

# The SQL-script language (v4)

## 1. Introduction

The **SQL-interpreter** application tries to fill a missing functionality of MS Access : executing SQL scripts. A script is a sequence of instructions stored in a text file that a DBMS (or any other server program) can execute. A script generally is written in some sort of elementary but powerful language whose use does not require intensive training. In this case, the language is SQL, augmented with some basic ancillary statements.

There are two ways to execute SQL queries in Access. The first and easiest one consists in creating a Query object in the Access main window, through the *Query* tab. Its main drawback is that a Query can include only one SQL statement. The second approach is to write a program in Visual Basic for Application (VBA) which hosts the SQL queries. VBA is a powerful but complex language obviously reserved to experienced programmers.

Access provides nothing in-between. However, many DBMS are able to execute a sequence of SQL queries stored in a text file. Such a file is a very simple SQL script. This possibility is very handy. For example, creating a database through the sequence of *create table* statements or bulk loading data from a sequence of *insert* statements does not require sophisticated programming. Some authors and vendors have developed independent scripting languages to serve the same goal, but they generally are very powerful and, as a natural consequence, fairly complex.

The goal of the **SQL-script** language described in this document is to provide non programmers with an fast and easy way to write small programs mainly made up of sequences of SQL queries. We have added a few additional statements to the language to make scripts more powerful when needed: exchanging data with the user<sup>1</sup>, using variables, performing simple computations, conditional and iterative execution.

An SQL-script file can be processed (i.e., executed) by a dedicated interpreter called SQL-interpreter. SQL-interpreter is a small MS Access application with no tables, no queries and no reports. It includes a small form and a module which codes the functionalities of the interpreter. The form allows a script to be selected (Locate script) and executed (Execute script). The application is available in French (SQL-interpreter-FR (v4).mdb) and in English (SQL-interpreter-EN (v4).mdb). A collection of script examples is available in file SQL-script-applications.zip.

This document describes the 4th version of the SQL-interpreter application and of its language.

## 2. SQL-script principles

An *SQL-script* script is a text file comprising a sequence of lines. A line is any contiguous chain of characters terminated by an *end-of-line* or an *end-of-file* mark. There are three kinds of lines:

1. An **blank line** is a line comprising no characters or comprising space and/or tab characters only. Empty lines are ignored by the interpreter, but they are useful to make the script more readable and to indicate the end of the preceding statement.
2. A **comment line** is a line whose first two non-space characters are "--". It will be used to insert information in the script. A comment line is ignored by the interpreter.

---

<sup>1</sup> We distinguish two roles: the **script writer** (the reader of this document) who designs and develops scripts and the **script user**, who has the responsibility to execute the script and to exploit its results.

### 3. A **statement line** contains a statement or a statement fragment.

A statement is a sentence that orders the interpreter to execute a specific action. A statement spans one or several consecutive statement lines. The leading and trailing spaces of a statement line are ignored. When a statement is made up of several lines, it is reconstructed, before execution, by the concatenation of all its lines (leading/trailing spaces excluded), a space character being inserted between any two consecutive lines. It is therefore not recommended to cut a statement in the middle of an SQL constant or keyword. The last line of a statement is recognized when at least one of the following conditions is encountered:

- it ends with a semicolon character (" ; "),
- the next line is a blank line,
- it is the last line of the file.

When the right-most component of a statement is defined as <long-string>, this component is made up of all the characters until the end of the statement, even if it spans several lines.

In a statement, two consecutive words are separated by at least one space character. For example, the expression "compute I = I + 1" is a valid *compute* statement, while "compute I=I+1" is incorrect (I=I+1 is understood as a variable name, so that the remaining of the statement is missing).

It is recommended to give script files the extension "\*.sql". An empty text is a valid script.

The language comprises five families of statements: *SQL* statements, *dialog* statements, *variable manipulation* statements, *control* statements and *context* statements.

- *SQL statements* include the main DDL and DML statements.
- *Dialog statements* allows the script to display data and to ask the user to introduce data.
- *Variable manipulation statements* assign values to variables, including the result of an SQL SFW statement or of a computation.
- *Control statements* allows branching, conditional execution and loops to be defined.
- *Context statements* defines the context in which statements will be executed.

The language is case insensitive as far as the key words are concerned: *compute*, *Compute* and *COMPUTE* all denote the same verb. On the contrary, variable names are case sensitive. SQL statements follow the SQL syntax when the variable names they include have been resolved.

*Note.* Since the current script is read as a sequential file and is not prefetched in main memory, there is no limit to the size of the script files.

### 3. SQL-script variables

An SQL-script variable is a small piece of memory which has a *name* and a contents, called its *value*. It is possible to create a variable, to assign it a value and to reuse this value later on.

The name of a variable uniquely identifies it in the script. A variable name is a *word*, that is, a character string that does not include spaces, tabs or any control characters; other symbols, punctuation signs and digits are allowed. Variable names are case sensitive: *PRICE* and *Price* are two different variables. *UNIT-PRICE DEPARTMENT/SERVICE/Head See\_Me M1234 (A+B)\*C/2 "Hello" ' ; , 0 12.34 0,0025* are valid variable names<sup>2</sup>.

---

<sup>2</sup> In this list, names are separated by one space character.

A variable always contains a value, which is a character string, possibly empty. A variable has no specific type : its value can be interpreted as a word, a name, a sentence, a statement fragment, a database name or a number, according to the way it is used. A variable is created the first time it is assigned a value through a *Set*, *Ask* or *Compute* statement. A variable can be reused but not renamed nor deleted.

By default, up to 20 variables can be created in a script. Going beyond this limit or referencing an unknown variable stop the script execution. This limit can be changed by modifying the constant `MaxIVV` in the code of the interpreter:

```
Const MaxIVV As Integer = 20      ' Maximum number of variables in a script.
```

Once it has been created, a variable can be used in any statement by inserting its name enclosed by variable delimiters (character `$` by default). Considering the variable `COL`, whose value is, say, "CITY" (double quotes excluded) each occurrence of `$COL$` in a statement is replaced by the value of `COL` before execution of the statement. For example, the statement

```
Delete CLIENT where $COL$ = 'Paris';
```

will be first transformed into

```
Delete CLIENT where CITY = 'Paris';
```

then executed. Similarly, the value of the variable `PRICE` will be displayed through the statement

```
Display The value of variable PRICE is $PRICE$;
```

## 4. SQL Statements

Most SQL-DDL and SQL-DML statements can be used in an *SQL-script* script. They allow a user to *create*, *modify* and *suppress* data structures, as well as to *insert*, *modify* and *delete* data. Since a script has very limited capabilities in user interaction and in internal data storage, the *select-from-where* statement cannot be included in a script. The recommended way to extract high volume of data from the database consists in defining a result table through a *create table* statement then in inserting in this table the result of a *select-from-where* instruction through an *insert* statement.

### ▪ Format

```
<Create-St> ::= any SQL create statement
```

```
<Alter-St>  ::= any SQL alter statement
```

```
<Drop-St>   ::= any SQL drop statement
```

```
<Insert-St> ::= SQL insert statement
```

```
<Update-St> ::= SQL update statement
```

```
<Delete-St> ::= SQL delete statement
```

### ▪ Examples

```
create table CLIENT(CNUM char(8) not null,
                   Name char(30) not null,
                   CITY char(20 not null);

insert into CLIENT values ('C400', 'FERARD', 'Poitiers');
insert into CLIENT values ('K111', 'VANBIST', 'Lille');
insert into CLIENT values ('F010', 'TOUSSAINT', 'Poitiers');
insert into CLIENT values ('L422', 'FRANCK', 'Namur');

set CITY = 'Poitiers';
```

```

create table CLI_${CITY$(CNUM char(8) not null,
                        Name char(30) not null);
insert into CLI_${CITY$(CNUM, NAME)
select CNUM, NAME from CLIENT where CITY = '${CITY$';

```

## 5. Dialog Statements

The *Dialog* statements allow a user and a script to exchange data at execution time. The *Ask* statement is the simplest way for a user to transmit values to the script, while the script will generally send results and messages to the user through the *Display* statement. A higher volume of data will be transmitted through result database tables.

### 5.1 Ask statement

- *Description*

Asks the user to enter a value and assign it to a variable.

- *Format*

```

<ask-St> ::= "ask" <var-name> "=" <prompt>
<prompt> ::= <long-string>

```

- *Function*

Opens a dialog box displaying the message <prompt> and waiting for the user to enter a value. This value is assigned to variable <var-name> as a character string. If the variable does not exist, it is created first.

- *Examples*

```

Ask TITLE = What is the book title?;
Set Q = Number of iterations;;
Ask N = $Q$;

```

### 5.2 Display statement

- *Description*

Displays a message in a dialog box.

- *Format*

```

<display-St> ::= "display" <displ-string>
<displ-string> ::= character string in which symbols "@" stand for EOL

```

- *Function*

Each character "@" in <displ-string> is replaced by a *end-of-line* character. Then, the resulting character string is displayed in a message box until the user closes it. Messages need not be enclosed between quotes.

- *Examples*

```

Display The script is finished;
Display This procedure computes the price of each product.
@Please first load the BOM database.@Be sure that all
the raw products have a unit price.;
Display The total cost is $Total-Cost$;

```

## 6. Variable manipulation Statements

A variable manipulation statement changes the value of a variable. Note that, according to this definition, the *Ask* statement is a variable manipulation statement as well. SQL-script includes four statements : *Ask*, which stores in a variable a value provided by the user, *Set*, which assigns a constant, *Compute*, which stores a value extracted from the database or computed from other values and *USnumber* and *FRnumber* which change the decimal symbol of a number.

### 6.1 Set statement

- *Description*

Assigns a value to a variable.

- *Format*

```
<set-St> ::= "set" <var-name> "=" <long-string>
```

- *Function*

The character string <long-string> is assigned to the variable <var-name>. If this variable does not exist, it is created first.

- *Examples*

```
Set TITLE = An Introduction to Database Systems;
Set QUESTION = Do you know "$TITLE$" from Date?;
Set TEN = 10
Set 1 = 1;
```

### 6.2 Compute statement

- *Description*

Computes a value and assign it to a variable.

- *Format*

```
<compute-St> ::= "compute" <var-name> "=" <comp-expression>
<comp-expression> ::= <SQL-statement> | <calc-expression>
<calc-expression> ::= <var-name> <num-oper> <var-name>
<num-op>          ::= "+" | "-" | "*" | "/"
```

- *Function*

The expression < comp-expression > is evaluated and its value is assigned to the variable < var-name > as a character string. If the variable does not exist, it is created first. There are two forms of the *compute* statement, according to the way the value is computed through <comp-expression>.

1. *Database value.* The value is extracted from the database through an SQL *select* statement that returns a single column, single row result. This result is assigned to the variable as a character string. If the expression returns more than one row, only the first one is used. The result expression of the select-list **must be assigned alias "A"**. If a numeric constant is injected in the *select*, *where*, *order by*, *group by* or *having* clauses, it must comply with the US format (decimal point) required by SQL. If no row is found, the value of the variable is unchanged. If the extracted value is numeric, it complies with the current numeric format (decimal point or comma). The execution halts if the SQL statement is invalid.

2. *Calculated value.* The arithmetic expression <calc-expression> is evaluated and its result is assigned to the variable as a character string. The result complies with the current numeric format (decimal point or comma) of the script. The arithmetic expression uses variables only. If a constant is needed, it must first be stored in a variable (whose name can be the constant itself). The execution halts if the arithmetic expression is invalid.

*Note.* More complex expressions can be built in two ways. First by decomposing the complex expression into a sequence of simpler, binary, computations. Second, by relying on the SQL capacity to carry out complex computations in the *select* clause. See examples in Appendix 2.

- **Examples**

```
Compute AVERAGE = select avg(ACCOUNT) as A from CUSTOMER;
USnumber AVERAGE;
Compute N = select count(*) as A from CUSTOMER
           where ACCOUNT >= $AVERAGE$;
Compute C = TOTAL / N;
Set 1 = 1;
Compute I = I + 1;
```

### 6.3 USnumber statement

- **Description**

Transforms the value of a variable into the US numeric format.

- **Format**

```
<USnumber-St> ::= "USnumber" <var-name>
```

- **Function**

Replaces the first comma (",") of the value of the variable by a point ("."). For instance, the value "123,45" is transformed into "123.45". The value is supposed to be numeric, otherwise, the result could be undefined.

- **Example**

```
USnumber TOTAL-AMOUNT;
```

### 6.4 FRnumber statement

- **Description**

Transforms the value of a variable into the FR numeric format.

- **Format**

```
<FRnumber-St> ::= "FRnumber" <var-name>
```

- **Function**

Replaces the first point (".") of the value of the variable by a comma (","). For instance, the value "123.45" is transformed into "123,45". The value is supposed to be numeric, otherwise, the result could be undefined.

- **Example**

```
FRnumber TOTAL-AMOUNT;
```

## 7. Control Statements

Normally, the statements of a script are executed sequentially, that is, in the order they have been written in the script file, from the first line to the last one. A control statement can be used to change this natural order. A *Goto* statement forces the execution to continue at a given point in the script, identified by a label (specified by a *Label* statement). An *If* statement specifies a statement that will be executed conditionally.

### 7.1 Label statement

- *Description*

Defines a labelled point in the script.

- *Format*

```
<label-St> ::= "label" <label>
```

- *Function*

This statement has no effect. This point can be the target of *goto*'s statements. If two points are given the same label, only the first one will be retrieved. This statement cannot include variables.

- *Examples*

```
Label SEARCH-NEXT;
```

### 7.2 Goto statement

- *Description*

Continues the execution at a given point in the script.

- *Format*

```
<goto-St> ::= "goto" <label>
```

- *Function*

The execution of the script continues at the statement that follows the point labelled <label>. If the label is unknown, the execution halts.

- *Examples*

```
Goto SEARCH-NEXT;  
Goto $PRODUCT-TYPE$;
```

### 7.3 If statement

- *Description*

Executes a statement if the specified condition is satisfied.

- *Format*

```
<if-St> ::= "if" <condition> <Statement>  
<condition> ::= <var-name> <comp-op> <var-name>  
<comp-op> ::= "=" | "<" | ">" | ">=" | "<=" | "<>"
```

- *Function*

The condition <condition> is evaluated. If it is *true*, then the statement <Statement> is executed. The condition compares the values of two variables. If the operator or a variable are unknown, the execution halts.

*Note.* More complex conditions can be built in three ways. First by decomposing the complex condition into a tree of simpler *if* statements. Second, provided only *and* and *not* operators are needed (logical conjunction), by nesting an *if* statement in another *if* statement. Third, by relying on the SQL capacity to evaluate complex conditions in the *where* clause. See examples in Appendix 2.

- **Examples**

```
If PRICE > MAX-PRICE goto PRICE-ERROR;
If I <> 0 compute M = M + I;
If I >= MIN If I <= MAX goto COMPUTE-REBATE;
if ANSWER = YES compute N = select count(*) as A from CLIENT;
```

## 8. Context Statements

A statement is executed in a definite context, which is made up of a set of default parameters that govern the way the statements are executed. These parameters apply until one decides to change them explicitly through Context statements.

### 8.1 Delimiter statement

- **Description**

Changes the variable delimiter.

- **Format**

```
<delimiter-St> ::= "Delimiter" <character>
```

- **Function**

The default variable delimiter is "\$", as in \$PRICES\$. This character can be changed, notably when the default character is part of a string. This new character will be used until changed.

- **Examples**

```
Delimiter ' ;
Ask C = Hello 'USER-NAME', please enter one the currencies : $, £, €.;
Set DOLLAR = $;
Delimiter $;
```

### 8.2 Numeric-If statement

- **Description**

Defines the interpretation of future comparisons as *Numeric* comparison.

- **Format**

```
<Numeric-If-St> ::= "Numeric-If"
```

- **Function**

All the SQL-script variables contain character strings. There is no means to associate a type with a variable. The way the value of a variable is interpreted depends on the nature of the operation that is carried out. The value "123,45" will be interpreted as a character string in a *Display* statement and as a numeric value in a *Compute* statement. In an *If* statement, when two values are compared, the result can be different according to whether the values are interpreted as character strings or as numeric values. For instance, "10" < "2", but 10 > 2. The *Numeric-If* statement states that, from now on, the



values of variables in the <condition> clauses of *If* statements will be considered as numerical values. Therefore, the condition "10" > "2" will be true. In the basic version of SQL-Interpreter, numeric interpretation is the default mode.

- **Examples**

```
Set A = 19,25;
Set B = 2,7;
Numeric-If;
If A > B goto A-greater-than-B;
```

### 8.3 String-If statement

- **Description**

Defines the interpretation of future comparisons as *String* comparison.

- **Format**

```
<String-If-St> ::= "String-If"
```

- **Function**

The *String-If* statement states that, from now on, the values of variables in the <condition> clauses of *If* statements will be considered as character strings. For instance, the condition "10" > "2" will be false. This mode must be chosen when comparing character strings.

- **Examples**

```
Set A = Alice;
Set B = Jean;
String-If;
If A < B set FIRST = $A$;
Numeric-If;
```

### 8.4 Default-If statement

- **Description**

Sets the interpretation of future comparisons to the default mode.

- **Format**

```
<Default-If-St> ::= "Default-If"
```

- **Function**

Set the interpretation of future comparisons to the default mode. In the basic version of SQL-Interpreter, the default mode is *Numeric*.

- **Examples**

```
Default-If
```

## Appendix 1 - Syntax of SQL-Script (v4)

<Comment> ::= "--" <string>

<Statement> ::= <SQL-St> | <Dialog-St> | <Variable-St>  
| <Control-St> | <Context-St>

<SQL-St> ::= <Create-St> | <Alter-St> | <Drop-St>  
| <Insert-St> | <Update-St> | <Delete-St>

<Dialog-St> ::= <ask-St> | <display-St>

<Variable-St> ::= <set-St> | <compute-St> | <USnumber-St>

<Control-St> ::= <label-St> | <goto-St> | <if-St>

<Context-St> ::= <delimiter-St> | <Numeric-If-St> | <String-If-St>  
| <Default-If-St>

### SQL Statements

<Create-St> ::= *any SQL create statement*

<Alter-St> ::= *any SQL alter statement*

<Drop-St> ::= *any SQL drop statement*

<Insert-St> ::= *SQL insert statement*

<Update-St> ::= *SQL update statement*

<Delete-St> ::= *SQL delete statement*

### Dialog Statements

<Ask-St> ::= "ask" < var-name > "=" <long-string>

<Display-St> ::= "display" <displ-string>

<displ-string> ::= <long-string> in which symbols "@" stand for EOL

### Variable manipulation Statements

<Set-St> ::= "set" <var-name> "=" <long-string>

<Compute-St> ::= "compute" <var-name> "=" <comp-expression>

<USnumber-St> ::= "USnumber" <var-name>

<FRnumber-St> ::= "FRnumber" <var-name>

<comp-expression> ::= <SQL-statement> | <calc-expression>

<calc-expression> ::= <var-name> <num-oper> <var-name>

<num-op> ::= "+" | "-" | "\*" | "/"

### Control Statements

<Label-St> ::= "label" <label>

<Goto-St> ::= "goto" <label>

<If-St> ::= "if" <condition> <Statement>

<label> ::= <long-string>

<condition> ::= <var-name> <comp-op> <var-name>

<comp-op> ::= "=" | "<" | ">" | ">=" | "<=" | "<>"

### **Context Statements**

<delimiter-St> ::= "Delimiter" <character>

<Numeric-If-St> ::= "Numeric-If"

<String-If-St> ::= "String-If"

<Default-If-St> ::= "Default-If"

### **Miscellaneous**

<var-name> ::= <word>

<word> ::= *character string without spaces*

<string> ::= *character string extracted from a single line*

<long-string> ::= *character string reconstructed by line concatenation*

<character> ::= *any non space character*

## Appendix 2 - Sample scripts

### A2.1 Creation and loading of the CLICOM database - July 2008

```
-- Create database tables

create table CLIENT
(NCLI char(8) not null, NOM char(18) not null,
ADRESSE char(24) not null, LOCALITE char(20) not null,
CAT char(2), COMPTE long not null,
constraint PKCLI primary key (NCLI));

create table PRODUIT
(NPRO char(10) not null, LIBELLE char(30) not null,
PRIX long not null, QSTOCK long not null,
constraint PKPRO primary key (NPRO));

create table COMMANDE
(NCOM char(8) not null, NCLI char(8) not null, DATECOM date not null,
constraint PKCOM primary key (NCOM),
constraint FKCOMCLI foreign key (NCLI) references CLIENT);

create table DETAIL
(NCOM char(8) not null, NPRO char(10) not null, QCOM integer not null,
constraint PKDET primary key (NCOM,NPRO),
constraint FKDETCOM foreign key (NCOM) references COMMANDE,
constraint FKDETPRO foreign key (NPRO) references PRODUIT);

-- Load sample data

insert into CLIENT values ('B112','HANSENNE' , '23, a. Dumont' , 'Poitiers' , 'C1',1250);
insert into CLIENT values ('C123','MERCIER' , '25, r. Lemaitre' , 'Namur' , 'C1',-2300);
insert into CLIENT values ('B332','MONTI' , '112, r. Neuve' , 'Geneve' , 'B2',0);
insert into CLIENT values ('F010','TOUSSAINT' , '5, r. Godefroid' , 'Poitiers' , 'C1',0);
insert into CLIENT values ('K111','VANBIST' , '180, r. Florimont' , 'Lille' , 'B1',720);
insert into CLIENT values ('S127','VANDERKA' , '3, av. des Roses' , 'Namur' , 'C1',-4580);
insert into CLIENT values ('B512','GILLET' , '14, r. de l'Ete' , 'Toulouse' , 'B1',-8700);
insert into CLIENT values ('B062','GOFFIN' , '72, r. de la Gare' , 'Namur' , 'B2',-3200);
insert into CLIENT values ('C400','FERARD' , '65, r. du Tertre' , 'Poitiers' , 'B2',350);
insert into CLIENT values ('C003','AVRON' , '8, ch. de la Cure' , 'Toulouse' , 'B1',-1700);
insert into CLIENT values ('K729','NEUMAN' , '40, r. Bransart' , 'Toulouse' , null,0);
insert into CLIENT values ('F011','PONCELET' , '17, Clos des Erables' , 'Toulouse' , 'B2',0);
insert into CLIENT values ('L422','FRANCK' , '60, r. de Wepion' , 'Namur' , 'C1',0);
insert into CLIENT values ('S712','GUILLAUME' , '14a, ch. des Roses' , 'Paris' , 'B1',0);
insert into CLIENT values ('D063','MERCIER' , '201, bvd du Nord' , 'Toulouse' , null,-2250);
insert into CLIENT values ('F400','JACOB' , '78, ch. du Moulin' , 'Bruxelles' , 'C2',0);

insert into PRODUIT values ('CS262','CHEV. SAPIN 200x6x2', 75, 45);
insert into PRODUIT values ('CS264','CHEV. SAPIN 200x6x4', 120,2690);
insert into PRODUIT values ('CS464','CHEV. SAPIN 400x6x4', 220, 450);
insert into PRODUIT values ('PA45' , 'POINTE ACIER 45 (1K)',105, 580);
insert into PRODUIT values ('PA60' , 'POINTE ACIER 60 (1K)', 95, 134);
insert into PRODUIT values ('PH222','PL. HETRE 200x20x2', 230, 782);
insert into PRODUIT values ('PS222','PL. SAPIN 200x20x2', 185,1220);

insert into COMMANDE values ('30178','K111','21/12/2005');
insert into COMMANDE values ('30179','C400','22/12/2005');
insert into COMMANDE values ('30182','S127','23/12/2005');
insert into COMMANDE values ('30184','C400','23/12/2005');
insert into COMMANDE values ('30185','F011','2/12/2006');
insert into COMMANDE values ('30186','C400','2/1/2006');
insert into COMMANDE values ('30188','B512','3/1/2006');

insert into DETAIL values ('30178','CS464',25);
insert into DETAIL values ('30179','PA60',20);
insert into DETAIL values ('30179','CS262',60);
insert into DETAIL values ('30182','PA60',30);
insert into DETAIL values ('30184','CS464',120);
```

```

insert into DETAIL values ('30184','PA45',20);
insert into DETAIL values ('30185','PA60',15);
insert into DETAIL values ('30185','PS222',600);
insert into DETAIL values ('30185','CS464',260);
insert into DETAIL values ('30186','PA45',3);
insert into DETAIL values ('30188','PA60',70);
insert into DETAIL values ('30188','PH222',92);
insert into DETAIL values ('30188','CS464',180);
insert into DETAIL values ('30188','PA45',22);

display Loading completed;

```

## A2.2 Update quantities on hand of products - July 2008

```

-- Update a table from values in another table.
-- Beware: Access "update" style. To change for another SQL DBMS

display Update products from quantities ordered from a given City;

ask VAL = Name of the city ?;

update PRODUIT P, DETAIL D
set   QSTOCK = QSTOCK - QCOM
where P.NPRO = D.NPRO
and   D.NCOM in (select NCOM
                  from COMMANDE
                  where NCLI in (select NCLI
                                from CLIENT
                                where LOCALITE = '$VAL$'));

```

## A2.3 Delete CLICOM database - July 2008

```

display Supprimer la base de données CLICOM;

drop table DETAIL;
drop table COMMANDE;
drop table CLIENT;
drop table PRODUIT;

```

## A2.4 Creation and loading of the BOM database - July 2008

```

-- Create the Bill-of-Material database

display
Script de création de la BD BOM comportant les deux tables suivantes :
@- PRODUIT
@- COMPOSITION
@@Création des deux tables et introduction de leur contenu.
@Juillet 2008

create table PRODUIT
(NPRO      char(10) not null,
LIBELLE   char(8)  not null,
PRIX_U    single,
constraint PKP primary key (NPRO));

create table COMPOSITION
(COMPOSE   char(10)  not null,
COMPOSANT char(10)  not null,
QTE       short not null,
constraint PKC primary key (COMPOSE,COMPOSANT),

```

```

constraint FKCP1 foreign key (COMPOSE) references PRODUIT,
constraint FKCP2 foreign key (COMPOSANT) references PRODUIT);

```

```
-- Load sample data
```

```

insert into PRODUIT(NPRO,LIBELLE) values ('p1','A-200');
insert into PRODUIT(NPRO,LIBELLE) values ('p2','A-056');
insert into PRODUIT(NPRO,LIBELLE) values ('p3','B-661');
insert into PRODUIT(NPRO,LIBELLE) values ('p4','B-122');
insert into PRODUIT(NPRO,LIBELLE) values ('p5','B-326');
insert into PRODUIT values ('p6','D-822',3.5);
insert into PRODUIT values ('p7','D-507',8.0);
insert into PRODUIT values ('p8','G-993',5.0);
insert into PRODUIT(NPRO,LIBELLE) values ('p9','F-016');
insert into PRODUIT(NPRO,LIBELLE) values ('p10','J-500');
insert into PRODUIT values ('p11','J-544',0.5);
insert into PRODUIT values ('p12','L-009',1.7);

```

```

insert into COMPOSITION values ('p1','p2',2);
insert into COMPOSITION values ('p1','p3',1);
insert into COMPOSITION values ('p1','p4',2);
insert into COMPOSITION values ('p2','p7',8);
insert into COMPOSITION values ('p2','p8',2);
insert into COMPOSITION values ('p3','p8',5);
insert into COMPOSITION values ('p4','p8',4);
insert into COMPOSITION values ('p4','p9',5);
insert into COMPOSITION values ('p4','p10',5);
insert into COMPOSITION values ('p5','p4',2);
insert into COMPOSITION values ('p5','p6',7);
insert into COMPOSITION values ('p9','p11',2);
insert into COMPOSITION values ('p10','p11',4);
insert into COMPOSITION values ('p10','p12',3);

```

## A2.5 Computing product prices in the BOM database - July 2008

```

--      Compute product price in the BOM database
--      -----

-- Only primary products have a unit price. The price of a compound product can be
-- computed from the price of its component products, provided each of them has a price.
-- The recursive procedure has been transformed into an iterative procedure.
-- The procedure ends when all thye products have a price.

display Calcul des prix des produits finis et semi-finis
@@ Au préalable, créer et charger la BD BOM.
@@ A chaque itération, on indique le nombre de produits qui restent à mettre à jour.;

      set 0 = 0;
      set TOTAL = 0;

label UPDATE;

-- y a-t-il encore des produits à mettre à jour ?

      compute N = select count(*) as A
                  from PRODUIT
                  where PRIX_U is null;

      compute TOTAL = TOTAL + N;

      display Il reste  $N$ produits sans prix;

      if N = 0 goto FIN;

-- MAJ des composés sans PRIX_U dont tous les composants ont un PRIX_U
-- (spécifique à la forme d'update d'Access)

```

```

update PRODUIT as H, COMPOSITION as C, PRODUIT as B
set H.PRIX_U = iif(H.PRIX_U is null,0,H.PRIX_U) + C.QTE*B.PRIX_U
where H.NPRO = C.COMPOSE and C.COMPOSANT = B.NPRO
and H.PRIX_U is null
and not exists(select *
                from COMPOSITION as CC, PRODUIT as BB
                where CC.COMPOSE = H.NPRO
                and CC.COMPOSANT = BB.NPRO
                and BB.PRIX_U is null);

goto UPDATE;

label FIN;

display Mise à jour terminée.@@$TOTAL$ produits ont été mis à jour.;

```

## A2.6 Database migration - July 2008

```

-- A new database is created from data extracted from the CLICOM database. This database
-- contains the data of customers that meet a definite criterion specified by the user.
-- These data comprise the CLIENT rows and the depending COMMANDE and DETAIL rows.
-- The migrated rows are then deleted from the CLICOM database.

-- Ask the selection criterion of the data to migrate (Column name and value)

ask CRITERION = Donnez le nom de la colonne de sélection sur CLIENT;
ask VALUE = Donnez la valeur de la colonne $CRITERION$;

-- Create new tables

create table CLIENT_$CRITERION$_$VALUE$
(NCLI char(8) not null,
 NOM char(18) not null,
 ADRESSE char(24) not null,
 LOCALITE char(20) not null,
 CAT char(2),
 COMPTE long not null);

create table COMMANDE_$CRITERION$_$VALUE$
(NCOM char(8) not null,
 NCLI char(8) not null,
 DATECOM date not null);

create table DETAIL_$CRITERION$_$VALUE$
(NCOM char(8) not null,
 NPRO char(10) not null,
 QCOM integer not null);

-- Migrate the selected data to the new tables

insert into CLIENT_$CRITERION$_$VALUE$
select NCLI,NOM,ADRESSE,LOCALITE,CAT,COMPTE
from CLIENT
where $CRITERION$ = '$VALUE$';

insert into COMMANDE_$CRITERION$_$VALUE$
select NCOM, NCLI, DATECOM
from COMMANDE
where NCLI in (select NCLI from CLIENT
               where $CRITERION$ = '$VALUE$');

insert into DETAIL_$CRITERION$_$VALUE$
select NCOM, NPRO, QCOM
from DETAIL
where NCOM in (select NCOM from COMMANDE
               where NCLI in (select NCLI from CLIENT
                             where $CRITERION$ = '$VALUE$'));

-- Delete migrated data from the source tables

```

```

delete from DETAIL
where NCOM in (select NCOM from COMMANDE
               where NCLI in (select NCLI from CLIENT
                              where $CRITERION$ = '$VALUE$'));

delete from COMMANDE
where NCLI in (select NCLI from CLIENT
               where $CRITERION$ = '$VALUE$');

delete from CLIENT
where $CRITERION$ = '$VALUE$';

-- Compute migration statistics

compute N = select count(*) as A from CLIENT_$CRITERION$_$VALUE$;
compute N = select $N$ + count(*) as A from COMMANDE_$CRITERION$_$VALUE$;
compute N = select $N$ + count(*) as A from DETAIL_$CRITERION$_$VALUE$;

display Migration completed@$N$ rows have been migrated to the new tables;

```

## A2.7 Complex computation - July 2008

```

-- How to perform complex computations with the help of the SQL engine?
-- The trick is based of an arbitrary working table that contains one row.
-- Oracle provides a built-in table for this purpose: the DUAL table.

-- Create temporary technical table (1 column, 1 row):

    create table T(A integer not null);
    insert into T(A) values(1);

    set X = 1;
    set Y = 2;
    set Z = 3;

-- Let's compute N = (X/Y)*Z in one operation:

    compute N = select ($X$ / $Y$) * $Z$ as A from T;
    display N = (X/Y)*Z : ($X$ / $Y$) * $Z$ = $N$

-- Another one : N = (X+Y)/5

    compute N = select ($X$ + $Y$) / 5 as A from T;
    display N = (X+Y)/5 : ($X$ + $Y$) / 5 = $N$

-- Let's improve the look of these statements:

    delimiter ';
    set = = select;
    set ; = as A from T;

    compute N = '=' ('X' / 'Y') * 'Z' ';';
    display N = (X/Y)*Z : ('X' / 'Y') * 'Z' = 'N';

-- Not perfect but more readable!

drop table T;

```

## A2.8 Complex conditions - July 2008

```

-- How to evaluate complex conditions with the help of the SQL engine?
-- The trick is based of an arbitrary working table that contains one row.

```



```
-- Create temporary technical table (1 column, 1 row) (= table DUAL in Oracle):

create table T(A integer not null);
insert into T(A) values(1);

ask A = Premier nombre ;;
ask B = Deuxième nombre ;;
ask C = Troisième nombre ;;

-- Let's evaluate (A < B or A < C) and B < C

compute N = select count(*) as A from T where ($A$ < $B$ or $A$ < $C$) and $B$ < $C$;
display Result of ($A$ < $B$ or $A$ < $C$) and $B$ < $C$ = $N$;

drop table T;

-- Through decomposition into standard If statements:

if B >= C goto False;
if A < B goto True;
if A < C goto True;
goto False;

label True;
display True;
goto End;

label False;
display False;

label End;
```

## A2.9 Creation of the table of the first N square numbers - July 2008

```
set 1 = 1;
set 40000 = 40000;

create table SQUARE
  (NUM long not null,
   SQUARE long not null);

label RETRY;
ask MAX = Number of squares to create (de 1 à 40000) : ;
if MAX >= 1 if MAX <= 40000 goto COMPUTATION;
goto RETRY;

label COMPUTATION;

set N = 0;
label NEXT-NUMBER;
if N = MAX goto FIN;
compute S = N * N;
insert into SQUARE(NUM, SQUARE) values($N$, $S$);
compute N = N + 1;
goto NEXT-NUMBER;

label FIN;
display Loading completed;
```

## A2.10 Comparison modes - July 2008

```
display Playing with numeric and string comparison modes
@Example : enter 2 and 10
@The script will display the comparisons that are true in each mode;

ask A = Give the first value;
ask B = Give the second value;

display Comparison in default mode (normally Numeric);
If A < B display A < B : $A$ < $B$;
If A = B display A = B : $A$ = $B$;
If A > B display A > B : $A$ > $B$;
If A <= B display A <= B : $A$ <= $B$;
If A <> B display A <> B : $A$ <> $B$;
If A >= B display A >= B : $A$ >= $B$;

display Comparison in String mode;
string-if;
If A < B display A < B : $A$ < $B$;
If A = B display A = B : $A$ = $B$;
If A > B display A > B : $A$ > $B$;
If A <= B display A <= B : $A$ <= $B$;
If A <> B display A <> B : $A$ <> $B$;
If A >= B display A >= B : $A$ >= $B$;

display Comparison in Numeric mode;
numeric-if;
If A < B display A < B : $A$ < $B$;
If A = B display A = B : $A$ = $B$;
If A > B display A > B : $A$ > $B$;
If A <= B display A <= B : $A$ <= $B$;
If A <> B display A <> B : $A$ <> $B$;
If A >= B display A >= B : $A$ >= $B$;
```

## A2.11 Simulation of an array - July 2008

```
-- SQL-script does not includes arrays of values (tables are normally used instead).
-- Th is script uses 12 variables named T1 to T12.
-- Theses variables are used as if they were the cells of an array.

set 0 = 0;
set 1 = 1;
set 12 = 12;

-- creation and initialisation of an array of 12 elements (12 variables T1 to T12)

set I = 0;
label CREATION;
if I = 12 goto LECTURE;
compute I = I + 1;
compute T$I$ = I + I;
goto CREATION;

-- Random reading array elements

label READING;
ask I = Indice de l'élément à consulter (de 1 à 12 ou 0) ;
if I = 0 goto END;
if I < 0 goto READING;
if I > 12 goto READING;
compute V = T$I$ + 0;
display T[$I$] = $V$;
goto READING;

label END;
display End;
```

## A2.12 A tiny interpreter - July 2008

```
-- This small script allows a user to enter a statement (typically an SQL instruction)
-- and executes it.

-- Sample input :
--   create table T(I integer, J integer)
--   insert into T values (1,2)
--   insert into T values (1,2)
--   display $N$ (not quite legal, but it works!)
--   STOP

display This script is an elementary interactive interpreter of SQL statements (and more)!
@Enter full SQL statements once at a time.
@Enter STOP to exit.;

set STOP = STOP;
set N = 0;
set l = 1;

label BEGIN;
  ask QUERY = Enter an SQL statement or STOP;
  if QUERY = STOP goto END;
  $QUERY$;
  compute N = N + 1;
  goto BEGIN;

label END;
  display Execution stopped@$N$ queries have been executed;
```