
  - Named Pipes **52**
  - old Netware not supported **54**
  - TCP/IP **51**
  - WNET **52**
- Network setup **51–59**
  - database path **55**
  - HOSTS file **54**
  - problems with Events **59**
  - RemoteAuxPort **59**
  - server address **54**
  - server name **55**
  - testing connections **57**
- NEXT VALUE FOR expressions **378**
- IS **357**
- MERGE statement
  - WHEN **321**
- Predicates
  - IS **357**
- SQL operators
  - IS **354**
- WHEN **321**
- NPTEL (Native POSIX Threading Library) **36**
- NULL **357**
  - demystifying **109**
  - in calculations **110, 361**
  - in expressions **109, 360**
  - in external functions **362**
  - setting a value to **111, 362**
- NULLIF() **368**
- Number types
  - autoincrementing **116**
  - decimal points **114**
  - fixed decimal **117**
  - floating point **122**
  - identity styled **116**

- Number types (continued)
  - limits **113**
  - mixing fixed and floating point **123**
  - operations **114**
  - scaled decimal **117**

## O

- Object naming conventions **205**
- ON CONNECT trigger **640**
- ON DISCONNECT trigger **640**
- ON TRANSACTION COMMIT trigger **640**
- ON TRANSACTION ROLLBACK trigger **640**
- ON TRANSACTION START trigger **640**
- On-disk structure
  - ODS 10.0 **82**
  - ODS 10.1 **82**
  - ODS 11.0 **83**
  - ODS 11.1 **83**
  - ODS 11.2 **84**
- Operating system **25**
- Operators **350**
- Optimistic locking **496**
- Optimization
  - duplicate chains in index tree **342**
  - getting index statistics **343**
  - housekeeping indexes **340**
  - index selectivity **340**
  - index toolkit **342**
  - optimal indexing **338**
  - using SET STATISTICS **342**
- Optimizer
  - bitmapping of streams **330**
  - BLR **331**
  - cost evaluation factors **330**
  - NATURAL order **339**
  - rivers **331**
  - streams **330**
  - use of indexes **330**
- Optional 1:1 relationship **287**
- ORDER BY
  - and indexing **269, 409**
- ORDER BY clause **309, 409**
  - nulls placement **413**
  - sorting items **409**
  - sorting precedence **410**

- Joining relations
  - FULL **395**
  - LEFT **394**
  - RIGHT **394**
- Outer joins **393**
- Output sets **296**

## P

- Page cache **227**
  - setting at database-level **227, 229**
    - for a single session **230**
    - using the DPB **230**
  - setting at server level **230**
  - verifying size of **231**
- Page size
  - factors influencing choice of **223**
- Parameterized queries **323**
- Parameters
  - using in DML queries **323**
- Pessimistic locking **310, 496, 531**
- PLAN clause **309**
- Platforms
  - mixed **53, 73**
  - porting a database **53**
  - Windows **30**
- Positioned vs searched DML operations **315**
- POSIX
  - Installation on **33**
- Post vs commit **494**
- POST\_EVENT statement **663**
- Precedence of operators **350**
- Pre-defined date literals **104**
- Predicates **346**
  - existential **357**
- PRIMARY KEY
  - adding to an existing table **249**
  - atomicity of **247**
  - syntaxes for declaring **247**
  - using a custom index for **249**
- PRIMARY KEY constraint **245**
- Procedural SQL (PSQL) **541–582**
- Protocol
  - for local access **52**
  - network **38, 51**
- PSQL Error Handling **649–??**
- PSQL Events **659–664**
  - asynchronous signal trap (AST) **663**
  - asynchronous signalling **662**

- PSQL Events **659–664** (continued)
  - elements of **660**
    - application side **660**
    - interface **660**
    - server-side **660**
  - event alerters **663**
  - synchronous listening **661**
  - uses for **659**
  - using a `POST_EVENT` statement **663**
- PSQL exceptions **649**
  - altering custom exceptions **650**
  - `CREATE EXCEPTION` statement **650**
  - creating custom exceptions **650**
  - custom **649**
  - dropping custom exceptions **650**
  - `gds/isc` **649**
  - implementing an error log **656**
  - in triggers **657**
  - nested, as savepoints **654**
  - re-raising **657**
  - run-time messaging with **658**
  - scope of **653**
    - custom exceptions **653**
    - `GDSCODE` **653**
    - `SQLCODE` **653**
    - `SQLSTATE` **653**
  - `SQL` **649**
  - trapping and handling **652**
  - types of **649**
  - `WHEN...DO...` statement **652**
- PSQL Modules
  - ‘Object is in use’ error **580**
  - about database triggers **544**
  - about events **546**
  - about exceptions **546**
  - about executable blocks **544**
  - about relation triggers **543**
  - about stored procedures **542**
  - adding comments **577**
  - altering and dropping **579**
  - assignment statement **561**
  - body elements **549**
  - case-sensitivity **577**
  - `COLLATE` in variable declaration **559**
  - colon (`:`) marker for variables **561**
  - compiling **579**
  - context variables **560**
  - `CREATE` statements **548**
  - `DECLARE VARIABLE` **555**
  - `DECLARE...AS CURSOR FOR` **573**
  - declaring variable types as column reference **558**
  - declaring variable types as domain **558**
  - default input values **559**
  - deleting source from **580**
  - `DELETING` variable in triggers **560**
  - dependencies **579**
  - developing **576**
  - editing tools **578**
  - effects of changes to **581**
  - elements of **547**
  - `EXECUTE STATEMENT` **567**
    - caveats **571**
    - using parameters with **569**
  - flow statements **564**
    - `BREAK` **566**
    - breaking out of a block **565**
    - `EXCEPTION` **567**
    - `EXIT` **565**
    - labelled loops **566**
    - `LEAVE` **565**
    - `POST_EVENT` **567**
  - `FOR SELECT...` loops **563**
  - header elements **549**
  - inserting variable in triggers **560**
  - internals **581**
  - language elements **550**
  - language extensions for **544**
  - managing code **578**
  - named cursor (explicit) **573**
  - named cursor (undeclared) **572**
  - `NOT NULL` in variable declaration **560**
  - object dependencies **579**
  - `OLD` and `NEW` column arrays **560**
  - overview **541**
  - programming constructs **552**
    - `BEGIN...END` blocks **552**
    - conditional blocks **552**
    - `IF...THEN...ELSE` **552**
    - `WHILE...DO` **553**
  - restrictions on **545**
  - retrieving multiple rows into **562**

PSQL Modules (continued)  
     retrieving singleton rows into **562**  
     ROW\_COUNT context variable **560**  
     security **546**  
     SELECT...INTO statement **562**  
     size **577**  
     statement terminator in **547**  
     SUSPEND statement **564**  
     UPDATING variable in triggers **560**  
     using cursors **571**  
     variable types **555**  
     white space in **577**  
 PSQL modules  
     trapping and handling exceptions in **652**  
 pthreads library (old POSIX platforms) **36**

## Q

Query plan **328**  
     elements of **329**  
     examples of **332–337**  
     for view **438**  
     introduction **269**  
     plan expressions **328**  
     specifying your own **337**  
 Query plans and optimizer **328–343**  
 Querying multiple tables **387–406**  
     joining **388, 390**  
     subqueries **388, 399**  
     unions **388, 402**  
     using relation aliases **388**  
 Quote-delimited identifiers **105**

## R

RDB\$ADMIN role **731, 733, 754**  
     ALTER ROLE statement **755**  
         SET AUTO ADMIN MAPPING  
             **755**  
         extending Windows administrator  
         powers **754**  
 RDB\$DB\_KEY **607–611**  
     duration of validity **610**  
     use in stored procedure **607**  
     with multi-table sets **611**  
 RDB\$FIELDS **237**  
 RDB\$GET\_CONTEXT expressions **379,**  
     **380**  
 RDB\$RELATION\_FIELDS **237**  
 RDB\$RELATIONS **237**

RDB\$SET\_CONTEXT expressions **379,**  
     **381**  
 Read Committed isolation **508**  
 Read-only database **231**  
     how to make it so **232**  
 RECREATE PROCEDURE statement **612**  
 RECREATE TABLE statement **263**  
 RECREATE TRIGGER statement **637,**  
     **639**  
 RECREATE VIEW statement **435**  
 Recursive queries **406**  
 Recursive stored procedures **592**  
 Re-entrant joins **397**  
 REFERENCES clause (foreign key  
     definition) **281**  
 REFERENCES privilege **286, 747**  
 Referential constraints **243**  
 Referential integrity **279–??**  
     ‘Object is in use’ error **291**  
     and lookup tables **284**  
     custom action triggers **284**  
     dealing with a circular reference **288**  
     FOREIGN KEY constraint **280**  
         defining **281**  
     implementing parent-child relationship  
         **280**  
     introduction **201**  
     mandatory relationships **291**  
     many-to-many relationship **287**  
     optional one-to-one relationship **287**  
     payback **279**  
     self-referencing relationship **290**  
     the REFERENCES privilege **286**  
     using an intersection table **289**  
 Regular expressions **894–897**  
     characters **894**  
         character classes **895**  
         wildcards **894**  
     escaping special characters **897**  
     OR-ing terms **897**  
     quantifiers **896**  
     sub-expressions **897**  
 Relation aliases **388**  
     legal names **389**  
 Relation triggers  
     about **543**  
 RELEASE SAVEPOINT statement **524**

- RemoteAuxPort **59**
  - introduction **13**
- Replaceable parameters in DML queries **323**
- Replication **852**
  - for “warm backups” **852**
  - third-party modules **852**
- Reserved words (list) **899–910**
- Resource usage
  - introduction **8**
- RETURNING clause **311, 318**
- REVOKE ADMIN ROLE statement **733**
- REVOKE ALL statement **757**
- REVOKE statement **756**
- Revoking SQL privileges **756**
- Right joins **394**
- Rivers and streams **298**
- Rollback **495**
- ROLLBACK statement **527**
- ROLLBACK TO SAVEPOINT statement **524**
- ROLLBACK WITH RETAIN statement **528**
- Root directory (Firebird) **60**
- Row locking **495**
- ROWS set quantifier **303**
- Running on MacOSX **72**
  - Classic **72**
  - Superserver and Superclassic **72**
- Running on POSIX **65**
  - Classic **68**
    - stopping and starting **68**
  - Superserver and Superclassic **66**
    - fbmgr utility **66**
- Running on Windows **69**
  - as a service **70**
    - stopping and starting **70**
    - using instsvc.exe **70**
  - as an application **71**
  - Guardian **69**
    - Don’t use with Classic **70**
- Runtime libraries
  - glibc on Linux **91**
  - libstds++.so.5 **34**
  - Microsoft C++ **31, 91**
- Run-time PSQL **615**

## S

- SAN (storage area network) devices **55, 73**
- SAVEPOINT statement **524**
- Savepoints **524**
  - releasing **524**
  - rolling back to **524**
- Scripts **480, 481**
  - basic steps **483**
  - chaining together **486**
  - comments in **481**
  - committing work in **485**
  - creating a script file **484**
  - executing **486**
  - managing errors **483**
  - managing your schema with **486**
  - manual scripting **487**
  - terminator symbols in **482**
  - terminators with PSQL definitions **482**
  - why use? **480**
- Search conditions **369**
  - using a subquery for **401**
- Searched vs positioned DML operations **315**
- Security database
  - migrating **91**
  - upgrade script for **1011**
- Security scripts **759**
- SELECT ALL **302**
- SELECT DISTINCT **302**
- SELECT statement **301**
  - clauses in **302**
  - FOR UPDATE clause **310**
  - FROM clause **305**
  - GROUP BY clause (introduction) **308**
  - HAVING clause **308**
  - JOIN clause **306**
  - optional set quantifiers **302**
    - ALL **302**
    - DISTINCT **302**
    - FIRST...SKIP **302**
    - ROWS **303**
  - ORDER BY clause **309**
  - output list specification **303**
  - PLAN clause **309**
  - search conditions **307**
  - UNION clauses (introduction) **309**
  - WHERE clause **307**
    - parameterized **307**

- SELECT statement **301** (continued)
  - WHERE clause **307**
    - using parameters **307**
  - WITH LOCK sub-clause **310**
- Selectable procedures
  - calling **598**
  - introduction **326**
- Self-referencing relationship **290**
- Sequences
  - caveats about resetting **204**
  - NEXT VALUE FOR **203, 378**
  - setting a new start value for **203**
  - using to populate a variable **204**
- Server Control applet (on Windows) **39**
- Server management
  - shutdown and restart **705**
- Server memory **23**
- Server models
  - page cache size differences **12**
- Server security **721–739**
  - assigning RDB\$ADMIN privileges **733**
  - backups **722**
  - DDoS attacks **727**
  - dedicated host machines **726**
  - embedded on Windows **725**
  - execution of arbitrary code **725**
  - filesystems **722**
  - Firebird “native” users **729**
  - firewalls **727**
  - gsec options **738**
  - introduction **7**
  - maintaining user credentials with gsec **734**
  - maintaining user credentials with SQL **732**
    - escalating privileges **733**
  - managing user access **728**
  - over the wire **726**
  - physical environment **721**
  - platform users **729**
    - POSIX **729**
    - Windows **730**
  - platform-based **722**
  - privileged users **731**
    - platform Superusers **732**
    - RDB\$ADMIN role **731**
    - SYSDBA **732**
- Server security **721–739** (continued)
  - rescinding RDB\$ADMIN privileges **734**
  - security database **728**
    - migrating **729**
  - server multi-hop **727**
  - storage of USER records **729**
  - the view USERS **738**
  - trustworthiness of clients **726**
  - user maintenance **729**
  - web applications and other n-tier **726**
- Service parameter block (SPB) **853**
- Services applet (on Windows) **40**
- Services file **679**
  - on POSIX **679**
  - on Windows **679**
- Services Manager **853–857**
  - clients **854**
  - fbsvcmgr utility **854**
    - API functions available **857**
    - modified SPB params **855**
  - headers in ibase.h **855**
  - service parameter block (SPB) **853**
    - syntax **855**
- SET AUTO ADMIN MAPPING
  - argument for ALTER ROLE **755**
- SET GENERATOR statement **203**
- SET SQL DIALECT command (isql) **455**
- SET STATISTICS statement **342, 474**
  - don’t confuse with SET STATS (isql) **474**
- SET TRANSACTION command (isql) **453**
- SET WARNING command (isql) **454**
- Sets
  - aggregated **298**
  - cardinality and degree **296**
  - concept of **295**
  - cursor **298**
  - grouped **298**
  - input **297**
  - nested **299**
  - output **296**
- Setting a value to NULL **111**
- Shadow database **236**
- Shadowing **847**
  - benefits and limitations **847**
  - conditional **849**
  - CREATE SHADOW statement **849**

- Shadowing **847** (continued)
  - creating in manual mode **850**
  - DROP SHADOW statement **851**
  - dropping a shadow **851**
    - using the gfix utility **852**
  - to implement **848**
- SHOW INDEX command (isql) **275**
- Single-user model **18**
- SINGULAR predicate **359**
- SMP
  - introduction **10**
- SMP support **25**
- Snapshot isolation **509**
- Snapshot Table Stability isolation **509**
- SOME predicate **359**
- Sort memory **408**
- Sort space on disk **408**
- Sorted sets **407**
  - degree number **411**
  - indexing for **409**
  - intermediate sets **408**
  - nulls placement **413**
  - ORDER BY clause **409**
  - ordering a UNION **412**
  - presentation order of clauses for **407**
  - sort direction **412**
  - sorting by expression **410**
  - sorting items **409**
  - syntax skeleton **407**
  - temporary sort space **408**
  - use of relation aliases **410**
- Special character sets **154**
  - ASCII **155**
  - NONE **154**
  - OCTETS | BINARY **154**
  - UNICODE\_FSS **155**
- SQL “dialect” **87, 102, 208, 221**
- SQL operators **350**
  - AND **356**
  - arithmetic **351**
    - precedence of **351**
  - BETWEEN **353**
  - character set vs collation **352**
  - comparison **351**
  - concatenation operator || **351**
  - CONTAINING **353**
  - IN **354**
  - SQL operators **350** (continued)
    - LIKE **355**
    - logical **356**
    - NOT **356**
    - OR **356**
    - precedence of **350**
    - SIMILAR TO **355**
- SQL privileges
  - ALL **743**
  - bundling multiple privileges **749**
    - ALL privilege **750**
    - lists **749**
    - using roles **750**
  - CREATE ROLE statement **750**
  - creating a role **750**
  - DELETE **743**
  - DROP ROLE statement **751**
  - dropping a role **751**
  - EXECUTE **743, 748**
  - for multiple users **751**
    - to a list of procedures **752**
    - to a list of users **752**
    - to a POSIX group **752**
    - to any native user (PUBLIC) **752**
    - TO PUBLIC **752**
  - for ordinary users **741**
  - for views, to limit access **747**
  - GRANT ROLE statement **751**
  - GRANT statement **746**
  - granting **746**
  - granting a role to a user **751**
    - WITH ADMIN OPTION **751**
  - in embedded applications **744**
  - INSERT **743**
  - installing from a procedure **762**
  - loading a role with privileges **750**
  - needed by objects **748**
  - objects of permissions **743**
  - on system metadata objects **742**
  - on views **749**
  - planning an access scheme **742**
  - REFERENCES **286, 743, 747**
  - restrictions **745**
  - REVOKE ALL statement **757**
  - REVOKE statement **756**
    - on multiple privileges **757**
    - simplified syntax **756**

## SQL privileges (continued)

- revoking **756**
  - EXECUTE privileges **758**
  - from grantees **758**
    - from a list **758**
    - from a role **758**
    - from a role-user **758**
    - from an object **759**
    - from user PUBLIC **759**
  - grant authority **759**
  - WITH GRANT OPTION **759**
- rights to grant privileges **753**
  - GRANTED BY **754**
    - on behalf of another **753**
- ROLE **743**
- scripting **759**
- SELECT **743**
- system role RDB\$ADMIN **754**
- types of users **743**
  - database objects as users **745**
  - native Firebird users **744**
  - POSIX users and groups **744**
  - PUBLIC **744**
  - users with escalated privileges **745**
  - Windows trusted users **744**
- unintended effects **755**
- UPDATE **743**
- UPDATE rights on columns **746**
- what are they? **742**
- WITH GRANT OPTION **753**
- SQL-89 implicit inner join **391**
- SQL-92 explicit inner join **391**
- SQLSTATE codes list **979**
- Statement-level locking **532**
  - “dummy update” hack **533**
  - FOR UPDATE clause **536**
  - WITH LOCK clause **534**
- Stored procedure
  - installing SQL privileges from **762**
- Stored procedures **583–618**
  - about **542**
  - ALTER PROCEDURE statement **612**
  - body elements **585**
  - combining executable and selectable **607**
  - CREATE OR ALTER PROCEDURE statement **613**

Stored procedures **583–618** (continued)

- CREATE PROCEDURE statement **614**
  - 584**
  - creating **584**
- DROP PROCEDURE statement **614**
- executable **583, 587**
  - calling **591**
- EXIT vs SUSPEND **592**
- header elements **584**
- local variables **585**
- main code block **586**
- nested **599**
- operations in **588**
- outputs and exits **592**
- processing in the loop **596**
- RECREATE PROCEDURE statement **612**
- recursive **592**
- RETURNING\_VALUES **602**
- rules for input arguments **591**
- SELECT...INTO... **590**
- selectable **583, 594**
  - calling **598**
  - technique **595**
- to change **611**
- to drop **614**
- to view an array column **605**
- transaction context **536**
- using RDB\$DB\_KEY **607**
- using the colon prefix for variables **591**
- with nested SELECTs **597**
- with running total **603**
- Streams and rivers **298**
- String concatenation **144**
- String functions **145**
- Strings **145**
  - case in **145**
  - character set of string literals **154**
  - length
    - BIT\_LENGTH **145**
    - CHAR\_LENGTH **145**
    - OCTET\_LENGTH **145**
  - metadata text conversion **167**
  - NULL inputs **145**
  - string functions **145**
  - SUBSTRING function **145**
  - TRIM function **145**



- Subqueries **399**
  - correlated **399**
  - the derived table **402**
  - use JOIN instead? **400**
  - using to search **401**
  - using to specify a column **399**
  - using with joins for INSERTs **401**

- Subquery
  - correlated **300**

- Sub-selects **399**

- SUBSTRING() expressions **376**

- Superclassic
  - install and configure **35**
  - MacOSX
    - enabling **37**
  - not for 32-bit systems **29**
  - on POSIX **66**
  - POSIX
    - enabling script **35**
    - fb\_smp\_server **35**

- Superserver **36**
  - MacOSX **37**
  - on POSIX **66**

- Sweep interval **225**

- Sweeping databases **233**

- Sweeping vs garbage collection **502**

- SYSDBA password **75**
  - temporary, Linux **34**

- SYSTEM namespace **379**
  - variables available **379**

- System requirements **23**

- System tables **917–949**
  - metadata **917–939**
  - views over **940–941**
  - monitoring **942–949**

## T

- Table reservation **511**
  - parameters for **512**
  - summary of settings for **512**
  - uses for **511**

- Table-level locking **532**
  - RESERVING clause **532**

- Table-level locks **496**

- Table-level triggers **619**

- Tables **237–266**
  - adding a column **261**

- Tables **237–266** (continued)
  - altering **258**
    - altering columns **258**
  - constraints **243**
    - CHECK **251**
    - choosing a primary key **245**
    - composite primary keys **247**
    - integrity **243**
    - named **244**
    - NOT NULL **244**
    - PRIMARY KEY **245**
    - PRIMARY KEY and FOREIGN KEY names **244**
    - referential **243**
    - scope of **243**
    - UNIQUE **250**
    - validation checks **251**
  - creating **238**
  - defining columns **238**
  - dropping **263**
  - external **253**
    - end-of-line characters **254**
    - format of **254**
    - operations on **255**
    - securing **254**
  - GTTs
    - creating **264**
  - GTTs (global temporary tables) **264**
  - ownership **238**
  - physical storage of **237**
  - RECREATE TABLE **263**
  - required attributes **239**
  - surrogate keys **246**
  - temporary **263**
    - for older versions **265**
  - tree structures **266**
- TCP/IP **51**
- TCP/IP local loopback **38**
- Terminal Server **20**
- Text BLOBs
  - malformed strings returned from **167**
- Trace and Audit **769–780**
  - AuditTraceConfigFile parameter **775**
  - fbtrace.dll plug-in **780**
  - fbtrace.so plug-in **780**
  - plug-in facilities **780**
  - scope restriction on Windows **770**

Trace and Audit **769–780** (continued)

- system audit session **775**
- system audit tracing **769**
- trace configuration file **775**
- trace on demand **770**
- trace over time **770**
- trace session **770**
- user trace session **775**
  - abnormal endings **777**
  - configuration samples **777**
  - fbtracemgr utility **778**
  - user interfaces for **776**
    - fbtracemgr utility **778**
  - workings of **776**

## Transaction

- access mode READ ONLY **510**
- access mode READ WRITE **510**
- aging **497**
- api\_commit\_transaction **520**
- atomicity of **494**
- COMMIT statement **526**
- COMMIT WITH RETAIN statement **526**
- Concurrency isolation **509**
- Consistency isolation **509**
- context of **493**
- dead **499**
- default **505**
- diagnosing exceptions **528**
- ending **526**
- explicit locking **534**
- handle **520**
- ID and age **497**
- ID overflow **497**
- IGNORE LIMBO **513**
- interesting **498**
- isc\_start\_transaction **520**
- isolation level **507**
- keeping the OIT and OAT moving **500**
- limbo **499, 531**
- lock timeout **510**
- locking and lock conflicts **514**
- locking policy (WAIT | NO WAIT) **509**
- logical context **525**
- NO AUTO UNDO **513**
- NO WAIT locking policy **510**
- oldest active (OAT) **498, 538**

## Transaction (continued)

- oldest interesting (OIT) **498**
- pessimistic locking **514**
- progress of **523**
- Read Committed isolation **508**
- read-only **498**
- receiving exceptions **529**
- record versions **513**
- RESERVING clause **511**
- role of client **493**
- role of server **493**
- ROLLBACK statement **527**
- ROLLBACK WITH RETAIN statement **528**
- snapshot isolation **509**
- Snapshot Table Stability isolation **509**
- starting **520**
- state bitmap **500**
- statement-level locking **532**
- statistics **497, 502**
- sweeping vs garbage collection **502**
- table reservation **511**
- table-level locking **509, 532**
- the gap **502**
- transaction parameter buffer (TPB) **521**
- user savepoints **524**
- WAIT locking policy **510**
- WITH LOCK clause **514**

Transaction Server **20**

## Transactions

- auto undo log **525**
- autonomous **537**
- concurrency **506**
- configuring **505–??**
- CURRENT\_TRANSACTION **522**
- deadlock **516**
- deadly embrace **516**
- explicit row locking **514**
- garbage collection of **499**
- in isql **453**
- in the API **520**
- internal savepoints **525**
- livelock **516**
- lock conflicts **515**
- multi-database **530**
- nested **523**
- optimizing behaviour of **537**
- overview **491–504**

Transactions (continued)  
     pessimistic locking **531**  
     programming with **519–538**  
     releasing a savepoint **524**  
     resolving limbo **531**  
     rolling back to a savepoint **524**  
     stored procedure context **536**  
     trigger context **537**  
     two-phase commit **714**

Tree structures **266**

Trigger  
     transaction context **537**

Triggers **619–641**  
     ‘Object is in use’ error **639**  
     “database” **620, 639**  
     about **543**  
     active/inactive **622**  
     ALTER TRIGGER statement **622, 637**  
     alternative syntax for creating **623**  
     attributes **623**  
     body elements **624**  
     classes of **619**  
     conditional **624**  
     CREATE OR ALTER TRIGGER  
         statement **637**  
     CREATE TRIGGER statement **622**  
     creating **622**  
     DDL **620**  
     deactivating **622**  
     DROP TRIGGER statement **641**  
     dropping **641**  
     exceptions in **657**  
     for “auto-stamping” **627**  
     for applying default values **627**  
     for data transformations **625**  
     for data validation **626**  
     for enforcing a mandatory relationship  
         **628**  
     for referential integrity support **631**  
     for self-referencing tables **635**  
     for updating other tables **628**  
     header elements **623**  
     higher-level **620, 639**  
     INSERTING,                   UPDATING,  
         DELETING variables **624**  
     introduction **5**  
     multi-action **624**

Triggers **619–641** (continued)  
     naming **623**  
     NEW and OLD context arrays **624**  
     phase and event **620**  
     RECREATE TRIGGER statement  
         **637**  
     sequence **621**  
     special PSQL for **624**  
     table-level **619**  
     to change **637**  
     to implement auto-incrementing keys  
         **635**  
     updating rows in the same table **634**  
 Trusted User authentication (Win) **730**  
     configuring **730**  
     forcing use of **731**  
     size of user names **731**  
 Truth testers  
     operators **347**  
     what is true? **348**  
 Two-phase commit **714**

## U

UDF **382**  
 UdfAccess parameter **386**  
 Uninstall **30**  
 Uninstalling Firebird **46**  
     Linux **46**  
     MacOSX/Darwin **48**  
     Windows **47**  
 UNION ALL **403**  
 UNION clauses (introduction) **309**  
 UNION DISTINCT **403**  
 UNION queries **402**  
     compatible sets **402**  
     recursive **406**  
     re-entrant **405**  
     search and ordering conditions **405**  
     UNION ALL | DISTINCT **403**  
     using run-time columns in **404**  
 Union-compatible sets **402**  
 UNIQUE constraint **250**  
 UPDATE OR INSERT statement **319**  
     MATCHING condition **320**  
     syntax elements **319**  
 UPDATE statement **311, 315**  
     RETURNING clause **318**

- UPDATE statement **311, 315** (continued)
  - RETURNING...INTO **319**
  - syntax **316**
  - the SET clause **316**
  - value-switching issues **317**
  - WHERE clause **318**
- Upgrade scripts
  - character metadata **1014**
  - security database **1011**
- UPPER() expressions **376**
- Uppercasing strings **376**
- User accounts
  - elevating privileges **216**
  - managing through DDL **215**
- User name and password
  - changing **49**
  - default **48**
- USER\_SESSION namespace **379**
- USER\_TRANSACTION namespace **379**
- User-defined functions **382**
  - introduction **6**
- Utilities
  - fb\_lock\_print **794**
  - fbsvcmgr **854**
  - fbtracemgr **778**
  - gbak **818**
  - gsec **734**
  - gstat **780**
  - isql **449**
  - nBackup **837**

## V

- Validation and repair **234, 713**
- Validation constraints
  - on domains **184**
  - on tables **251**
- Value-switching in UPDATE statements **317**

- Verifying page cache size **231**

- Views **426–439**

- about **426**
- computed columns in **429**
- CREATE VIEW statement **427**
- creating **427**
- dropping **436**
- inferred column names **427**
- keys and indexes **427**
- making updatable **433**

- Views **426–439** (continued)
  - modifying **435**
  - naturally updatable **435**
  - ordering and grouping **427**
  - privileges for **436**
  - read-only **432**
  - some simple specifications **431**
  - specified column names **428**
  - the SELECT specification **429**
  - unions in **429**
  - uses for **431**
  - using in SQL **437**
  - using query plans for **438**
  - WITH CHECK OPTION **430**
- Virtual set objects **446**

## W

- Well-formed strings **152**
- WHERE clause **307, 369**
  - UPDATE statement **318**
- Windows
  - Very old versions (98x, ME, XP Home) **40**
- Windows “local” protocol configuration issues **52**
- Windows “local” transport channel **52**
- SQL operators
  - STARTING **356**
- STARTING **356**
- WITH CHECK OPTION in views **430**
- WITH GRANT OPTION
  - revoking **759**
- WITH GRANT OPTION clause **753**
- WITH LOCK clause **534**
- WITH LOCK sub-clause **310**
- WNET **52**
  - connection string for customised port service **678**
  - redirecting the port service **676**

## X

- xinetd daemon **68**
  - xinetd.conf **676**
- XNET **52**
- XSQLVAR sqlsubtype **154**

## Z

- Zip kit (Windows) **31**

## APPENDIX

## XIV

## RESOURCES

In this Appendix is an eclectic collection of resources that you might not know about if you've only recently discovered Firebird.

Use the alphabetical Glossary at the end of the book to look up any unfamiliar terms.

## Free Documentation

---

The *Language Reference* volume of the original InterBase 6 beta documentation set from 2000 is your starting point. Still available from archives as a PDF book, it comprehensively covers all of the SQL elements that were implemented at the point where Firebird began.

Since then, members of the Firebird open source project, most notably Paul Vinkenoog, have written updates covering changes and new implementations in Firebird's SQL language over the many versions. There is one update book for each major version from v.1.5 onward, each accumulating onto the previous update. That means, for example, that if you are working with Firebird 2.5, you need only the original InterBase 6 **Language Reference** and the **Firebird 2.5 Language Reference Update** to have the complete language documentation.

Links for downloading these and many other useful documents can be found at <http://www.firebirdsql.org/en/reference-manuals/>

## Books

---

If your previous contact with SQL has been minimal, a good book on SQL basics is invaluable. The following list—not exhaustive—may be helpful.

**Joe Celko** writes SQL books aimed at problem-solving. The SQL is mostly standard, with perhaps a bias toward Oracle. Titles:

*SQL For Smarties: Advanced SQL Programming*

Classic book of magic spells for SQL programming.

*Data and Databases: Concepts in Practice**SQL Puzzles and Answers*

*The Essence of SQL* by **David Rozenstein and Tom Bondur**: Very concise, very much a beginner book.

*SQL For Dummies, 7th Edition* by **Allen G. Taylor**: Approachable and standards-friendly introduction to basic language elements, with usable examples.

*The Practical SQL Handbook* by **Judith S. Bowman** et al.: How-to and desktop reference for standard SQL, well-reviewed.

## Free help

---

The Firebird community is known for its free peer support. The Firebird Project itself manages a number of mail lists where resident experts can answer questions at all levels. Some run from the Sourceforge list servers while others run as Yahoo groups.

You can subscribe to any of these lists directly from the page <http://www.firebirdsql.org/en/mailling-lists/> at the Firebird web site. At that page, you will also find links to servers that have the list archives available with keyword searching.

## SQL and Firebird server support

---

For all Firebird SQL-related questions, join the firebird-support forum at <http://tech.groups.yahoo.com/group/firebird-support/>. This is a volunteer email list where long-time and new Firebird users share experience. You can also subscribe from the link at the top of this topic.

## Client interface support

---

Separate lists exist for most of the application language interfaces that wrap the Firebird client API. At the page linked at the top of this topic you can find lists for Java (firebird-java), ADO.NET (firebird-net-provider), ODBC (firebird-odbc-devel), PHP (firebird-php) and Python (firebird-python). For Delphi and Lazarus developers, the owners of IBOjects ([www.ibobjects.com](http://www.ibobjects.com)), FIBPlus ([www.devrace.com](http://www.devrace.com)) and UIB Components ([www.progdigy.com/forums/](http://www.progdigy.com/forums/)) all run free support lists or newsgroups with registration available at their websites.

A fuller range of connectivity options can be found through links at <http://www.ibphoenix.com/download/connectivity/>

## Third-Party Tools

---

The following is a selection of established tools that developers and DBAs are likely to find useful. It is far from exhaustive—the field is huge. Many more options can be found at a links section at the IBPhoenix web-site: start at <http://www.ibphoenix.com/download/tools/>.

**Table 2.1** Third-party tools for use with Firebird

| Tool                      | Vendor        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <u><i>dbBackup</i></u>    | IBPhoenix     | Multi-faceted administration front-end for configuring, managing and logging <i>gbak</i> backups locally or remotely for one or many databases on multiple platforms. Includes scheduling, optional zip compression, automatic restart after an interruption and many more DBA-friendly features. Commercial:one-off lifetime license fee, free trial available                                                                                                   |
| <u><i>dbFile</i></u>      | IBPhoenix     | A cross-platform command-line tool for importing and exporting data between Firebird databases and a range of external data file formats. The supported formats are delimited data-sensitive, fixed position data-sensitive and fixed format text (fixed field position and record size). Available for Linux, MacOSX and Windows. IBPhoenix can build <i>dbFile</i> to order for other platforms. Commercial:one-off lifetime license fee, free trial available. |
| <u><i>FBDataGuard</i></u> | IBSurgeon Ltd | Monitors databases and the server environment, captures statistics and sends alerts when possible problems appear; performs backups. Acts as an automatic administrative assistant and has tools for recovering data from seriously corrupted databases.                                                                                                                                                                                                          |
| <u><i>FBFirstAID</i></u>  | IBSurgeon Ltd | Automatic diagnosis and repair of corrupted databases that Firebird's own tools cannot handle. Commercial. Free trial of analysis tool available.                                                                                                                                                                                                                                                                                                                 |
| <u><i>FBScanner</i></u>   | IBSurgeon Ltd | Profiler for database applications using TCP/IP networks, monitoring user activity, managing database connections and troubleshooting network "Connection reset by peer" errors directly through Firebird's network layer. Can audit applications and tune performance, audit applications, tune performance and log SQL queries, transactions and connections.                                                                                                   |

| Tool                                | Vendor              | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------------------------|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <u>FB TraceManager</u>              | Upscene Productions | A commercial product to ease the use of the new Firebird 2.5 Trace API in various levels. The integrated parser processes the Trace API raw output and optional storage of parsed trace data in the FB TraceManager database enables further reporting and analysis. The Event Processing Engine allows to define customizable script-based event rules to spot user-defined conditions in near real-time. Additionally, beside the Trace API, per-database monitoring capabilities makes the product an integrated toolset for database and server monitoring. |
| <u>IBAnalyst</u>                    | IBSurgeon Ltd       | Administrator's graphical toolset for analysing detailed database statistics and identifying possible problems with database performance, maintenance and interaction with applications. Outputs improvement suggestions automatically, according to state of statistics.                                                                                                                                                                                                                                                                                       |
| <u>IB LogManager Product Family</u> | Upscene Productions | Commercial set of tools to integrate a trigger-based, server-side auditing mechanism into a Firebird database to keep track of data changes. The additional addons <i>IBLMPump</i> and <i>IBLMRedo_cmd</i> can help to set up and maintain a master-to-many-slave database standby environment.                                                                                                                                                                                                                                                                 |

(continued next page)



| Tool                        | Vendor    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <u>IB Replication Suite</u> | IBPhoenix | <p>A tool set for implementing and controlling database replication in all Firebird versions. Replicator integrates with databases, allowing users to replicate data seamlessly and transparently between databases on both local and remote sites.</p> <p>The Server component runs as a service (or a daemon) on Windows and Linux 32-bit and 64-bit servers, as a 32-bit application. A 64-bit version is available for Linux. Versions for InterBase v.5+ and Oracle v.9+ are also available.</p> <p>The graphical administrative component, Replication Manager, is a Windows client application that can interact with Server components running on any host platform.</p> <p>A database that is configured as a participant in replication is called a “replicant”. Any “source” or “target” database may be a replicant, including InterBase 5+ and Oracle 9+ databases, even a specially-formatted text file, provided the Server component can find the required replicant licences installed and available.</p> <p>A wide variety of replication topographies is achievable and multiple configurations can be defined for running at different times or simultaneously.</p> <p>Commercial: Server and Replicant licences are purchased according to the configuration (or configurations) of servers and replicants.</p> |

---

## Commercial Help and Support

---

The most prominent commercial provider of support and consultancy related to Firebird is IBPhoenix. It offers a variety of support plans and operates, via the Internet, in most countries world-wide.

IBSurgeon Ltd specialises in recovering severely corrupted databases, a service it can provide either by working on a file copy sent by international courier or, if urgency or security restraints require it, by remote-connecting directly to your system.

Its other prime service is optimization of the structure of a poorly-performing database and its physical environment.





The  
**Firebird Book**  
A Reference for Database Developers

SECOND EDITION

# GLOSSARY





## THE FIREBIRD BOOK SECOND EDITION

# Glossary

|                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>ADO</b>                                     | Acronym for ActiveX Data Objects, a high-level application-to-data source interface introduced by Microsoft in 1996. Earlier versions could only access relational databases with fixed columns and data types but later versions could interface also with other DBMS models, filesystems, data files, email messages, tree structures and heterogeneous data structures.                                                                                                                                                                                                                                                                                   |
| <b>Aggregate (function)</b>                    | <p>An aggregate function returns a result derived from a calculation that aggregates (collects together) values from a set of rows that is grouped in some fashion by the syntax of the SQL statement.</p> <p>For example, the internal function SUM() operates on a non-null numerical column to return the result of adding all of the values in that column together for all of the rows cited by a WHERE or GROUP BY clause. Output which is aggregated by a WHERE clause returns one row of output, whereas that aggregated by a GROUP BY clause potentially returns multiple rows.</p>                                                                 |
| <b>Alerter (events)</b>                        | A term coined to represent a client routine or class which is capable of “listening” for specific database EVENTs generated from triggers or stored procedures running on the server.                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>‘ALICE’</b>                                 | Internal name for the source code for the <i>gfx</i> utilities - a corruption of the words “all else”.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Alternative key (‘alternate key’)</b>       | This is a term used for a unique key that is not the primary key. A unique key is created by applying the UNIQUE constraint to a column or group of columns. A foreign key in a formal referential integrity relationship can link its REFERENCES clause to an alternative key.                                                                                                                                                                                                                                                                                                                                                                              |
| <b>API</b>                                     | See <i>Application Programming Interface</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Application Programming Interface (API)</b> | An application programming interface (API) provides a set of formal structures through which application programs can communicate with functions internal to another software. The Firebird API surfaces such an interface as a dynamically-loaded client library compiled specifically for each supported platform. The Firebird API structures are C structures, designed to be translatable to virtually any host application language. Translations can be found for Java, Pascal, Perl and, to some degree, PHP 4, Python and others.                                                                                                                   |
| <b>Argument</b>                                | <p>An argument is a value of a prescribed data type and size which is passed to a function or stored procedure to be operated upon. Stored procedures can be designed to both accept input arguments and return output arguments. For the returned values of functions (both internal and user-defined) the term result is more commonly used than argument</p> <p>The terms <i>parameter</i> and <i>argument</i> are often used interchangeably with regard to stored procedures, thanks to Borland’s adoption of the term <i>parameter</i> in its Delphi data access classes to name the properties to which stored procedure arguments are assigned..</p> |
| <b>Array slice</b>                             | A contiguous range of elements from a Firebird array is known as a <i>slice</i> . A slice can consist of any contiguous block of data from an array, from a single element of one dimension to the maximum number of elements of all defined dimensions.                                                                                                                                                                                                                                                                                                                                                                                                     |

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Atomicity</b>                       | <p>In the context of transactions, <i>atomicity</i> refers to the character of the transaction mechanism that wraps a grouping of changes to rows in one or more tables to form a single unit of work that is either committed entirely or rolled back entirely.</p> <p>In the context of a key, a key is said to be <i>atomic</i> if its value has no meaning as data.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>AutoCommit</b>                      | <p>When a change is posted to the database, it will not become permanent until the transaction in which it was posted is committed by the client application. If the client rolls back the transaction, instead of committing it, the posted changes will be cancelled.</p> <p>Some client development tools provide a mechanism by which posting any change to any table invokes a follow-up call to commit the transaction, without further action by the user. This mechanism is usually called <i>AutoCommit</i>, or some similar term. It is not a Firebird mechanism—Firebird never commits transactions started by clients.</p>                                                                                                                                                                                                  |
| <b>Autonomous transaction</b>          | <p>A transaction that can be started inside a trigger or stored procedure to execute some operation separately from the “main” transaction in which the module itself is running. Autonomous transactions are supported in Firebird PSQL from v.2.5 onward.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Backup/restore (Firebird-style)</b> | <p><i>Backup</i> is an external process initiated by a user—usually SYSDBA—to decompose a database into a collection of compressed disk structures comprising metadata and data which are separated for storage. <i>Restore</i> is another external process—also user-initiated—which completely reconstructs the original database from these stored elements. The backup process also performs a number of housecleaning tasks on a database whilst reading it for backing up; and a restored database will be completely free of “garbage”. See also <i>gbak</i> and <i>nBackup</i>.</p>                                                                                                                                                                                                                                             |
| <b>BDE</b>                             | <p><i>Borland Database Engine</i>. Originally designed as the database engine of Paradox and, later, dBase, it was extended to provide a generic middleware connectivity layer between a variety of relational database engines and Borland application development tools for the Microsoft DOS and Windows platforms. The vendor-specific rules applicable to each RDBMS are encapsulated in a set of driver libraries known as SQL Links. The SQL Links drivers are version-specific.</p> <p>By 2000, when Borland released the codebase from which Firebird 1.0 was developed, it had already deprecated the BDE in favour of lighter, more modern driver technologies. The last known version of the BDE (v.5.2) shipped with Borland Delphi 6 in 2000. An InterBase driver in this distribution only partly supports Firebird.</p> |
| <b>Binary tree</b>                     | <p>A logical tree structure in which a node can subtend a maximum of two branches. Firebird indexes are formed in binary tree structures.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>BLOB</b>                            | <p>Mnemonic for <i>Binary Large Object</i>. This is a data item of unlimited size, in any format, which is streamed into the database byte-by-byte and stored without any format modification. Firebird allows BLOBs of different types to be classed by means of sub-types. Firebird’s ancestor, InterBase, was the first relational database to support storage of BLOBs. See also <i>CLOB</i>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>BLOB Control Structure</b>          | <p>A C language structure, declared in an external function module as a <i>typedef</i>, through which a blob function call accesses a blob. An external BLOB function cannot refer to actual BLOB data: it refers to a pointer to a blob control structure instead.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>BLOB Filter</b>                     | <p>A BLOB Filter is a specialized external function (UDF) that transforms BLOB data from one subtype to another. Firebird includes a set of internal BLOB filters which it uses in the process of storing and retrieving metadata. One internal filter converts text data transparently between SUB_TYPE 0 (none) and SUB_TYPE 1 (text, sometimes referred to as ‘Memo’). Developers can write BLOB filters to transform the contents of BLOBs. They are declared to individual databases, as other kinds of external functions are.</p>                                                                                                                                                                                                                                                                                                |

|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>BLR</b>                             | <i>Binary Language Representation</i> , an internal relational language with a binary notation that is a super set of the “human-readable” languages that can be used with Firebird, viz.. SQL and GDML. Firebird’s DSQL interface to the server translates queries into BLR. The BLR versions of compiled triggers and stored procedures, check constraints, defaults and views are stored in BLOBs. Some client tools, for example IB_SQL and the command-line tool isql, have facilities to inspect this BLR code. In <i>isql</i> , execute the command SET BLOB ALL, perform SELECT statements to get the appropriate BLR fields from the system tables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Buffer</b>                          | A buffer is a block of memory for caching copies of pages read from the database. The term buffer is synonymous with cache page. See also <i>Page</i> and <i>Cache</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>‘BURP’</b>                          | Internal name for the <i>gbak</i> code—a mnemonic for “Backup [and] Restore Program”.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Cache</b>                           | When a page is read from disk it is copied into a block of memory known as the database cache or, simply, the cache. The cache consists of buffers, each the size of a database page, determined by the <i>page_size</i> parameter declared when the database is created.<br><br>The word buffer in this context means a block of memory exactly the same size as one database page. The cache size is configurable, as a number of pages (or buffers). Hence, to calculate the size of the cache, multiply the <i>page_size</i> by the number of cache pages ( <i>buffers</i> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Cardinality (of a set)</b>          | The number of rows in a physical or specified set. The cardinality of a row refers to its position in the top-to-bottom order of the set.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Case-insensitive index</b>          | This is an index using a collation sequence in which lower-case characters are treated as though they were the same as their upper-case equivalents. Firebird 1.0 does not support case-insensitive indexes. Case-insensitive collation sequences have been appearing regularly from Firebird 1.5 forward.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Cascading integrity constraints</b> | Firebird provides the optional capability to prescribe specific behaviors and restrictions in response to requests to update or delete rows in tables which are pointed to by the REFERENCES sub-clause of a FOREIGN KEY constraint. The CASCADE keyword causes changes performed on the “parent” row to flow on to rows in tables having the FOREIGN KEY dependencies. ON DELETE CASCADE, for example, will cause dependent rows to be deleted when the parent is deleted.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Casting</b>                         | Casting is a mechanism for converting output or variable values of one data type into another data type by means of an expression. Firebird SQL provides the CAST() function for use in both dynamic SQL (DSQL) and procedural SQL (PSQL) expressions. From the “2” series forward, casting to the type of a <i>domain</i> is possible.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Character set</b>                   | Two super-sets of printable character images and control sequences are in general use today in software environments—ASCII/ANSI and UNICODE. ASCII characters, represented by one byte, have 256 possible variants whilst UNICODE, of two, three or four bytes, can accommodate tens of thousands of possibilities. Because databases need to avoid the prohibitive overhead of making available every possible printable and control character used for programming anywhere in the world, the super-sets are divided up into code pages, also known as code tables. Each code page defines a subset of required characters for a specific language, or family of languages, mapping each character image to a number. The images and control sequences within each code page are referred to collectively as a character set. A character image might be mapped to different numbers in different characters sets.<br><br>Firebird supports a default character set for a database and definition of an explicit character set for any character, varchar or BLOB SUB_TYPE 1 (text BLOB) column. If no character set is defined for a database, its character set defaults to NONE, causing all character data to be stored exactly as presented, with no attempt to convert characters (transliterate) to any particular character set. |

|                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Classic architecture</b>          | The term “Classic architecture” refers to the original InterBase model where a separate server process is started for each client connection. It predated the “Superserver” model which threads all client processes from a single server process. SuperClassic, a variant of Classic that superintends threaded Classic instances, was developed to take advantage of multiple processors and larger RAM configurations on 64-bit hosts.                                                                                                                                                    |
| <b>CLOB</b>                          | Mnemonic for ‘Character Large OBject’. This term has crept into use recently, as other RDBMSs have mimicked Firebird’s support for storing large objects in databases and have made up their own names for them. A CLOB is equivalent to Firebird’s BLOB SUB_TYPE 1 (TEXT). See also BLOB.                                                                                                                                                                                                                                                                                                   |
| <b>Coercing data types</b>           | In the Firebird API’s XSQLDA structures, converting a data item of one SQL type to another, compatible SQL type is known as data type coercion.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Collation order</b>               | Defines how a sort operation orders character columns in output sets, the pairing of lower case and upper case characters for the UPPER() function and how characters in character columns are compared in searches. A collation order applies to a specific character set. If multiple collation orders are available for a particular character set, one collation order will be treated as the default. By convention, the default collation order has the same name as the character set and is binary, ordering items only by their numerical codes without recourse to specific rules. |
| <b>Column</b>                        | In SQL databases, data are stored in structures which can be retrieved as tables or, more correctly, sets. A set consists of one or more rows each identical in the horizontal arrangement of distinctly defined items of data. One distinct item of data considered vertically for the length of the set is known as a column. Application developers often refer to columns as fields.                                                                                                                                                                                                     |
| <b>Commit</b>                        | When applications post changes affecting rows in database tables, new versions of those rows are created in temporary storage blocks. Although the work is visible to the transaction in which it occurred, it cannot be seen by other users of the database. The client program must instruct the server to commit the work in order for it to be made permanent. If a transaction is not committed, it must be rolled back in order to undo the work.                                                                                                                                      |
| <b>CommitRetaining</b>               | Also known as “soft Commit”, this is a transaction setting that implements the COMMIT WITH RETAIN attribute of a transaction. With this attribute, a transaction’s context is kept active on the server until the client application finally calls Commit (a “hard Commit”) and allows the transaction inventory management processes to pass it through for garbage collection. The widespread use CommitRetaining in applications is a common cause of performance degradation. See also Oldest Interesting Transaction.                                                                   |
| <b>Common table expression (CTE)</b> | A non-persistent e set object that is set up by a client application at run-time with a “preamble” to define a complex set to interact with one or more other sets within a single DSQL statement. A CTE is highly optimized and usually performs faster than a stored procedure that could do the equivalent processing. A CTE can be self-referencing, providing an efficient way to perform a recursive operation on a set.                                                                                                                                                               |
| <b>Concurrency</b>                   | <p>The term <i>concurrency</i> broadly refers to multiple users accessing the same data simultaneously. It is also widely used in documentation and support lists to refer to the particular set of attributes that apply to a transaction— isolation level, locking policy and others. For example, someone may ask you “What are your concurrency settings?”</p> <p>Even more specifically, the word <i>concurrency</i> is often used as a synonym for the <i>SNAPSHOT isolation level</i>.</p>                                                                                            |
| <b>Constraint</b>                    | Firebird makes many provisions for defining formal rules which are to be applied to data. Such formal rules are known as constraints. For example, a PRIMARY KEY is a constraint which marks a column or group of columns as a database-wide pointer to all of the other columns in the row defined by it. A CHECK constraint sets one or more rules to limit the values which a column can accept.                                                                                                                                                                                          |
| <b>Contention</b>                    | When two transactions attempt to update the same row of a table simultaneously, they are said to be in contention—they are contending (or competing).                                                                                                                                                                                                                                                                                                                                                                                                                                        |



|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Correlated Subquery</b> | A query specification can define output columns that are derived from expressions. A subquery is a special kind of expression that returns a single value which is itself the output of a SELECT statement. In a correlated sub-query, the WHERE clause contains one or more search keys that are relationally linked to (and matched to) columns in the main query.                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Crash</b>               | A slang term for abnormal termination of the server or a client application.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Crash recovery</b>      | Processes or procedures that are implemented to restore the server and/or client applications into running condition following an abnormal termination of either the server or the application, or both.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>CTE</b>                 | See <i>Common table expression</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>CVS</b>                 | Acronym for Concurrent Versions System, an open-source program that allows developers to keep track of different development versions of source code. CVS is widely used for open source projects, including some sub-projects of Firebird. The Firebird core code's version control system is <i>Subversion</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Cyclic links</b>        | In a database context, this is a term for an inter-table dependency where a foreign key in one table (TableA) refers to a unique key in another table (TableB) which contains a foreign key which points back, either directly, or through a reference to another table, to a unique key in TableA.                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Database</b>            | In its broadest sense, the term <i>database</i> applies to any persistent file structure which stores data in some format which permits it to be retrieved and manipulated by applications.<br><br>Firebird is not a database. It is a system that manages databases.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Database trigger</b>    | A PSQL module that is defined to run automatically when a client session opens or closes or when a transaction starts, commits or rolls back. First supported in Firebird "2" series.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>DB_KEY</b>              | See <i>RDB\$DB_KEY</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>DDL</b>                 | Mnemonic for <i>Data Definition Language</i> —the subset of SQL that is used for defining and managing the structure of data objects. Any SQL statement starting with the keywords CREATE, ALTER, RECREATE, CREATE OR REPLACE, or DROP is a DDL statement. In Firebird, some DDL statements start with the keyword DECLARE, although not all DECLARE statements are DDL.                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Deadlock</b>            | When two transactions are in contention to update the same version of a row, the transactions are said to be <i>in deadlock</i> when one transaction (T1), having a lock on row A, requests to update row B, which is locked by another transaction (T2) and T2 wants to update row A. Normally, genuine deadlocks are rare, because the server can detect most deadlocks and resolves them itself without raising a deadlock exception. The Firebird server unfortunately generalizes all lock conflict exceptions into a single error code that reports a "deadlock", regardless of the actual source of the conflict. Client application code must resolve a lock conflict by having one transaction roll back in order to allow the other to commit its work. |
| <b>Degree (of a set)</b>   | The number of columns in a tabular set. The degree of a column refers to its position in the left-to-right sequence of columns, starting at 1.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Deployment</b>          | The process of distributing and installing software components for production use.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Derived table</b>       | A non-persistent set object that is defined in a SELECT expression by the client application at run-time, to provide a set intended to interact with other sets inside a single statement.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Dialect</b>             | A term that distinguishes Firebird's native SQL language set from an older language set that was implemented in Firebird's predecessor, InterBase® 5. The older language set remains available to Firebird for near-compatibility with legacy databases, as "dialect 1". The native Firebird set is "dialect 3".                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>DML</b>                 | Mnemonic for <i>Data Manipulation Language</i> , the major subset of SQL statements, which perform operations on sets of data.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Domain</b>            | A Firebird SQL feature whereby you can assign an identifying name to a set of data characteristics and constraints (CREATE DOMAIN) and then use this name in lieu of a data type when defining columns in tables and, under some conditions, in CAST() expressions and when defining arguments and variable in PSQL modules (stored procedures, triggers and executable blocks).                                                                                                                                                                                             |
| <b>DPB</b>               | Mnemonic for <i>Database Parameter Buffer</i> , a character array defined by Firebird's application programming interface (API). It used by applications to convey the parameters that specify the characteristics of a client connection request, along with their specific item values, across the API.                                                                                                                                                                                                                                                                    |
| <b>DSQL</b>              | <i>Dynamic SQL</i> — refers to statements that an application submits in run-time, with or without parameters, as contrasted with “static SQL” statements which are coded directly into special code blocks of a host language program and are subsequently preprocessed for compilation within older-style embedded SQL applications. Applications which use Firebird API calls, either “raw” or through a class library which encapsulates the Firebird API, are using DSQL.<br><br>DSQL can also be executed inside a PSQL module by way of the EXECUTE STATEMENT syntax. |
| <b>DTP</b>               | Desktop publishing, the activity of using computer methods to prepare documents for publication in print or on the Web.                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>‘DUDLEY’</b>          | Internal name for the source-code for the deprecated metadata utility, gdef. The name derives from the mnemonic “DDL” ( <i>Database Definition Language</i> ).                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>dyn, also DYN</b>     | A byte-encoded language for describing data definition statements. Firebird's DSQL subsystem parses data definition language (DDL) statements and passes them to a component that outputs DYN for interpretation by the Y-Valve, another subsystem which is responsible for updating the system tables.                                                                                                                                                                                                                                                                      |
| <b>Error</b>             | A condition in which a requested SQL operation cannot be performed because something is wrong with the data supplied to a statement or procedure or with the syntax of the statement itself. When Firebird encounters an error, it does not proceed with the request but returns an <i>exception message</i> to the client application. See also <i>Exception</i> .                                                                                                                                                                                                          |
| <b>Error Code</b>        | An integer constant returned to the client or to a calling procedure when Firebird encounters an error condition. See also <i>Error</i> , <i>Exception</i> .                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>ESQL</b>              | Mnemonic for <i>Embedded SQL</i> , the mostly outmoded subset of SQL provided for embedding static SQL in special blocks within a host application, usually written in C.                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Event</b>             | An event implements Firebird's capability to pass a notification to a “listening” client application if requested to do so by a POST_EVENT call in a trigger or stored procedure.                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Executable block</b>  | A block of executable code, written in PSQL, that is submitted from a client application at run-time, using the syntax implemented as EXECUTE BLOCK. Supported in Firebird versions from the “2” series onward.                                                                                                                                                                                                                                                                                                                                                              |
| <b>Executable string</b> | A string variable composed as a DSQL statement, that is passed to EXECUTE STATEMENT inside a PSQL module, for execution outside the module. From v.2.5, syntax is available to use replaceable parameters in the string and to execute the statement in a separate transaction, in the same database or another database..                                                                                                                                                                                                                                                   |
| <b>Exception</b>         | An exception is the Firebird server's response to an error condition that occurs during a database operation. Several hundred exception conditions are realized as error codes, in a variety of categories, which are passed back to the client in the <i>Error Status Vector (Array)</i> . Exceptions are also available to stored procedures and triggers, where they can be handled by a custom routine.<br><br>Firebird supports user-defined exceptions as well.                                                                                                        |

|                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>External function</b>           | Until v.2.1, Firebird has but a few built-in, SQL-standard functions. To extend the range of functions available for use in expressions, the Firebird engine can access custom functions, written in a host language such as C/C++ or Delphi, as if they were built-in. Several free libraries of ready-made external functions—also known as “UDFs” (“user-defined functions”) exist among the Firebird community, including two that are included with the Firebird binary release distributions. New UDFs can be written easily in languages that support the C calling conventions.                                                                                                                                                                                                     |
| <b>Executable stored procedure</b> | Stored procedure that is called using EXECUTE PROCEDURE and does not return a multi-row result set. <i>c.f. Selectable stored procedure.</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Execute</b>                     | <p>In a client application, <i>execute</i> is commonly used as a verb which means “perform my request” when a data manipulation statement or stored procedure call has been prepared by the client application.</p> <p>In DSQL and PSQL, the phrase EXECUTE PROCEDURE is used with a stored procedure identifier and its input arguments, to invoke executable stored procedures. PSQL also supports EXECUTE STATEMENT, whereby a DSQL statement is taken as a string argument and executed from within the PSQL module.</p>                                                                                                                                                                                                                                                                |
| <b>fdb or FDB</b>                  | By convention, the suffix used for a Firebird primary database file. It is no more than a convention: Firebird works with any other file suffix, or none at all.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>FIBPlus</b>                     | Trade name of the extended, commercial version of the older <i>FreeIBComponents</i> of Greg Deatz, data access components encapsulating the Firebird and InterBase APIs, for use with the Embarcadero RAD development products.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Foreign Key</b>                 | <p>A foreign key is a formal constraint on a column or group of columns in one table that links that table to another table's corresponding primary or unique key. If the foreign key is non-unique and the table itself has a primary key, the table is capable of being the “many” side of a 1:Many relationship.</p> <p>Firebird supports the declaration of a formal foreign key constraint which will enforce referential integrity automatically. When such a constraint is declared, Firebird automatically creates a non-unique index on the column or columns to which the constraint applies and also records the dependencies between the tables that are linked by the constraint.</p>                                                                                          |
| <b>Garbage collection</b>          | General term for the cleanup process that goes in the database during normal use to remove obsolete back-versions of rows that have been updated. In Superserver, garbage collection runs as a background thread to the main server process. Garbage collection can also be performed by sweeping and by backing up the database.                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>gbak</b>                        | <p>The command-line utility (found in the /bin directory of your Firebird installation) that backs up and restores databases. It is not a file-copy program: its backup operation decomposes the metadata and data and stores them separately, in a compressed binary format, in a filesystem file. By convention, backup files are often given the filename suffix “gbk” or “fbk”. Restoring decompresses this file and reconstructs the database as a new database file, before feeding the data into the database objects and rebuilding the indexes.</p> <p>Apart from normal data security tasks expected of a backup utility, gbak performs important roles in the regular maintenance of “database hygiene” and in the recovery of corrupted databases. See also <i>nBackup</i>.</p> |

|                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gdb or GDB</b>                   | By convention, this is the file extension traditionally used for InterBase databases. However, a Firebird database file can have any extension, or none at all. Many Firebird developers use “.fdb” instead, to distinguish their Firebird databases from their InterBase databases, or as part of a solution to a performance-crippling “safety feature” introduced by Microsoft Corporation into their XP and Server 2003 operating systems, targeted at files having the “.gdb” extension.<br><br>Why “GDB”? It is a mnemonic artifact of the name of the company that created the original InterBase, Groton Database Systems. |
| <b>GDML</b>                         | Mnemonic for Groton Data Manipulation Language, a high-level relational language similar to SQL. GDML was the original data manipulation language of InterBase, functionally equivalent to DML in Firebird SQL but with some data definition capabilities. It is still supported through the interactive query utility, <i>qli</i> .                                                                                                                                                                                                                                                                                               |
| <b>gdef</b>                         | <i>gdef</i> was an older InterBase utility for creating and manipulating metadata. Since <i>isql</i> and the dynamic SQL interface can handle DDL, <i>gdef</i> is virtually redundant now. However, because it can output DYN language statements for several host programming languages including C and C++, Pascal, COBOL, ANSI COBOL, Fortran, BASIC, PLI and ADA, it still has its uses in the development of embedded SQL applications.                                                                                                                                                                                       |
| <b>Generator</b>                    | A number-generating engine for producing unique numbers in series. The statement <code>CREATE GENERATOR generator_name</code> seeds a distinct series of signed 64-bit integer numbers. <code>SET GENERATOR TO n</code> sets the first number in the series. The function <code>GEN_ID (generator_name, m)</code> causes a new number to be generated, which is <i>m</i> higher than the last generated number. See also <i>Sequence</i> .                                                                                                                                                                                         |
| <b>gfix</b>                         | <i>gfix</i> is a collection of command-line utilities, including database repair tools of limited capability. It includes tools to activate database shadows, to put the database into single-user (exclusive access) mode (shutdown) and to restore it to multi-user access (restart). It can resolve limbo transactions left behind by multiple-database transactions, set the database cache, enable or disable forced (synchronous) writes to disk, perform sweeping and set the sweep interval, switch a Firebird database from read/write to read-only or vice versa and set the database dialect.                           |
| <b>Global temporary table (GTT)</b> | A table definition that can be set up, much like a regular table, and available for any client or transaction to instantiate and populate. Each instantiation has a limited scope, either transaction “life” or the life of the client session. The instantiation is stored in a private file not visible to other connections or (if transaction-scoped) other transactions.                                                                                                                                                                                                                                                      |
| <b>gpre</b>                         | In ESQL application development (“embedded applications” that do not use the API), <i>gpre</i> is the preprocessor for static SQL language blocks in host language source code, translating them to BLR format in preparation for compiling. It can preprocess C, C++, COBOL, Pascal and ADA host code, on selected platforms.                                                                                                                                                                                                                                                                                                     |
| <b>Grant/Revoke</b>                 | SQL commands <code>GRANT</code> and <code>REVOKE</code> , which are used for setting up user privileges for accessing the objects in a database.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Groton</b>                       | <i>Groton Data Systems</i> , the name of the original company that designed and developed the relational database system that was named “InterBase”. From InterBase, eventually, evolved Firebird. Two of the original Groton directors, Jim Starkey and Ann Harrison, were actively involved in aspects of Firebird development in its early years. Ann is still a regular contributor to the Firebird peer support forums.                                                                                                                                                                                                       |
| <b>gsec</b>                         | <i>gsec</i> is Firebird’s command-line security utility for managing the server-level user/password database (security2.fdb for v.2.0+, security.fdb for v.1.5, isc4.gdb for v.1.0), which applies to all users and all databases. This utility cannot be used to create or modify roles, since roles exist within a database.                                                                                                                                                                                                                                                                                                     |

|                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gstat</b>                 | <i>gstat</i> is the command-line utility with which to gather statistics about a Firebird database. It analyzes the internal layout, like the fill factor, header page, index pages, log page and system relations. Table-by-table information about record versions (usually lengthy) can be optionally obtained, using the <i>-r</i> and <i>-t</i> tablename switches together.                                                                                                                                                                                                                                                                   |
| <b>GTT</b>                   | See <i>Global temporary table</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Hierarchical database</b> | An old design concept for implementing table-to-table relationships in databases by building up tree structures of inherited indexes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Host-language</b>         | General term referring to the language in which application code is written.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Identity attribute</b>    | Some RDBM systems—MSSQL for example—support a table attribute which automatically implements a surrogate primary key integer column for a table and causes a new value to be generated automatically for each new row inserted. Firebird does not support this attribute directly. A similar mechanism can be achieved by explicitly defining an integer field of adequate size, creating a generator to populate it and defining a Before Insert trigger that calls the <code>GEN_ID()</code> function to get the next value of the generator.                                                                                                     |
| <b>IBO</b>                   | Mnemonic for IB Objects, data access components and data-aware controls encapsulating the Firebird and InterBase APIs, for use with the Embarcadero RAD products on Windows and, latterly, Lazarus and FreePascal for cross-platform development.                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>IBX</b>                   | Mnemonic for <i>InterBaseXpress</i> , data access components encapsulating the InterBase API, distributed previously by Borland for interfacing InterBase with the Delphi and C++ Builder products of which they were once proprietors.                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Index</b>                 | A specialized data structure, maintained by the Firebird engine, providing a compact system of data pointers to the rows of a table.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>INET error</b>            | In the <code>firebird.log</code> , marks an error received by the Firebird network layer from a TCP/IP client/server connection.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Installation</b>          | Refers to the procedure and process of copying the software to a computer and configuring it for use. Often also used to refer to the file system layout of the Firebird components on the server.                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>InterBase</b>             | <p>A relational database management system that was the ancestor of Firebird. Developed originally by a company named Groton Data Systems, it passed into the ownership of the Borland Software company early in the 1990s. InterBase version 6 was released into open source in 2000 under the InterBase Public License. Firebird was developed by independent developers from this open source code base and immediately became a forked development when the IB6 source code tree was made read-only.</p> <p>Closed, proprietary versions of InterBase are still developed and marketed by the Embarcadero company that acquired it in 2008.</p> |
| <b>InterClient</b>           | Obsolete JDBC Type 2 Java client for the original InterBase 6 open source server. In Firebird, it is superseded by Jaybird, a family of open source JDBC/JCA compliant Java drivers (Type 4 and Type 2).                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>InterServer</b>           | Obsolete, Java-driven server-based communication layer originally distributed with the InterBase 6 open source code release. Neither InterServer nor its companion InterClient have ever been developed by the Firebird Project, having been superseded by a more modern and open Java interface, named Jaybird.                                                                                                                                                                                                                                                                                                                                    |
| <b>ISC, isc, etc.</b>        | Error messages, some environment variables and many identifiers in the Firebird API have the prefix “ISC” or “isc”. As a matter of purely historical interest, these initials are derived from the initial letters of “InterBase Software Corporation”, the name of a subsidiary company of Borland that existed during some periods of Borland’s life as a software producer.                                                                                                                                                                                                                                                                      |

|                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Isolation level</b>          | <p>This attribute of a transaction prescribes the way one transaction shall interact with other transactions accessing the same database, in terms of visibility and locking behavior. Firebird supports three levels of isolation: Read Committed, Repeatable Read (also known as “Snapshot” or “Concurrency”) and Snapshot Table Stability (also known as “Consistency”). Although Read Committed is the default for most relational engines, Firebird’s default level is Repeatable Read, thanks to optimistic, row-level locking.</p> <p>See also <i>Transaction Isolation</i>.</p>  |
| <b>isql</b>                     | <p>Name of Firebird’s console-mode interactive query utility, which can connect to one database at a time. It has a powerful set of commands, including its own subset of SQL-like commands, supplementary to the regular dynamic SQL command set. It has a large set of embedded macros for obtaining metadata information. isql can output batches of commands, including embedded comments, to a file; and it can “run” such a file as a script—the recommended way to create and alter database objects.</p>                                                                         |
| <b>JDBC</b>                     | <p>Java Database Connectivity, a set of standards for constructing drivers for connecting Java applications to SQL databases.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Join</b>                     | <p>JOIN is the SQL keyword for specifying to the engine that the output of the SELECT statement involves merging columns from multiple tables, linked by matching one or more pairs of keys.</p>                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>jrd</b>                      | <p>The internal name of the Firebird database engine kernel. It is a mnemonic for <i>Jim’s Relational Database</i>, an artifact of the original engine, invented by Jim Starkey, which became InterBase and, later, Firebird.</p>                                                                                                                                                                                                                                                                                                                                                        |
| <b>Key</b>                      | <p>A key is a constraint on a table which is applied to a column or group of columns in the table’s row structure. A <i>primary key</i> or a <i>unique key</i> points to the unique row in which it exists; while a <i>foreign key</i> points to a unique row in another table by linking to its primary key or another unique key.</p>                                                                                                                                                                                                                                                  |
| <b>Kill (shadows)</b>           | <p>When a database shadow is created using the MANUAL keyword, and the shadow becomes unavailable, further attachments to the database are blocked. In order to re-enable attachments to the database, it is necessary to issue a –kill database command from the <i>gfix</i> utility to delete references to the shadow.</p>                                                                                                                                                                                                                                                            |
| <b>Leaf bucket</b>              | <p>In a binary index tree, the data item in the last index on a node of the tree is known as a leaf bucket. The leaf buckets figure reported in <i>gstat</i> index statistics provides an approximate count of the number of rows in the table.</p>                                                                                                                                                                                                                                                                                                                                      |
| <b>Limbo (transaction)</b>      | <p>A <i>limbo transaction</i> can occur where a transaction spans more than one database. Multi-database transactions are protected by <i>two-phase commit</i> which guarantees that, unless the portions of the transaction residing in each database get committed, all portions will be rolled back. If one or more of the databases involved in the transaction becomes unavailable before the completion of the two-phase commit, the transaction remains unresolved and is said to be <i>in limbo</i>.</p>                                                                         |
| <b>Locking conflict</b>         | <p>Under Firebird’s optimistic locking scheme, a row becomes locked against update from other transactions as soon as its transaction posts an update request for it. Where the isolation level of the transaction is SNAPSHOT TABLE STABILITY (also known as <i>consistency</i>), the lock occurs as soon as the transaction reads the row. A locking conflict occurs when another transaction attempts to post its own update to that row. Locking conflicts have various causes, characteristics and resolutions according to the specific settings of the transactions involved.</p> |
| <b>Lock resolution (policy)</b> | <p>As a general term, refers to the measures taken in application code to resolve the conditions that occur when other transactions attempt to update a row that is locked by a transaction because of an update request. As a specific term, <i>lock resolution policy</i> refers to the WAIT/NO WAIT parameter setting of a transaction, that specifies how a transaction should react if a locking conflict arises.</p>                                                                                                                                                               |

|                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Metadata</b>                              | A generic quantity noun referring to the structure of all the objects that comprise a database. Because Firebird stores the definitions of its objects right inside the database, using its own native tables, data types and triggers, the term “metadata” also refers to the data stored in these system tables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Monitoring tables (MON\$ tables)</b>      | A suite of read-only tables, system-defined at database-create time, for use by clients on demand, for system and database monitoring.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Multi-generational architecture (MGA)</b> | Term used for the engine mechanism of Firebird, that enables <i>optimistic row locking</i> and a high level of <i>transaction isolation</i> that protects a transaction’s dynamic view of the database and its changes to rows in that view, without blocking other readers. It is achieved by the engine’s capability to store multiple versions of rows concurrently and to “age” these versions with respect to the original view. See also <i>Versioning architecture</i> .                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Natural (scan)</b>                        | Seen sometimes in the query plans created by the optimizer, indicating that the associated table will be scanned in “natural order”, i.e. in no particular order and without reference to an index.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>nBackup</b>                               | Multi-level incremental backup utility first available in the Firebird “2” series. Unlike <i>gbak</i> , it is “data-unaware”, backing up snapshot page images while storing active work in files known as <i>deltas</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Non-standard SQL</b>                      | A term often heard with reference to relational database management systems that have a low degree of conformance with the ISO language and syntax standards for SQL. It is often used to describe or refer to idiomatic SQL implementations in Firebird. See also <i>Standard SQL</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Non-unique key</b>                        | This is a column or group of columns that can act as a pointer to a grouping of rows in a set. A foreign key constraint to implement a 1:Many relationship is typically formed by matching a non-unique column or group in a “child” or “detail” set to a unique key in a “parent” or “master” set.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Normalization</b>                         | A technique commonly used during the analysis of data preceding database design to abstract repeating groups of data out of multiple tables and eliminate the potential to duplicate the same “facts” in related tables.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Null</b>                                  | <p>Sometimes wrongly referred to as a “null value”, null is the <i>state</i> of a data item which has no known value. Logically it is interpreted as unknown and is thus unable to be evaluated in expressions.</p> <p>Null is never equivalent to zero, blank, an empty (zero length) string and it does not represent either infinity or NaN. It represents the state of a data item which either has never been assigned a value or has been set NULL by an operation.</p> <p>“NULL is a state, not a value.” (Now, where is that T-shirt?)</p>                                                                                                                                                                                                                                                                                                |
| <b>ODBC</b>                                  | Mnemonic for Open Database Connectivity. It is a call-level interface standard that allows applications to access data in any database which has a driver supporting this standard. Several ODBC drivers are available that support Firebird, including an open source one developed within the Firebird Project that is internally consistent with the JDBC (Java Database Connectivity) standard.                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>ODS</b>                                   | <p>Mnemonic for On-Disk Structure. It is a number that refers to the version of the internal structure and layout of a Firebird or InterBase database. Firebird 1 creates ODS 10, v.1.5 creates ODS 10.1, v.2.0 creates ODS 11.0, v.2.1 creates ODS 11.1 and v.2.5 creates ODS 11.2. A Firebird server can work with a database of a lower ODS than it creates but not with one higher than it creates; nor can it create a database of a higher or lower ODS.</p> <p>A database can be raised to a higher ODS by backing it up with the <i>gbak</i> -b[ackup] -t[ransportable] option using the old version’s <i>gbak</i> program, and restoring from that <i>gbak</i> file using the new version’s <i>gbak</i>.. If migrating from v.2.0 or lower to v.2.1 or higher, be sure to check the <i>Migration Notes</i> (Ch. 5) before you start.</p> |

|                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>OLAP</b>                                 | Mnemonic for On-Line Analytical Processing, a technology that is applied to databases that have grown to such a volume that it is impracticable for them to be queried directly as the basis of business decisions. Typically, OLAP systems are designed to analyze and graph trends, identify and capture historical milestones or anomalies, manufacture projections and hypothetical scenarios, crunch large volumes of data for reports, and so on, in reasonable time.                                                                                                                                                                                                                                                                                                                 |
| <b>OS</b>                                   | Abbreviation for “operating system”.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Oldest active transaction (OAT)</b>      | A statistic maintained by the Firebird engine, global to a database, the oldest transaction still in the database that has not been either committed or rolled back.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Oldest interesting transaction (OIT)</b> | A statistic maintained by the Firebird engine, global to a database, it is the ID of the oldest transaction that has not been committed. It is sometimes the same transaction as OAT but it may not be, since any transaction that is still active, has been rolled back, committed using COMMIT WITH RETAIN (CommitRetaining) or left in limbo by a failed two-phase commit remains “interesting”. When the OIT gets “stuck” at a much lower transaction ID than the newest transactions, garbage collection (cleanup of old record versions) can not proceed past it (to higher-numbered transactions) and database operations slow down and, eventually hang completely. OIT can be retrieved as OIT can be retrieved using the -header switch of the <i>gstat</i> command-line utility. |
| <b>OLE-DB</b>                               | Mnemonic for <i>Object Linking and Embedding for Databases</i> . OLE is a Microsoft standard developed and promoted for incorporating binary objects of many diverse types (images, documents, etc.) into Windows applications, along with application-level linking to the software engines that create and modify them. OLE-DB was added in an attempt to provide developers with the means to supply a more vendor-specific support for database connectivity—especially relational databases—than can be achieved with “open” standards such as ODBC. More recently, Microsoft layered the ADO (Access Data Objects) technology on top of OLE-DB.                                                                                                                                       |
| <b>OLTP</b>                                 | Mnemonic for <i>On-line Transaction Processing</i> , recognized as one of the essential requirements for a database engine. Broadly speaking, OLTP refers to support for clients performing operations that read, alter or create data in real time.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Optimization</b>                         | Generally, optimization refers to techniques for making the database and application software perform as responsively as possible. As a specific term, it is often applied to the techniques used by the Firebird engine to analyze SELECT statements and construct efficient plans for retrieving data. The routines in the Firebird engine which calculate these plans are known collectively as <i>the Firebird optimizer</i> .                                                                                                                                                                                                                                                                                                                                                          |
| <b>Page</b>                                 | A Firebird database is made up of fixed-sized blocks of disk space called <i>pages</i> . The Firebird engine allocates pages as it needs to. Once it stores data, a page could be any one of 10 different <i>types</i> of pages, all of equal size—the size defined in the PAGE_SIZE attribute during database creation. The type of page the engine stores to disk depends on the type of data object being stored on the page—data, index, blob, etc.                                                                                                                                                                                                                                                                                                                                     |
| <b>Page_size</b>                            | The size of each fixed block on disk is determined by the page_size specified for the database when the database is created or restored. Chunks of cache memory are also allocated in page_size units.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Parameter</b>                            | <p>A widespread term in many Firebird contexts, it can refer to the values that are passed as arguments to and returned from stored procedures (input parameters, output parameters). The term can also refer to the data items that are passed to the function blocks of the Firebird API (database parameter block, transaction parameter block, service parameter block) or to the attributes, as seen from an application, of a connection instance (connection parameters) or a transaction (transaction parameters).</p> <p>In client applications, placeholder tokens that are accepted for passing into WHERE clauses of SQL statements, for substitution by constant values at run-time are often implemented as “parameters”—hence the term <i>parameterized queries</i>.</p>     |



|                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PHP</b>            | Short for <i>PHP: Hypertext Preprocessor</i> , an open source embedded HTML scripting language used to create web applications, especially those with database back-ends. It has good support for a number of network protocols. Its strength lies in its compatibility with many types of database engines. Also, PHP can talk across networks using IMAP, SNMP, NNTP, POP3, or HTTP. PHP's originator, in 1994, was Rasmus Lerdorf. Since 1997, it has been in the hands of a large open source community.                                                    |
| <b>Plan</b>           | See <i>Query Plan</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>Platform</b>       | Term loosely used to refer to the combination of hardware and operating system software, or operating system software alone. For example, “the Windows Server 2011 platform”, “the Linux platform”, “UNIX platforms”. “Cross-platform” usually means “applicable to multiple platforms” or “portable to other platforms”.                                                                                                                                                                                                                                       |
| <b>Prepare</b>        | An API function that is called before a query request is submitted for the first time, requesting validation of the statement, construction of a query plan and several items of information about the data expected by the server.                                                                                                                                                                                                                                                                                                                             |
| <b>Primary Key</b>    | A table-level constraint that marks one column or a group of columns as the key which must identify each row as unique within the table. Although a table may have more than one unique key, only one of those keys may be the primary key. When you apply the PRIMARY KEY constraint to columns in a Firebird table, uniqueness is enforced by the automatic creation of a unique index that is, by default, ascending, and named according to a convention.                                                                                                   |
| <b>PSQL</b>           | Mnemonic for <i>Procedural SQL</i> , the subset of SQL extensions implemented for writing stored procedures and triggers. There are minor differences between the subsets of PSQL allowed for stored procedures and for triggers.                                                                                                                                                                                                                                                                                                                               |
| <b>qli</b>            | Query Language Interpreter, an interactive query client tool for Firebird. It can process DDL and DML statements in both SQL and GDML, the pre-SQL query language of Firebird's ancestor, InterBase 3. Although largely succeeded by <i>isql</i> and a number of third-party GUI tools, <i>qli</i> has some value for its capability to realize some engine features not so far implemented in Firebird SQL. Unlike its successor, <i>isql</i> , <i>qli</i> can connect to more than one database at the same time and can simulate multi-database joins.       |
| <b>Query</b>          | A general term for any SQL request made to the server by a client application.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Query Plan</b>     | A strategy for the use of indexes and access methods for sorting and searching in queries. The Firebird optimizer always computes a plan for every SELECT query, including subqueries. It is possible to specify a custom plan using the PLAN clause syntax.                                                                                                                                                                                                                                                                                                    |
| <b>RDB\$—</b>         | Prefix seen in the identifiers of many system-created objects in Firebird, a relic from “Relational DataBase”, abbreviated as “RDB”, the name of an early relational database developed by DEC. The design of RDB was the precursor to Firebird's ancestor, InterBase.                                                                                                                                                                                                                                                                                          |
| <b>RDB\$DB_KEY</b>    | The hidden, volatile, unique key that is calculated internally for every row in a table, from the physical address of the page on which the row starts and its offset from the beginning of the page. It is directly related to the cardinality of tables and sets and can change without trace or warning. It will always change when the database is restored from a backup. RDB\$DB_KEY should never be treated as persistent. With care, it can be used dependably within an atomic operation to speed up certain operations in DSQL and PSQL dramatically. |
| <b>RDBMS</b>          | <i>Relational Database Management System</i> . Generically, it is a concept for storing data according to an abstract model that uses matching keys to link a formal group of data to another formal group of data, thereby representing a relationship between the two groups.                                                                                                                                                                                                                                                                                 |
| <b>Read Committed</b> | The least restrictive <i>isolation level</i> for any Firebird transaction, Read Committed permits the transaction to update its view of the database to reflect work committed by other transactions since the transaction began. The isolation levels SNAPSHOT and SNAPSHOT TABLE STABILITY do not permit the original view to change.                                                                                                                                                                                                                         |

|                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Read Uncommitted</b>      | A theoretical isolation level in which one transaction can see the uncommitted work of another: also known as “Dirty Read”. Firebird does not support this isolation level at all.                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Redundancy</b>            | A condition in a database where the same “fact” is stored in two unrelated places. Ideally, redundancy should be avoided by attention to normalization during analysis. However, there are circumstances where a certain amount of redundancy is justified. For example, accounting transactions often contain data items which arguably could be derived by joining, looking up or calculation from other structures. However, a legal requirement to store a permanent record that will not be altered if database relationships subsequently change would overrule the case for eliminating redundancy. |
| <b>Redundant Indexes</b>     | Redundant indexes often arise when existing databases are imported to Firebird from another RDBMS. When a PRIMARY KEY, UNIQUE or FOREIGN KEY constraint is applied to a column or columns, Firebird automatically creates an index to enforce the constraint. In so doing, Firebird ignores any existing indexes that duplicate its automatic index. Having duplicate indexes on keys or any other columns can defeat the query optimizer, causing it to create plans that are inherently slow.                                                                                                            |
| <b>Referential integrity</b> | <p>Generally, referential integrity is a term that refers to the way an RDBMS implements mechanisms that formally support and protect the dependencies between tables. Referential integrity support refers to the language and syntax elements available to provide these capabilities.</p> <p>Firebird provides formal mechanisms for supporting referential integrity, including cascading constraints for foreign key relationships. It is sometimes referred to as <i>declarative</i> referential integrity.</p>                                                                                      |
| <b>Regular expression</b>    | In the simplest terms, a regular expression (also known as a <i>regex</i> or a <i>regexp</i> ) is a pattern that is matched against an operand string expression from left to right. A regex is composed of alphanumeric characters and various wildcards and can be very complex and exacting about the string patterns it seeks to match. In Firebird, a regex is used with the SIMILAR TO operator.                                                                                                                                                                                                     |
| <b>Relation</b>              | <p>In relational database theory, a self-contained body of data formally arranged in columns and rows. The term is almost interchangeable with ‘table’—except that a relation cannot have duplicate rows, whereas a table can. In Firebird, the terminology persists in the names of some system tables, e.g. RDB\$RELATIONS, which contains an entry for each table in the database.</p> <p>A view is considered by some documentation writers to be a relation. The author disagrees, since view cannot participate in relationships.</p>                                                                |
| <b>Relationship</b>          | An abstract term referring to the way relations (or tables) are linked to one another through matching keys. For example, an Order Detail table is said to be in a dependency relationship or a Foreign Key relationship with an Order Header table.                                                                                                                                                                                                                                                                                                                                                       |
| <b>Replication</b>           | Replication is a systematic process whereby data are copied from one database to another on a regular basis, according to predictable rules, in order to bring two or more databases into a synchronized state.                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Result table</b>          | A result table is the set of rows output from a SQL SELECT query. More correctly, <i>result set</i> . Synonymous with <i>output set</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Role</b>                  | A standard SQL mechanism for defining a set of permissions to use objects in a database. Once a role is created, permissions are assigned to it, using GRANT statements, as if it were a user. The role is then GRANTED to individual users as if it were itself a privilege, thus simplifying the maintenance of user permissions in a database.                                                                                                                                                                                                                                                          |
| <b>Rollback</b>              | In general, rollback is the act or process of undoing all of the work that has been posted during the course of a transaction. As long as a transaction has work pending that is posted but not committed, it remains unresolved and its effects are invisible to other transactions. If the client application calls for a ROLLBACK, all of the posted work is cancelled and the changes are lost. Once a transaction is committed, its work cannot be rolled back.                                                                                                                                       |

|                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Schema</b>                      | A term for the formal description of a database, usually realized as a script, or set of scripts, containing the SQL statements defining each and every object in the database. The term schema is often interchangeable with the term metadata.                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Schema cache</b>                | A mechanism whereby some descriptive elements of a database are stored on a client's local disk or in its local memory for quick reference at run-time, to avoid the need for constantly requering the database to obtain schema (metadata) attributes.                                                                                                                                                                                                                                                                                                                                                               |
| <b>Scrollable cursor</b>           | A cursor is a pointer to a row in a database table or output set. A cursor's position in the database is determined from the cardinality of the row to which it is currently pointing, i.e. its offset from the first row in the set. Repositioning the cursor requires returning the pointer to the first row in order to find the new position. A scrollable cursor is one which is capable of locating itself at a specified new position (upward or downward) relative to its current position. It is not supported in Firebird. Several connectivity interfaces simulate a scrollable cursor at the client side. |
| <b>Selectivity of an index</b>     | As a general term, refers to the spread of possible values for the index column throughout the table. The fewer the possible values, the lower the selectivity. Low selectivity can also occur where an index with a higher number of possible values is represented in actual data by a very high rate of duplication of a few values. Low selectivity is bad, high is good. . A unique index has the highest possible selectivity.                                                                                                                                                                                  |
| <b>Selectable stored procedure</b> | Stored procedure that is written using special PSQL syntax to output a multi-row result set to the caller. It is called using a SELECT statement. cf <i>Executable stored procedure</i> .                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Sequence</b>                    | A standards-compliant number-generating engine for producing unique numbers in series. The statement CREATE SEQUENCE <i>sequence_name</i> seeds a distinct series of signed 64-bit integer numbers. The NEXT VALUE FOR <i>sequence_name</i> function causes a new number to be generated, which is 1 higher than the last generated number. See also <i>Generator</i> .                                                                                                                                                                                                                                               |
| <b>Services API</b>                | An application programming interface to the functions accessed by several of the Firebird command-line server utilities, providing a function-driven interface to backup, statistics, housekeeping, user management and other utilities. The Services API, or parts of it, may be inaccessible to some early Classic server versions.                                                                                                                                                                                                                                                                                 |
| <b>Sets</b>                        | In relational database terms, collections of data are managed in sets consisting of one or more rows made up of one or more columns of data, each column containing one data item of a specific size and type. For example, a SELECT query specification or a view defines a set for output to a client application or PSQL module, while an UPDATE query specification defines a set upon which the specified operation is to be performed.                                                                                                                                                                          |
| <b>Shadowing and shadows</b>       | Shadowing is an optional process available on a Firebird server, whereby an exact copy of a database is maintained, warts and all, in real time, on a separate hard-disk on the same server machine where the database resides. The copy is known as a <i>database shadow</i> . The purpose is to provide a way for a database to quickly resume operation after physical damage to the hard-drive where the original database resides. A shadow is not a useful substitute for either replication or backup.                                                                                                         |
| <b>SMP</b>                         | Acronym for <i>Symmetric Multiprocessing</i> , a computer architecture that makes multiple CPUs available to a single operating system, to execute individual processes simultaneously. In theory, any idle processor can be assigned any task and, the more CPUs on the system, the better performance and load capacity.                                                                                                                                                                                                                                                                                            |
| <b>Snapshot</b>                    | SNAPSHOT is one of the three <i>transaction isolation levels</i> supported by Firebird. It provides a stable view of the database which remains current to the user of the transaction throughout the life of that transaction. It is also known as <i>concurrency level</i> isolation. See also <i>Read Committed</i> , <i>Snapshot Table Stability</i> .                                                                                                                                                                                                                                                            |

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Snapshot Table Stability</b>   | SNAPSHOT TABLE STABILITY is the most protective of the three transaction isolation levels supported by Firebird. It enforces a consistent view of the database for the user of the transaction, by preventing any other transaction from updating any row which it has selected, even if the STS transaction has not yet posted any change. It is also known as <i>consistency level</i> isolation. See also <i>Read Committed</i> , <i>Snapshot</i> .                                                                                                                                 |
| <b>SQL</b>                        | The name of a query language designed to extract meaningful sets of data from a relational database and write data to it in a completely predictable fashion. Its correct pronunciation is “ess-cue-ell”, not “sequel” as some people believe. (“Sequel” was the name of another query language, in medieval times.) It is also not an acronym for “Structured Query Language”.                                                                                                                                                                                                        |
| <b>Standard SQL, SQL Standard</b> | Refers to the syntax and implementation of SQL language elements as published by the International Standards Organization (ISO). This very complex standard prescribes definitions across an exhaustive range of syntax and functionality, at a number of levels.                                                                                                                                                                                                                                                                                                                      |
| <b>Stored Procedure</b>           | Stored procedures are compiled modules of code which are stored in the database for invocation by applications and by other server-based code modules (triggers, other procedures). They are defined to the database in a source language—procedural SQL, or PSQL—consisting of regular SQL statements as well as special SQL language extensions which supply structure, looping, conditional logic, local variables, input and output arguments, exception handling and more.                                                                                                        |
| <b>Subquery</b>                   | A query specification can define output columns that are derived from expressions. A subquery is a special kind of expression that returns a result which is itself the output of a SELECT statement. Also known as a <i>sub-select</i> or <i>embedded query</i> .                                                                                                                                                                                                                                                                                                                     |
| <b>Sub-select, subselect</b>      | See <i>Sub-query</i> . A <i>sub-selected column</i> is one which is specified or output from a sub-query. Such columns are not updatable.                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Subversion</b>                 | An open-source version control system that allows developers to keep track of different development versions of source code. The Firebird core code’s version control system is Subversion although, for many years, it was CVS, another version control software system.                                                                                                                                                                                                                                                                                                              |
| <b>Superclassic architecture</b>  | Implemented in Firebird 2.5, it is a process-threading model that superintends multiple Classic instances as threads. It is recommended for well-resourced 64-bit hosts as it makes effective use of multiple CPUs and large RAM installations. Not recommended for 32-bit host machines with more than a handful of users.<br><br>Superclassic is the model implemented as the embedded engine on all platforms from v.2.5 onwards.                                                                                                                                                   |
| <b>Superserver architecture</b>   | SuperServer is the name originally given to the process-threading multi-user model, to distinguish it from the original InterBase model that instantiates one server process for each client connection. The original model is referred to as <i>Classic</i> .                                                                                                                                                                                                                                                                                                                         |
| <b>Surrogate key</b>              | In defining a unique key, e.g. a primary key, it may occur that no column, or combination of columns, can be guaranteed to provide a unique identifier for each and every row. In that case, a column can be added and populated by values that are certain to be unique. Such a key is known as a <i>surrogate key</i> .<br><br>In Firebird, surrogate keys are most commonly implemented using generators (sequences). Surrogate keys are also frequently used as a matter of good design principle, to conform with rules of <i>atomicity</i> . See also <i>atomic</i> .            |
| <b>Sweeping</b>                   | Sweeping is the process that collects and frees obsolete versions of each record in a database when a threshold number is reached. This number, which defaults to 20,000 and is known as the sweep interval, is calculated on the difference between the <i>oldest interesting transaction</i> and the <i>oldest snapshot transaction</i> . Automatic sweeping can be disabled by setting a sweep interval of zero. A manual sweep can be invoked ad hoc, using the <i>gfx</i> utility. Sweeping is not a feature of RDBM systems that do not store multiple obsolete record versions. |
| <b>SYSDBA</b>                     | SYStem DataBase Administrator, i.e. a person with responsibility for administering databases.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>System tables</b>          | <p>Because relational database engines are self-contained, all of the metadata, or schema—data that describe the structure and attributes of database objects—are maintained within the database, in a suite of pre-defined tables which is created by the CREATE DATABASE command. These tables which store “data about data” are known as system tables. Firebird system tables all have identifiers that begin with the prefix ‘RDB\$’ and actually include data about themselves, as well as every other object in the database.</p> <p>Firebird versions 2.1+ also store definitions for <i>Monitoring Tables</i> that are instantiated and populated on demand by a user request.</p> |
| <b>Table</b>                  | A term borrowed from desktop database technology and entrenched perpetually in SQL, depicting a logical structure that stores sets of data in a tabulated format, as records (rows) of fields, each record being, by definition, identical from left to right in the number and relative placement of fields and their data types and sizes. In reality, Firebird does not store data in a physically tabulated form at all, but in contiguous blocks of disk space known as pages.                                                                                                                                                                                                         |
| <b>Transaction</b>            | A logical unit of work, which can involve one or many statements or executions of statements. A transaction begins when the client application starts it and ends when it either commits the transaction or rolls it back. A transaction is an <i>atomic</i> action: a commit must be able to commit every piece of pending work, otherwise all of its work is abandoned. A rollback, similarly, will cancel every piece of pending work that was posted since the start of the transaction.                                                                                                                                                                                                |
| <b>Transaction isolation</b>  | A mechanism by which each transaction is provided with an environment that makes it appear (to itself and its owner) that it is running alone in the database. When multiple transactions are running concurrently, the effects of all of the other transactions are invisible to each transaction, from when it starts until it is committed. Firebird supports not just one but three levels of isolation, including one level which can see the effects of other transactions as they are committed. See <i>Read Committed</i> , also <i>Snapshot</i> , <i>Snapshot Table Stability</i> .                                                                                                |
| <b>Transitively-dependent</b> | A constraint or condition where one table (C) is dependent on another table (A), because table C is dependent on another table (B) which is dependent on A. Such a dependency would arise, for example, if table B had a foreign key referencing table A’s primary key and table C had a foreign key referencing table B’s primary key. The term is also used in data modeling to refer to a condition where, during normalization, an attribute in one entity has a partial (but incomplete) dependency on a unique attribute set in another entity.                                                                                                                                       |
| <b>Table-level trigger</b>    | <p>A table-level trigger is a module of compiled code belonging to a table, that performs an action when a DML event happens to a row in that table. Any number of event actions can be coded to occur in a prescribed sequence, before and/or after an insert, update or delete operation on a table’s row, with virtually the full range of procedural SQL (PSQL) being available.</p> <p>See also <i>Database trigger</i>.</p>                                                                                                                                                                                                                                                           |
| <b>Tuple</b>                  | In relational database terminology, the “strictly correct” name for a row in a table, or a group of columns that is a subset of a row. Purists would say that <i>row</i> is the SQL name for a tuple.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>UDF</b>                    | Mnemonic for <i>User Defined Function</i> , more correctly named external function. See <i>External Function</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Unbalanced index</b>       | Firebird indexes are maintained as binary tree structures. These structures are said to be unbalanced when new nodes are added continually in a manner that causes major branching on one “side” of the binary tree. Typically, this could occur when a process inserts hundreds of thousands of new rows inside a single transaction. For this reason, it is recommended that indexes be deactivated during massive inserts. Subsequent re-activation will rebuild fully balanced indexes.                                                                                                                                                                                                 |
| <b>Uninstallation</b>         | An ugly, back-formed word, confusing to non-English speakers, since it is not found in any self-respecting dictionary—yet! Its approximate meaning is ‘a process that is the reverse of installation’, i.e. removing a previously installed software product from a computer system.. It usually doesn’t mean “deletion of component files” as installation often loads or creates artefacts in odd locations, that could cause trouble if left “installed”.                                                                                                                                                                                                                                |

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Union</b>                   | A clause in a SELECT query specification that enables the output rows of two or more SELECT statements to be combined into one final output set, as long as each of the UNIONed sets matches all of the others by the degree, data type and size of its output columns. The sets may be selected from different tables.                                                                                                                                                                                                                                                                                                                    |
| <b>Updatable view</b>          | A view is said to be updatable if it is constructed from a regular query on a single table and all of its columns exist in the underlying table. Some non-updatable views can be made updatable by the creation of triggers. See also <i>View</i> .                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Validation</b>              | Validation is a mechanism whereby new data applied to a column in a table are checked by some means to determine whether they fit a required format, value or range of values. Two ways to implement validation in the database are CHECK constraints and triggers. A CHECK constraint will throw an exception if the input data fail to test true against the given expression or constant. With triggers, the new.value can be tested more thoroughly and, if it fails, can be passed to a custom exception.                                                                                                                             |
| <b>Versioning architecture</b> | Also known as multi-generational architecture (MGA), the feature, until recently unique to Firebird and InterBase, of storing a new version of a changed row, or an inserted row on disk for the duration of a transaction, where it remains visible to that transaction even though it is not yet committed to the database. When the commit occurs, the new version becomes permanent in the database and the old version is flagged as obsolete. When considering contenders for a concurrent update in a conflict situation, the engine also uses attributes of the pending record versions concerned to determine precedence, if any. |
| <b>View</b>                    | A view is a standard SQL object which is a stored query specification that can behave in many ways like a physical table. A view does not provide physical storage for user data: it acts as a predefined container for a set of output data that exist in one or more tables.                                                                                                                                                                                                                                                                                                                                                             |
| <b>WNET</b>                    | The “Windows Networking” protocol, also known as Named Pipes and (erroneously) NetBEUI. (NetBEUI is a transport layer, not a protocol.)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>WNET error</b>              | In the firebird.log, marks an error received by the Firebird network layer from a Windows Named Pipes (“Windows networking”) client/server connection.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>XSQLDA</b>                  | Mnemonic for “eXtended SQL Descriptor Area”. It is a host-language data structure in the API which is used to transport data between a client application and the server's dynamic SQL parser module. XSQLDAs come in two types: input descriptors and output descriptors.                                                                                                                                                                                                                                                                                                                                                                 |
| <b>XSQLVAR</b>                 | Structure for defining <i>sqlvar</i> , an important field in the XSQLDA structure, used in the API for passing and returning input and output parameters.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Y valve</b>                 | Name given to the Firebird subsystem that determines which of Firebird’s several “internal engines” should be used when attaching to a database. For example, one decision is whether to attach locally or as a remote client; another is to determine whether the attachment is being attempted to a database with an incompatible on-disk structure (ODS)                                                                                                                                                                                                                                                                                |

---