UNIVERSITÉ DE NAMUR

# SQLfast manual

## Database processing made easy

***Learning*** databases and database programming
***Prototyping*** database applications
***Developing*** light database applications

Version 6

September 2024

University of Namur ● Faculté d'informatique
Namur Digital Institute (NADI) - *http://www.info.fundp.ac.be/libd*

**Namur Digital Institute (NADI)**
**PReCISE Research Center** - **Data Engineering group**
University of Namur
Faculté d'Informatique
Rue Grandgagnage, 21 - B-5000 Namur (Belgium)

*NADI web site*: https://directory.unamur.be/entities/nadi
*Data Engineering group*: https://bit.ly/3596et9
*Contact*: Jean-Luc Hainaut, jean-luc.hainaut@unamur.be

*Printed 5/10/24*

# Foreword

(to be updated)

**SQLfast** is a *language* and a *software environment* that allow novice or casual users to write and execute small scripts to create, modify, consult, manipulate and process relational databases.

This volume covers the main programming concepts of the SQLfast language: database structure, SQL, variables, graphical user interaction, programming structures (sequence, alternative, loops, procedure), expressions, data formatting, data presentation, large data objects, metadata, recursive programming, etc.)

As a companion to this book, a collection of representative database applications is also available. These applications encompass a wide range of problems from data management (e.g., a complete management system for a school library and a bill of material system) to system applications (directory management, SQL injection, temporal databases, large test data set generation, database prototyping), to decision systems (topological sorting, formal concept analysis, graph algorithms, text analysis) and to more playful applications (an adventure game, a novel generator, Conway's game of life, maze solver, writing a successful book).

This material can be downloaded from the SQLfast website (**https://bit.ly/3LfUa8z**). Most tutorial chapters are already available, though sometimes in a draft version. Others still are being written. The code of these chapters is available in the **Script** folder.

Document *SQLfast Commands* contains a short description of all the SQLfast statements, functions, parameters and system variables. It is available on the SQLfast website.

Jean-Luc Hainaut - Namur, December 2022

# Table of contents

**Chapter 1 - Introduction 17**

1.1 What is SQLfast? 17
1.2 The target audiences of SQLfast 19
1.3 Installing and starting SQLfast 20
1.4 The levels of the SQLfast interface 20
1.5 Starting SQLfast with the Basic interface 21
1.6 Selecting another level 22
1.7 Starting SQLfast with the Expert interface (Level 1) 22
1.8 The example databases: ORDERS.db and CLICOM.db 23
1.9 SQLfast resources and documentation 25
1.9.1 Integrated help 26
1.9.2 Integrated tutorials 28
1.9.3 Scripts 29
1.9.4 The SQLfast e-book tutorial 30

**Chapter 2 - Basic scripts 31**

2.1 Preparing the work 31
2.2 Writing and running an SQLfast script 32
2.3 Saving the current script 33
2.4 Loading and modifying a script 34
2.5 Other ways to execute a script 34
2.6 Representing scripts and results in this tutorial 35
2.7 How to write SQLfast statements? 36
2.8 Questions on this first script? 37
2.9 A second script, just as simple 38
2.10 Using variables - First encounter 38
2.11 Variable’s way of life 40
2.12 Interactive SQLfast scripts 41
2.13 To quote or not to quote 42
2.14 SQLfast sessions 43
2.15 Summary 44

**Chapter 3 - Creating a database 45**

3.1 How to create a new database 45
3.2 How (not) to loose data 47

3.3 Creating and loading the ORDERS database 49
3.4 Calling secondary scripts (first look) 50
3.5 The schema of the ORDERS.db database 52
3.6 A new skill: writing blank lines 53
3.7 Temporary tables 54
3.8 In memory databases 54
3.9 Upper case or lower case? 55
3.10 Summary 56

**Chapter 4 - Updating data 59**

4.1 Extending user-driven data selection 59
4.2 Data modification statements: a short reminder 61
4.3 Inserting rows into a table 63
4.4 About single quotes in SQL values 63
4.5 Updating rows of a table 64
4.6 Variables in statement customization 65
4.7 Variables in full statement specification 65
4.8 Improving data entry box layout 66
4.9 Dialog box for updating values 67
4.10 Summary 68

**Chapter 5 - SQL in a nutshell 71**

5.1 Introduction 71
5.2 Creating and modifying the schema of a database 72
5.3 Single table extraction queries 75
5.4 Subqueries 76
5.5 SQL functions 77
5.6 Aggregate (statistical) functions 79
5.7 Joining tables 79
5.8 Set operators 81
5.9 Data grouping 81
5.10 Generalization of the select and from clauses 83
5.11 Data modification 84
5.12 SQL views 85
5.13 Stored procedures and triggers 86
5.14 About SQLite 87
5.15 Further reading 88

**Chapter 6 - More on variables 91**

6.1 Time variables and statements 91
6.2 Measuring elapsed time 92
6.3 Oh! Wait a minute! 94
6.4 Variable delimiters - Reminder 94
6.5 We can change the delimiters 95
6.6 Sometimes variable substitution must be disabled 96
6.7 Beware substitution ambiguity 99
6.8 Numeric computation 100
6.9 Character string computation 101
6.10 Set or compute? 102
6.11 Multi-variables assignment 103
6.12 Simulating an array 104
6.13 Variables and SQLfast procedures 104
6.14 What is a variable value after all? 105
6.15 Summary 106

**Chapter 7 - Taking decisions 107**

7.1 Alternative execution of a script 107
7.2 Skipping statements 109
7.3 If-else-endif block structure 109
7.4 Better controlling user values 112
7.5 Conditions 113
7.6 If, goto, labels and variables 114
7.7 Complement: more on goto and label statements 115
7.7.1 Looping and counting 115
7.7.2 Looping again 116
7.7.3 Reading the buttons of the dialogue box 117
7.8 Summary 118

**Chapter 8 - Reading data 119**

8.1 Can we process individual rows? 119
8.2 Forcing a query to return one row only 120
8.3 Can we force a query to returns two rows? 121
8.4 Queries that return no row 122
8.5 Looping in the database 122
8.6 Embedded loops 124
8.7 A more elegant script 126
8.8 SQLfast arrays revisited 126

8.9 Empty sets, null values and statistics 128
8.10 Summary 130

**Chapter 9 - Looping - The final chapter 133**

9.1 Introduction 133
9.2 SQL for loop 134
9.3 Range for loop 134
9.4 File for loop 135
9.5 List for loop 136
9.6 While loop 136
9.7 Forced loop iteration 138
9.8 Forced loop exit 138
9.9 While (True) loop 139
9.10 Correct loop structure 139
9.11 Variables in loops and if statements 141
9.12 How does a loop work, actually? 142
9.12.1 The while (True) loop 142
9.12.2 The while (condition) loop 142
9.12.3 The SQL for loop 143
9.12.4 The file for loop 143
9.12.5 The range for loop 144
9.13 Summary 145

**Chapter 10 - SQLfast expressions 147**

10.1 Introduction 147
10.2 Value of a variable 148
10.3 Binary values 148
10.4 Syntax of SQLfast expressions 149
10.5 Numeric functions 150
10.6 Character string functions 151
10.7 Logical functions 155
10.8 Temporal functions 156
10.9 File/web functions 159
10.10 General functions 159
10.11 User defined functions (UDF) 160
10.12 Functions from external libraries 161
10.13 Summary 162

**Chapter 11 - Output channels, files and directories 163**

11.1 The SQLfast output window: the default output channel 164
11.2 The (SQLfast) output window 165
11.3 User output files 165
11.4 The standard output file 167
11.5 Output variables 167
11.6 Closing an output channel 168
11.7 Setting the default output channel and output mode 168
11.8 Clearing the output window 168
11.9 write or print? 169
11.10 Example: playing with output channels 169
11.11 Managing files and directories 170
11.11.1 About file and directory names 170
11.11.2 Selecting a file or a directory 172
11.11.3 Examining the content of a directory 172
11.11.4 Examining the properties of a file or a directory 173
11.11.5 Renaming, copying and deleting files 174
11.12 Summary 174

**Chapter 12 - Dialogues 177**

12.1 The SQLfast GUI 177
12.2 Standard data entry box 178
12.3 Controlling user action 181
12.4 Update a data set 181
12.5 Text entry box 182
12.6 Selecting one item among several 186
12.7 Selecting items among several 188
12.8 List boxes 190
12.9 Summary 191

**Chapter 13 - Displaying data 193**

13.1 Information message box 193
13.2 Displaying a list of values on the screen 195
13.3 Displaying a text 195
13.4 Displaying a picture 196
13.5 Displaying drawings 199
13.6 Won't you please, please help me 201
13.7 Summary 202

**Chapter 14 - Advanced dialogues 205**

14.1 Multicolumn data entry and selection 205
14.2 Logical field grouping 207
14.3 Logical button grouping 208
14.4 No label please! 210
14.5 Application: building a 2D menu 212
14.6 Data entry through selection in a predefined value list 213
14.7 Predefined values from the database 215
14.8 Displayed vs returned values 217
14.9 Customizing standard buttons 219
14.10 Composite dialog boxes 220
14.11 Heterogeneous composite boxes 224
14.12 Of styles and colors 227
14.13 Example of complex composite box 228
14.14 Summary 228

**Chapter 15 - Large data objects 233**

15.1 Beyond elementary data items 233
15.2 Displaying large character strings 234
15.3 Reading and writing (large) text files 235
15.4 Reading and writing large binary files 236
15.5 Storing large objects in a database 237
15.6 About binary values 239
15.6.1 Text files: from cp850 to UTF-8 240
15.6.2 Binary files 242
15.6.3 Displaying extracted binary data 242
15.7 Binary data encoding/decoding 244
15.7.1 Hexadecimal encoding/decoding 244
15.7.2 base64 encoding/decoding 245
15.7.3 Secure hashing 247
15.8 A touch of cryptography 248
15.9 Where to store large objects 251
15.10 Loading large volumes of data 251
15.11 Summary 253

**Chapter 16 - Writing data tables - Part 1 255**

16.1 An in-depth look at output tables 255
16.2 Changing the layout of an output table 258
16.3 CSV output format 259

16.4 Key-Value output format 263
16.5 List output format 265
16.6 Text output format 265
16.7 Summary 266

**Chapter 17 - Writing data tables - Part 2 267**

17.1 XML output format 267
17.2 HTML output format 270
17.3 Tuple output format 272
17.4 RTF output format 273
17.5 SQL output format 278
17.6 LateX output format 279
17.7 JSON output format 280

**Chapter 18 - Importing and exporting data 283**

18.1 External data import 283
18.2 CSV data import / export 284
18.2.1 CSV syntax rules 285
18.2.2 Exporting data in csv format 286
18.2.3 Refining export function 287
18.2.4 Values including double-quotes 287
18.2.5 Exporting binary values 287
18.2.6 Importing rows from a csv file 288
18.2.7 Example: save and restore relational data 291
18.2.8 Estimating column types through data analysis 291
18.3 dBase data import 293
18.3.1 DBF file format 294
18.3.2 Importing rows from a dbf file 294
18.3.3 Example: migrating a dBase database 296
18.4 SQL-DML data import/export 296
18.5 JSON data export/import 296
18.6 Summary 296

**Chapter 19 - Recursive programming 297**

19.1 Introduction 297
19.2 SQL recursive queries 299
19.2.1 Common Table Expressions (CTE) 299
19.2.2 Recursive CTE 300
19.2.3 Union all or union? 301

19.3 SQL recursive queries: the Staff application 302
19.4 Recursive queries against cyclic data 306
19.4.1 Arbitrarily limiting the number of executions 308
19.4.2 Limiting the number of executions to the size of the problem 308
19.4.3 Preventing the query from visiting a person twice 308
19.4.4 Discarding dupllcates in the union operator 309
19.5 Can recursive queries compute everything? 310
19.6 Tree traversal ordering 313
19.7 SQL recursive triggers 314
19.7.1 Factorial 315
19.7.2 The Staff application 315
19.8 Loop-based SQLfast implementation of recursive procedures 318
19.8.1 Computing the factorial of a number 318
19.8.2 The Staff application 319
19.9 SQLfast recursive scripts 321
19.9.1 Recursive programming in general 322
19.9.2 The Staff application - Version 1 323
19.9.3 Computing the factorial of a number 324
19.9.4 The Staff application - General approach 326
19.10 Comparison and conclusions 327

**Chapter 20 - Metadata - Walking on the wild side 329**

20.1 Introduction 329
20.2 The main SQLfast dictionary tables 332
20.2.1 The SYS_TABLE and SYS_COLUMN tables 332
20.2.2 The SYS_KEY and SYS_KEYCOMP tables 334
20.2.3 The SYS_INTER_TABLE view 335
20.3 The additional system tables 336
20.4 How to examine the SQLfast dictionary? 338
20.5 Dictionary or no dictionary? 339
20.6 Examining the schema of an unknown database 340
20.7 From metadata to data 343
20.8 Complement: SQL group_concat function 346
20.9 Summary 347

**Chapter 21 - SQLdraw 349**

21.1 Vector graphics in SQLfast 349
21.2 The SQLdraw scripting language 349
21.3 Displaying an SQLdraw script 351

## Chapter 22 - Extending SQLfast 353

22.1 Introduction 353
22.2 SQLfast utilities 353
22.2.1 Service scripts 354
22.2.2 Functional scripts 354
22.3 Python external applications and libraries 354
22.3.1 External SQLfast applications 354
22.3.2 External SQLfast function libraries 355
22.4 User-defined functions (UDF) 355
22.5 SQLfast language extension 356
22.6 Foreign applications 356
22.7 Developing custom-made external applications and libraries 357
22.7.1 General procedure 357
22.8 Developing custom-made UDF for SQLfast and SQL 358
22.9 SQL queries in UDF 361
22.9.1 SQL statements in an SQL UDF 361
22.9.2 SQL statements in an SQLfast UDF 362
22.10 Complements: standard external SQLfast function libraries 363
22.10.1 The LStr library 363
22.10.2 The LFile library 364
22.10.3 The LWeb library 364
22.10.4 The LGeo library 365
22.10.5 Some recommandations for the development of libraries 366

## Chapter 23 - Aid to SQLfast script development 367

23.1 Introduction 368
23.2 Two SQLfast language levels 368
23.3 Examining the contents of variables and parameters 370
23.4 Enabling/disabling script output display 372
23.5 Tracing script execution 372
23.6 Tracing selected code sections 375
23.7 Tracing trigger execution 377
23.8 Analyzing and processing traces 380
23.8.1 Elementary queries 380
23.8.2 Analyzing variables and values flow 382
23.8.3 Auditing business activities 386
23.8.4 Auditing computer activity 387
23.8.5 Auditing SQL usage 387

23.9 Checking trigger circuits 388
23.10 Summary 388

**Chapter 24 - Storing transient data - The SQLfast lists 391**

24.1 Transient data: temporary tables or SQLfast lists? 392
24.2 SQLfast lists 392
24.3 Creating an SQLfast list 393
24.4 Manipulation of SQLfast lists 393
24.5 Set-theoretic manipulations of SQLfast lists 395
24.6 List-based for loop (from version 3.0) 396
24.7 Standard versions of SQLfast list functions 396
24.8 Light versions of list functions 397
24.9 Three application examples 397

**Chapter 25 - Dynamic columns and schema-less tables 403**

25.1 Introduction 404
25.2 Dynamic columns 404
25.3 Practical syntax 405
25.4 Metadata of dynamic columns 406
25.5 Technical note 408
25.6 Performance: static vs dynamic columns 409
25.7 Dynamic columns in DBMS 410
25.8 Summary 413

**Chapter 26 - Developing a tutorial 415**

26.1 Introduction 415
26.2 Opening a tutorial 416
26.3 A tutorial is a formatted text 416
26.4 Adding images 420
26.5 Navigating within and between tutorials 420
26.6 Code snippets 422
26.7 Embedded scripts 423
26.8 Embedded script variants 425
26.9 Embedded scripts as a programming system 427
26.9.1 Calling an embedded script 427
26.9.2 Minimizing script internals 429
26.10 Inserting video and audio messages 430
26.11 Embedded wrappers 431
26.12 What is the role of a tutorial, after all? 436

26.12.1 The tutorial as ... a tutorial! 436
26.12.2 The tutorial as a case study narrative 438
26.12.3 The tutorial as a technical documentation 439
26.12.4 The tutorial as a standalone application 441
26.13 Color settings 441
26.14 Summary 441
26.15 Technical complement: about wrappers 446

**Chapter 27 - A basket of SQLfast examples 449**

27.1 Introduction 452
27.2 Consulting customer data 453
27.3 Interactive data loading 454
27.4 Set and multiset operators 455
27.5 Displaying the content of two tables in a single table 460
27.6 Dumping a database in XML format (version 1) 463
27.7 Dumping a database in XML format (version 2) 465
27.8 Data-driven multiple aggregations (table pivoting) 468
27.9 Paged reading of data 472
27.10 Raising temperature 475
27.11 Dynamic branching 477
27.12 Simulating arrays in SQLfast 478
27.13 Smart writing of a data table 481
27.14 Checking functional dependencies in a table 484
27.15 Storing photos in a database 487
27.16 A photo album 490
27.17 Collecting web statistics 497
27.18 Computing missing data in incomplete statistical series 499
27.19 The Travelling Salesman problem 505
27.20 Numbering the rows of a table 510
27.20.1 First technique: applying a mathematical definition 510
27.20.2 Second technique: adding the source rowids 511
27.20.3 Third technique: adding the target rowids 511
27.20.4 Fourth technique: adding an auto incremente column 511
27.20.5 Fifth technique: recursive CTE 512
27.20.6 Sixth technique: Window function 512
27.20.7 Seventh technique: SQLfast list 513
27.20.8 Comparing the numbering techniques 513
27.20.9 Pairing two tables 514

27.21 Counting letters and words 515
27.22 Creating empty arrays 523
27.22.1 Full arrays 523
27.23 Computing the powerset of a set 526
27.23.1 First technique 526
27.23.2 Second technique 527
27.23.3 Performances 528
27.23.4 Storing the powerset members in a table 528
27.24 The relational division 530
27.25 Checking set inclusion 533
27.25.1 The inclusion problem 533
27.25.2 The algorithms 534
27.25.3 Performance evaluation 536
27.25.4 The set equality problem 536
27.25.5 Transitive reduction 537
27.25.6 Transitive reduction with equality relationships 538
27.26 Solving logic puzzles 544
27.26.1 The dwellers 544
27.26.2 The eight queens problem 546
27.27 Nest and unnest operators 554
27.27.1 1NF and N1NF tables 554
27.27.2 The unnest operator 554
27.27.3 The nest operator 556
27.27.4 Are the nest and unnest really reversible? 556

**Volume index 423**

# Chapter 1

# Introduction

**Objective**: This introductory chapter describes briefly the SQLfast language and environment, in particular, their objectives and the intended audiences. It also shows how to start the SQLfast graphical interfaces.The last sections show and describe the schema and the contents of the example database that will be used in several chapters of this tutorial.

**Keywords**: SQLfast language, SQLfast environment, SQLfast levels, non expert user, script, help, tutorials

**Statements and functions**: openDB, select, closeDB

**System variables and parameters**: -

## 1.1 What is SQLfast?

SQLfast is a *language* and a *software environment* that allow novice or casual users

- to create, populate, examine, modify and query relational databases through a user-friendly graphical interface,
- to write and execute small scripts to create, modify, consult, manipulate and process relational databases.

Beyond learning databases, SQL and database programming, they also allow more knowledgeable users to rapidly develop and execute data intensive programs of any size and complexity.

The core of the SQLfast language is **SQL**, the universal language for database definition and manipulation. Any sequence of SQL statements, whatever its length, is a valid SQLfast script. SQLfast also comprises statements that provide for graphical interaction with users, text generation, file manipulation, decision and iteration, and much more.

The advanced features of SQLfast are intended for the **rapid and easy development** of **small database applications** by **non-professionals**. Of course, it does not compete with standard programming languages, such as C, Java or C# with which large and complex database applications are written. But, by automating most tedious and obscure parts of database programming and user interaction, it allows script writers to concentrate on the problem logic.

For instance, extracting and formatting data from the database requires about 40 Java/JDBC statements but only one SQLfast statement. A 3-field data entry dialogue box is built with more than 100 Python/tkinter statements but requires one SQLfast statement only. Similarly, downloading a file from the web needs one SQLfast statement, against more than 100 Python/urllib2 statements. A document (**Code_Comparison_EN.pdf** or its French version **Code_Comparison _FR.pdf**), available on the SQLfast website (**https://bit.ly/3LfUa8z**), compares the SQLfast code for these three examples with their equivalent standard code.

Despite its simplicity, the SQLfast language includes **powerful features** such as the full SQL language, script variables, GUI, comprehensive control structures (procedures, recursive calls, if-else, for-endfor, while-endwhile, exit, next, continue, stop, pause, etc.), a powerful text generator coupled with the *select-from-where* query, dynamic statement and script execution, external program interface, transactions, metadata management, vector graphics, etc.

One of the distinctive components of SQLfast is its **tutorial engine**. A *tutorial* (or *help* document) is a hypertext written in the **SQLtuto** language (think of it as a micro-HTML). This document is active, in that, any code fragment (SQL queries, SQLfast scripts, GUI, procedure call) can be **executed** from within the tutorial in which it appears. Developing complex, illustrated and active course systems is therefore quite easy. SQLast already includes a portfolio of several dozens of tutorials devoted to SQLfast use and SQL training.

Finally, an important feature of the SQLfast environment is its **extensibility**: new functions, new resources, and even new statements can be developed quite easily.

SQLfast has been developed in Python and compiled for MS Windows platforms. Its graphical interface strictly uses the standard *tkinter* graphic API and the underlying RDBMS is SQLite. SQLfast requires no installation and can be used free of charge.

### Warning

As is (unfortunately) common in software development, the documentation is not always synchronized with the current state of the software. This is true for SQLfast. Maintaining its huge documentation (text books, technical, tutorials, manuals, case studies, etc.) both in English and in French, in addition to the maintenance and evolution of the software is particularly time consuming. As a consequence, the information of this book may sometimes be different from the functions of the last SQLfast version.

## 1.2 The target audiences of SQLfast

SQLfast is primarily intended for people who need to collect, process, analyze and visualize (possibly large) collections of complex data but who have little time either to develop data processing programs or to learn and master databases processing and programming.

- For ***casual, non expert, users***, SQLfast provides an easy way to write small ad hoc database processing tools customized to their exact needs, without resorting to complex programming or using powerful but complex software like SAS and SPSS.
- For ***non expert but motivated users***, SQLfast is quite appropriate for the development of small and simple applications for, e.g., library management, statistical analysis, simple data mining, geographic information systems, photo album, text analysis, graphical display of data, and so on.[1]
- For ***expert users***, SQLfast is a fine environment for fast database application prototyping. For example, a simple but operational library management system with its own GUI can be written in no more than 70 statements, while, based on users feedback, the full-fledged system has been developed in about 1000 statements. In addition, the metadata subsystem and dynamic programming features of SQLfast (a script can create then execute scripts dynamically) provide an ideal environment for *schema-driven* programming.
- For ***students***, since the SQLfast language basically is SQL, and thanks to its user-friendly graphical interface, it is also an ideal support to practical work in database introductory and advanced courses. It allows students to play with SQL in a secure environment, but also to develop in a few lines powerful scripts without the burden of learning and using standard programming languages. An SQL tutor has been developed (in SQLfast of course!) which lets students practice the SQL language through a series of exercises then automatically evaluate their answers (see SQLtutor-STUDENT and SQLtutor-INSTRUCTOR applications).

1. Several of them are studied in SQLfast case studies (**https://bit.ly/3x0fj2i**) .

It must be clear that SQLfast is not designed for the development by professional programmers of large and complex multi-user programs for high-performance transactional processing of shared databases.

### Note: Script *versus* Program

The term *script* generally refers to small programs written in some simple high-level interpreted, dynamic, programming language. This does not prevent some so-called scripting languages from providing power and functionalities close to those of standard programming languages and to allow the development of large and complex programs. In this sense, we will call SQLfast a *scripting language* and the programs developed in SQLfast can be called *scripts*. In this book, we will use the terms *script*, *program* and *procedure* depending on the context.

## 1.3 Installing and starting SQLfast

SQLfast is a portable software that does not need to be installed but merely extracted from a *zip* or *rar* archive. That archive includes a folder, **SQLfast**, which must be copied on any support: internal HD, external HD, USB flash drive. Once this folder has been copied, SQLfast is ready to run.

To start SQLfast, just double-click on executable **SQLfast.exe** or on a shortcut of that executable.

More detail can be found in the document **SQLfast_Installation_EN.pdf** (or its French version **SQLfast_Installation_FR.pdf**), available on the SQLfast website.

## 1.4 The levels of the SQLfast interface

SQLfast provides two levels of usage depending on the skill and task requirements of the users. A specific graphical interface is associated with each of them.

- **Basic Level**: intended for novice users, appropriate to explore databases, to solve small problems and to learn databases and SQL. This is the default level of the standard distribution.
- **Expert Level**: intended for more skilled users, appropriate to learn database programming and to solve more complex problems.

Let us first describe the *Basic Level* of SQLfast.

## 1.5 Starting SQLfast with the *Basic* interface

We double-click on **SQLfast.exe** in folder **SQLfast** or on its shortcut. The two standard windows of the graphical interface open (Figure 1.1):

– **SQLfast main window**: This window allows us to execute simple operations: *create*, *open*, *restore* and *close* a database, *run* a query or a script, *display* the schema and the contents of a database, etc. It also comprises a text area, called *script area*, in which we write SQL queries and SQLfast scripts.

– **SQLfast output window**: The result of SQL queries is displayed in this window. We will also find in the text area various messages and information on the way queries have been executed.

Figure 1.1 shows the state of the interface when the query entered in the script area has been executed against database **ORDERS.db** through the following steps:

1. database **ORDERS.db** has been opened through button **Open DB**,
2. the SQL query has been typed in the script area
3. and executed through button **Run**.

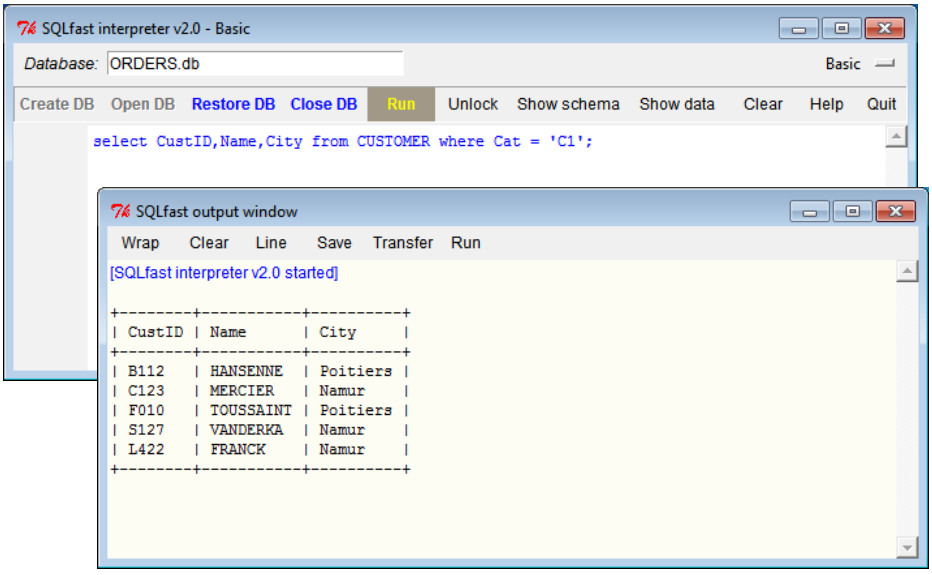



**Figure 1.1 -** The standard windows of *Basic* SQLfast interface (the look of the current SQLfast version may vary)

Since this execution was successful, the result appears in the output window and the query is colored in blue.

This interface is particular simple to use. To master it, we suggest the reading of the two help files integrated to SQLfast. They can be opened through button **Help**.

- The first one, ***Getting started***, introduces the user to the components and functions of the *Basic* interface.
- The second one, ***SQLfast environment***, provides more detail on this interface.

Then, to get acquainted with the concepts of databases and with the basics of the SQL language, we suggest practicing the tutorials that can be accessed through the **Help** button.

## 1.6 Selecting another level

The current level is indicated by the label at the top right corner of the main window. To select another level, click on that label to open the list of levels available, as shown in Figure 1.2, then click on the desired level.[2]

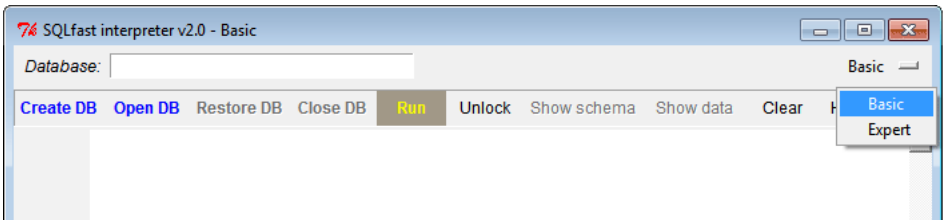



**Figure 1.2 -** Selecting an interface level

## 1.7 Starting SQLfast with the *Expert* interface (*Level 1*)

The *Expert* level is the interface we will use in this tutorial. Its main window (Figure 1.3) includes, among others, various functions to create, manage and execute SQLfast scripts.

Figure 1.3 shows the state of the main window when a short script has been introduced. This script comprises three statements:

2. In more recent versions, the level is selected through two different buttons.

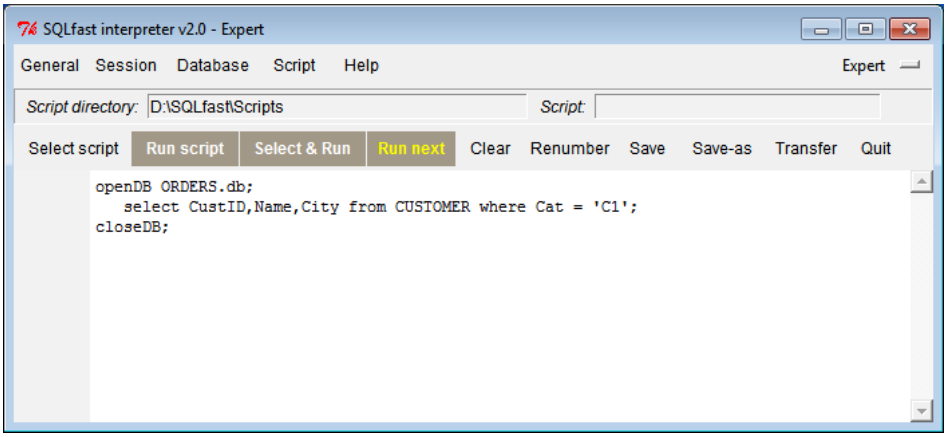



**Figure 1.3 -** The main window of the *Expert* interface

- `openDB ORDERS.db`: a database is opened with a statement instead of with a button as in the Basic level,
- `select CustID,Name,City ...`: same query as in the Basic level example,
- `closeDB`: closing the database (this statement is optional).

The functions of the Expert interface are described in two documents, **Getting started** and **SQLfast environment** that can be opened as follows:

menu **Help** > **Getting started**

menu **Help** > **SQLfast references** > **SQLfast environment**

The reader is expected to have read these documents before studying Chapter 2.

## 1.8 The example databases: ORDERS.db and CLICOM.db

Many examples developed and discussed in this book are based on a reference database describing *customers*, the *products* they can order and the *orders* they have placed. Though seemingly very simple, it is sufficiently rich to illustrate the most sophisticated SQL queries and applications.

This database is available in system folder **SQLfast/Databases** in two equivalent variants: **ORDERS.db** in English and **CLICOM.db** in French. **ORDERS.db** is made up of four tables: **CUSTOMER**, **PRODUCT**, **CUSTORDER** and **DETAIL** (CLICOM.db comprises tables CLIENT, PRODUIT, COMMANDE and DETAIL).

– Table **CUSTOMER** contains data on customers. It comprises six columns:

- **CustID**: the unique ID of the customer,
- **Name**: the name of the customer,
- **Address**: the address of the customer,
- **City**: the city of the customer,
- **Cat**: a 2 character code specifying the category of the customer; this column is optional (*nullable*),
- **Account**: the amount available in the customer account.

Its primary key is {**CustID**}.

– Table **PRODUCT** contains data on the products customers can order. It comprises four columns:

- **ProdID**: the unique ID of the product,
- **Description**: a short textual description of the product,
- **Price**: the unit price of the product,
- **QonHand**: the quantity on hand (i.e., the remaining quantity) of the product.

Its primary key is {**ProdID**}.

– Table **CUSTORDER** records data on orders placed by customers. It comprises three columns:

- **OrdID**: the unique ID of the order,
- **CustID**: the ID of the customer who placed the order,
- **DateOrd**: the date on which the order was placed.

Its primary key is {**OrdID**}. It has a foreign key:

{**CustID**} that references table **CUSTOMER**.

– Table **DETAIL** specifies the products referenced by the orders and their quantities. It comprises three columns:

- **OrdID**: the ID of the order,
- **ProdID**: the ID of the product referenced by the order,
- **Qord**: the quantity of product ordered by the order.

Its primary key is {**OrdID,ProdID**}.

It has two foreign keys:

- {**OrdID**} that references table **CUSTORDER**,
- {**ProdID**} that references table **PRODUCT**.

SQLfast can display the schema of a database in an independent window, either as a *summary* (Figure 1.4) or as the full SQL-DDL code. This window is opened

- in the *Basic* interface through button **Show schema**,
- in the *Expert* interface, through menu **Database** > **Show schema**.

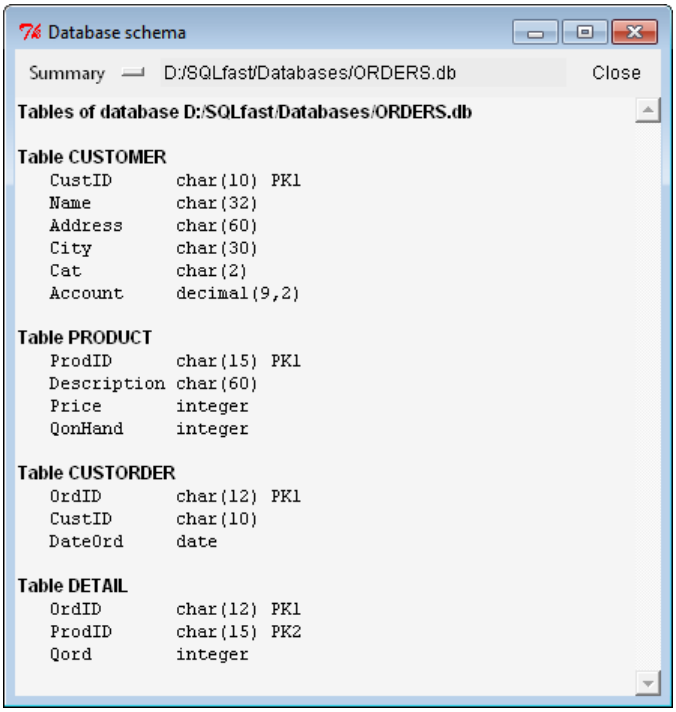



**Figure 1.4 -** The schema of database ORDERS.db (Summary version) (the look of the current SQLfast version may vary)

The content of the tables can be displayed in a series of independent windows that are opened,

- in the *Basic* Level, through button **Show data**,
- in the *Expert* Level, through menu **Database** > **Show data**.

The initial content of these tables is shown in Figure 1.5.

## 1.9 SQLfast resources and documentation

The SQLfast environment provides resources of interest depending on the initial skills and the objectives of the reader. Among them, five kinds of documents that may be used at different steps of its usage.

### 1.9.1 Integrated help

Help documents are permanently available when SQLfast is running. They provide information on the functions and features of SQLfast environment and languages. In the current version, the following documents are available:

**Basic interface** (Level 0), through button **Help**:

- **Getting started**: how to start using SQLfast. This document is intended for novice readers.
- **SQLfast environment**: description of the functions and features of the Basic interface of the SQLfast environment.

**Expert interface** (Level 1), through menu **Help**:

- **Getting started**: how to start using SQLfast environment to write, execute and manage scripts.
- **Survival guide**: a collection of about fifty representative code fragments that illustrate the use of the SQLfast language to solve a wide range of problems. These fragments are documented and, for most of them, executable. They can be used as models in more complex SQLfast scripts. This guide is available in English and in French.
- **SQLfast references**
  - **SQLfast environment**: description of the functions and features of the Expert interface of the SQLfast environment
  - **SQLfast commands**: summary of the statements and functions of the SQLfast language, of the system parameters and of the other ancillary languages of the environments
  - **SQLtuto language**: short description of the SQLfast tutorial language.

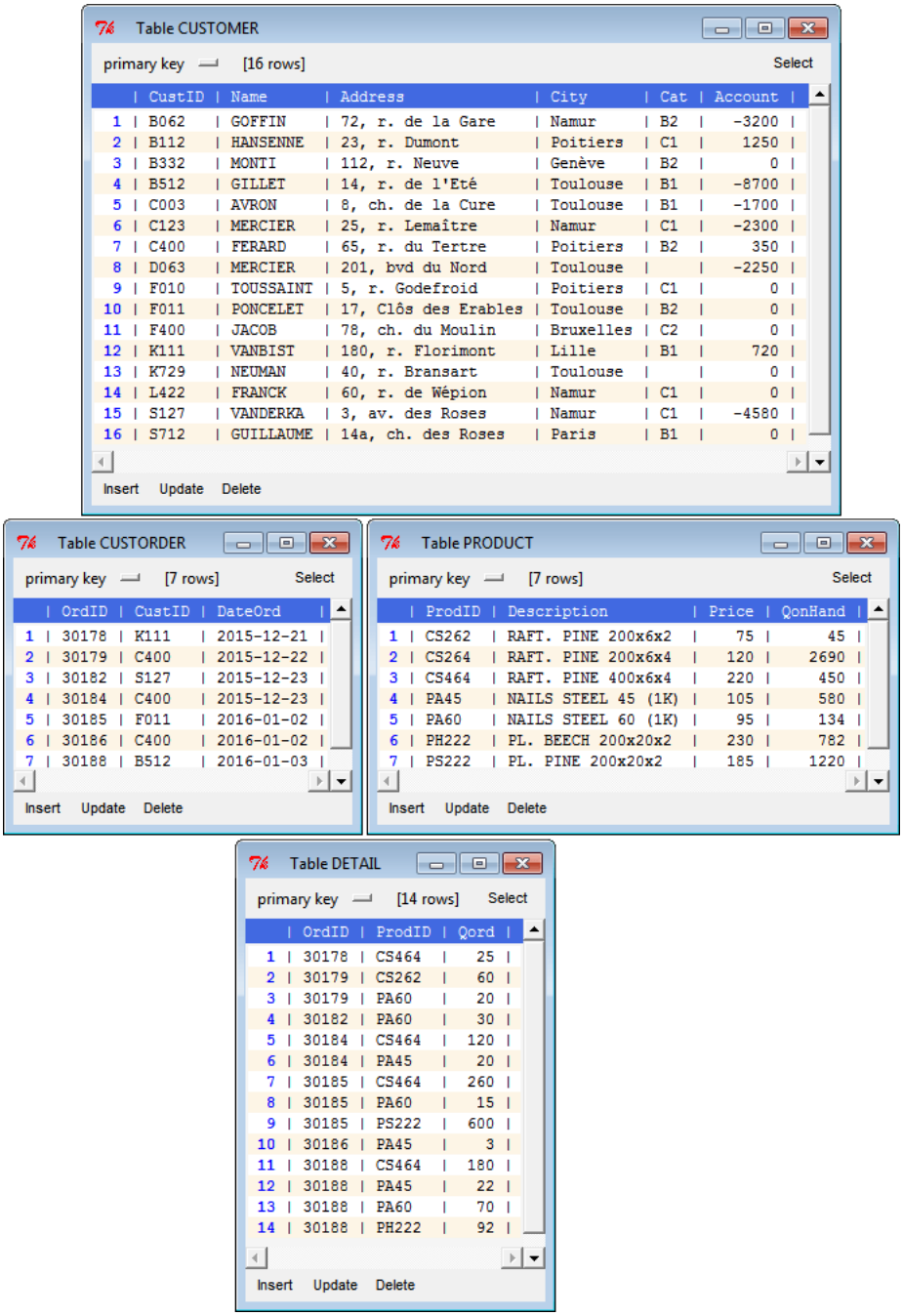



**Figure 1.5 -** The initial contents of database **ORDERS.db** (the look of the current SQLfast version may vary)

These documents are located in the SQLfast help directory, named **SQLfastHelp_FR** for French readers and **SQLfastHelp_EN** for English readers. They are written in **SQLtuto**, the SQLfast tutorial language. They can be modified by users but they cannot be deleted nor added. Note that, for the moment, not all documents are available in both languages.

To change the default directory, either use the *welcome screen* (aka Help > About SQLfast screen) or change the value of parameter **helpDirectory** in initialization file **SQLfast.ini**:

```
helpDirectory = SQLfastHelp_FR

helpDirectory = SQLfastHelp_EN
```

### 1.9.2 Integrated tutorials

Tutorials are documents typically written by teachers and used by students to help them learn database concepts and usage. A first set of tutorials is integrated into the SQLfast distribution for the *Basic* and *Expert* Levels.

**Basic Level tutorials**, through button **Help > Tutorials**:

The tutorials are devoted to the SQL language, from elementary queries to more advanced functions and applications.

They are derived from the reference book "*Bases de données - Concepts, utilisation et développement*", 5th Edition, Dunod 2012 (in French).

Though these tutorials have been written (initially in French) as a support of this reference book, they can be used independently, provided the reader has been introduced to the SQL language by another way.

These documents are located in the SQLfast tutorial directory, named **SQLfastHelp_FR/Lev0_Tutorials** for French readers and **SQLfastHelp_EN/Lev0_Tutorials** for English readers.

**Expert Level tutorials**, through menu **Help > Tutorials**:

The tutorials are devoted to database programming based on the SQLfast language, from elementary interactive scripts to more advanced functions and applications.

They are derived from some chapters of the *SQLfast tutorial* (the book to which this chapter belongs!), available as a *pdf* e-book on the SQLfast website.

These documents are located in the SQLfast tutorial folder, named **SQLfastHelp_FR/Lev1_Tutorials** for French readers and **SQLfastHelp_EN/Lev1_Tutorials** for English readers.

Tutorials are written in SQLtuto, the SQLfast tutorial authoring language. Users can modify, delete and add tutorials. Note that, for the moment, not all of these documents are available in both languages.

### 1.9.3 Scripts

In parallel with the integrated tutorials, system directory **Scripts** of the standard distribution includes a collection of *ready to run* SQLfast scripts derived from each of these tutorials. They are collected in two subdirectories:

- **Scripts/BD-2022**: scripts associated with the *Basic* Level tutorials
- **Scripts/SQLfast-Tutorials**: scripts associated with the *Expert* Level tutorials, derived from the chapters of this *SQLfast manual*.

Executing these scripts requires the *Expert* graphical interface (the concept of script is not relevant in the *Basic* Level).

## 4. Independent text documents

The SQLfast website allows users to download introductory and technical documents in *pdf* format. Some of them are derived from integrated tutorials.

**SQLfast installation**

A 2-page description of the (fictitious!) installation procedure.

- in French: **SQLfast_Installation_FR.pdf**
- in English: **SQLfast_Installation_EN.pdf**

**Getting started at the Basic Level**

Text version of the Basic Level tutorial **Getting started**

- in French: **Getting_started_SQL_FR.pdf**
- in English: **Getting_started_SQL_EN.pdf**

**Survival guide**

Text version of tutorial **Survival guide**.

- in French: **Survival_guide_FR.pdf**
- in English: **Survival_guide_EN.pdf**

**SQLfast statements and functions**

Extended text version of tutorial **SQLfast commands**.

- in French: *not translated yet*
- in English: **SQLfast-Commands.pdf**

**Code_Comparison**

Comparison of the expression of three elementary primitives in SQLfast and in standard code.

– in French: **Code_Comparison_FR.pdf**
– in English: **Code_Comparison_EN.pdf**

### 1.9.4 The SQLfast e-book tutorial

The SQLfast tutorial (of which this document is the first chapter) comprises two parts.

The first part (called *SQLfast manual*) covers the main database programming concepts: database structure and manipulation principles, SQL, variables, graphical user interaction, programming structures (sequence, alternative, loops, procedure calling, expressions, data formatting, data presentation, large data objects, metadata, recursive programming, etc.)

The second parts (called *SQLfast Case studies*) is a collection of representative database applications, from data management (e.g., a complete management system for a school library and a bill of material system) to system applications (directory management, SQL injection, temporal databases, large test data set generation, database prototyping), to decision systems (topological sorting, formal concept analysis, graph algorithms, text analysis) and to more playful applications (an adventure game, a novel generator, Conway's game of life, maze solver, writing a successful book). They demonstrate the use of the database concepts and technologies to solve complex problems.

This material can be downloaded from the SQLfast website (**https://bit.ly/3x0fj2i**). Most tutorial chapters are already available, though sometimes in a draft version. The others chapters still are being written. The code of these chapters is available in the **Scripts/Case-Studies** folder.

# Chapter 2

# Basic scripts

**Objective**: This *getting started* chapter introduces the first SQLfast scripts. It shows how to open a database, to extract data and to write them on the output console. It then shows how to make these scripts interactive by letting the user enter variable data.

**Keywords**: SQLfast script, write/execute/save/load scripts, SQLfast variable, variable/value substitution operator, variable delimiters, quoted character strings, SQLfast session

**Statements and functions**: openDB, select, closeDB, --, set, write, ask

**System variables and parameters**: SQLfast.ini, userLevel

## 2.1 Preparing the work

In this chapter as well in the next ones, we assume that the SQLfast installation is that of the recently downloaded version. In particular, the initialization file **SQLfast.ini**, located in system folder **SQLfast**, must be the original one. This text file defines the way SQLfast behaves when started.

SQLfast has been started and set to the **Expert** level. The main window must look like that in Figure 2.1. If needed, we change the level by clicking on label **Basic** then choosing the appropriate level. To avoid this manipulation, we can change the first parameter of **SQLfast.ini** as follows:

```
userLevel = 1
```

There is a lot of other interesting parameters the value of which may be worth to be changed to adapt SQLfast to our environment, but we will leave this to further chapters.

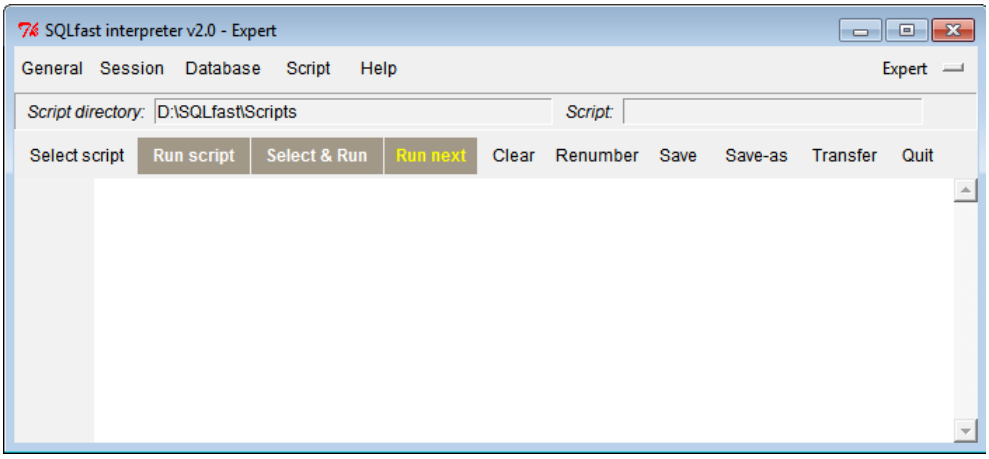



**Figure 2.1 -** Initial state of the main window

## 2.2 Writing and running an SQLfast script

Let us consider the following question: *who are the customers living in Poitiers*?

Visual inspection of the contents of table CUSTOMER in Chapter 1 shows that customers B112, F010 and C400 live in this French city. Let us verify if SQLfast agrees.

The data required can be extracted by a simple `select-from-where` SQL query:

```
select CustID,Name from CUSTOMER where City = 'Poitiers';
```

Before these data can be extracted, we must specify and open the database where the customer data are stored:

```
openDB ORDERS.db;
```

We type these two statements in the **script area** of the main window as shown in Figure 2.2. These statements form a simple **script**. Since it is visible in the script area, it is called the *current script*.

To run it, we just click on button **Run script**. If there is no typing error in this script, the result appears in the output window (Figure 2.3).

SQLfast statements are terminated by a semi-colon character. Blank lines can be inserted if we think they make the text more readable.

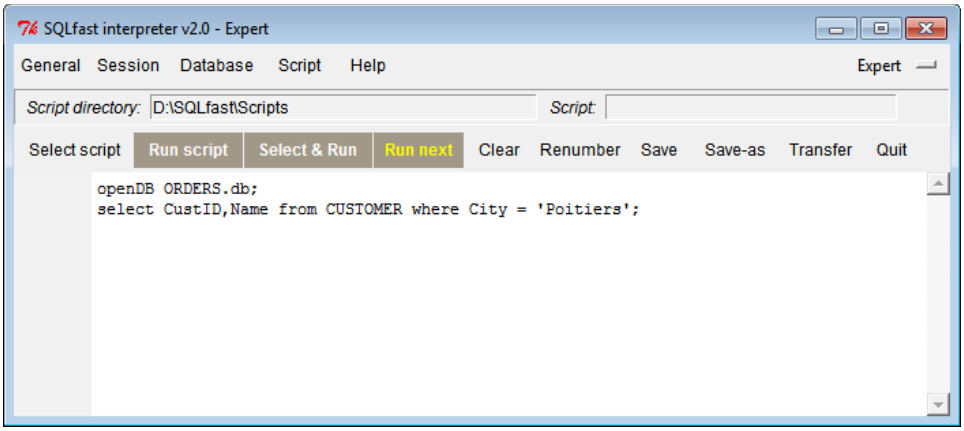



**Figure 2.2 -** Our first SQLfast script

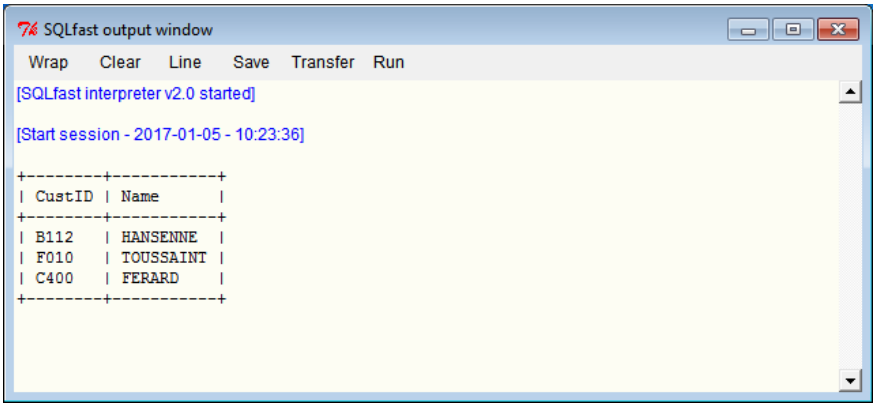



**Figure 2.3 -** The result of the execution of the current script in the output window

## 2.3 Saving the current script

If we want to keep this script for further use, we ask SQLfast to save it on disk through button **Save-as**[1]. The default folder for placing scripts is **Scripts**, in system folder **SQLfast** (Figure 2.4).

1. At this stage, button **Save** does nothing, since there is no copy of the script on disk yet.

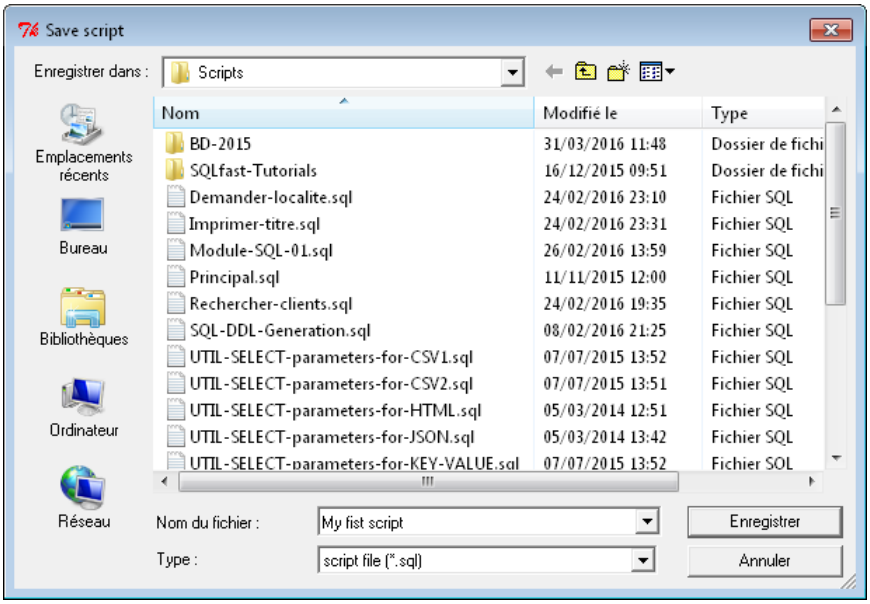



**Figure 2.4 -** Saving the current script in default folder **Scripts**

## 2.4 Loading and modifying a script

Let us empty the script area through button **Clear**. To load a saved script back into the script area, we click on button **Select script** (Figure 2.5) then we proceed as usual.

When the script has been loaded, we can execute it with button **Run script**. If we want to modify it, we can also save it on disk through button **Save** if we want it to keep its name or **Save-as** if we want to create another copy.

## 2.5 Other ways to execute a script

There are two other ways to execute a script:

- through button **Select & Run**: we select a script on disk then we execute it. The main difference with the sequence **Select script** then **Run script** if that the script is not loaded, so that the current script is left unchanged. This function will be appreciated, for instance, to execute preparatory scripts before running the current script. We will examine some examples later on.

- through button **Run next**: this is the **Run** button of the *Basic Level*. It executes all the statements of the script area that have not been executed yet (those statements written in black). Then, if things went right, it colors these statements in blue. This function will be used, for example, to build and test a script step by step. The **Unlock** operation still is available in the **Script** menu.

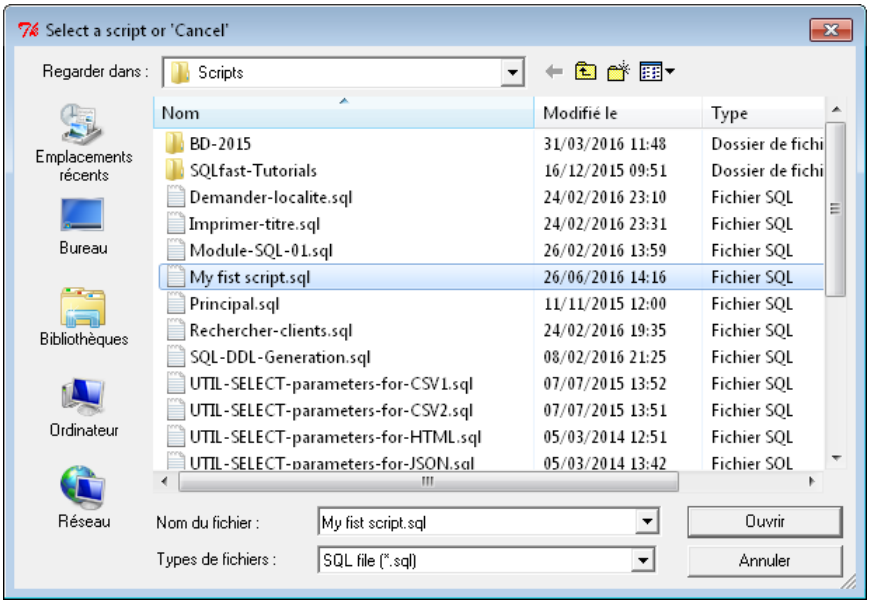



**Figure 2.5 -** Loading a script into the script area

## 2.6 Representing scripts and results in this tutorial

In this chapter and in most of the following ones, we will develop and examine dozens of scripts and results. We will adopt the following representation conventions. Scripts will be displayed on a grey background as in Script 2.1 while results as they appear in the SQLfast output window will be represented as in Figure 2.6. When the script, or an improved version thereof, exists in the directory of the chapter, its name appears between square brackets in the legend of the script [Chapter-02-01.sql].

```
openDB ORDERS.db;
select CustID,Name from CUSTOMER where City = 'Poitiers';
```

**Script 2.1 -** Listing the customers of Poitiers [Chapter-02-01.sql]

```
+--------+-----------+
| CustID | Name      |
+--------+-----------+
| B112   | HANSENNE  |
| F010   | TOUSSAINT |
| C400   | FERARD    |
+--------+-----------+
```

**Figure 2.6 -** Basic representation of results. Not particularly *glamorous*, but neat and concise!

## 2.7 How to write SQLfast statements?

The syntax of SQLfast is fairly tolerant. Blank lines can be inserted between statements, statements can be written in several lines, provided they are consecutive, and leading spaces are ignored (Script 2.2). So, long statements such as complex SQL queries and dialogue boxes can be written in a readable way.

Scripts can include **comments**, whose first two non space characters are `--` (Script 2.2). They can be inserted anywhere, even between two consecutive fragments of an SQL query. They are ignored when the script is executed.

```
-- Our first script

openDB ORDERS.db;

select CustID,Name
   from  CUSTOMER
   where City = 'Poitiers';

   -- That's all
```

**Script 2.2 -** Multiline statements, blank lines, leading spaces and comments

Let us precise the **statement formation rules**.

1. An SQLfast statement starts at the first non-space character of a line (leading spaces or tabs of a statement line are ignored).
2. It is written in one or several consecutive lines. 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 and comment lines ignored), 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 is terminated by a semicolon (`;`),
   - the next line is a blank line,

- it is the last line of the file.

So, terminating statements with a semi-colon, though not mandatory, is highly recommended.

3. If the last word of a statement is followed by one or several spaces, the latter are not thrown, but they also are part of the statement.
4. Blank lines and comment lines are ignored.

As an illustration of these rules, Script 2.2 is internally transformed into Script 2.1 before execution.

## 2.8 Questions on this first script?

Though this book primarily is **not a tutorial on SQL**,[2] we will make three comments on the statements of this first script.

1. *Why must we open the database before working on it? Let us suppose that there is only one database in directory* D:/SQLfast/Databases*!*

   No, SQLfast is not *that* smart. The database must be opened before its contents can be exploited. If we remove the statement **`openDB`** `ORDERS.db`, the script stops after displaying the SQLfast error message shown in Figure 2.7.

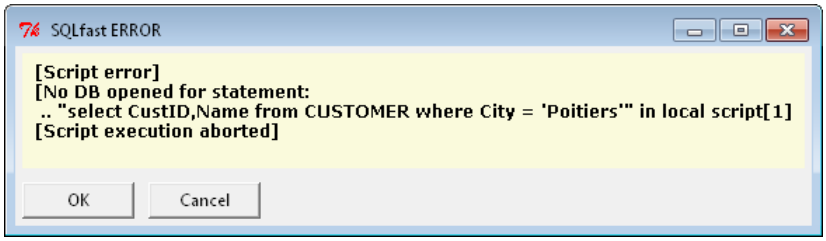



**Figure 2.7 -** Trying to extract data from a closed database!

2. *What does the SQL* **select** *statement* ***exactly*** *means?*

   I thought you were familiar with SQL! Aren't you? This statement has three parts.

2. Chapter 5 is a short reminder on relational databases and SQL. However, the best way to proceed in this case is to follow the tutorials of the *Basic* Level.

The **from** part specifies the table from which we want to extract[3] data (table CUSTOMER).

The **where** part tells from which rows we want to extract data (the rows where `City = 'Poitiers'`).

The **select** part details what these data are (the values of columns CustID and Name).

The result is written, in a tabular form, on the output window. It is just that simple! But don't forget to read Chapter 5 if you are not (yet) comfortable with SQL.

3. *We had to open the database before using it. Why don't we **close the database** after use?*

   That's right. We should have asked SQLfast to close it. Our only excuse is that SQLfast will close it for us automatically. More on this later on.

## 2.9 A second script, just as simple

Script 2.3 is an extension of the first example. It queries two categories of customers. Now, we carefully close the database once we have finished to query it (statement **closeDB**).

```
-- Examining two categories of customers
openDB ORDERS.db;
  select CustID,Name from CUSTOMER where City = 'Poitiers';
  select CustID,Name from CUSTOMER where Account > 0;
closeDB;
```

**Script 2.3 -** Examining customers [Chapter-02-02.sql]

## 2.10 Using variables - First encounter

Examining customers of, say, Genève, in Switzerland, is as easy. We just have to change the constant `'Poitiers'` into `'Genève'` in the SQL query (Script 2.4) and ask for its execution.

3. *Extracting data* from a database is a bit different from what we (or will, one day) experience at the dentist's for a similar operation. When data are *extracted*, the original values still remain in the database.

```
openDB ORDERS.db;
select CustID,Name from CUSTOMER where City = 'Genève';
```

**Script 2.4 -** Examining the customers of *Genève*

However, changing the SQL query in this way may become delicate in larger scripts. So, we introduce the concept of ***variable***.

A **variable** is a small memory in which we can store a *value*. Typically, a value will be a number, a character string (e.g., the name of a city), a date and the like. A variable need not be declared as in many languages. It is created the first time it is assigned a value[4].

In script 2.5 (from now on, for simplicity, we often omit the `openDB` and `closeDB` statements), we create a variable with name **`city`** and we assign it the value `Genève` through statement **set**:

```
set city = Genève;
```

We then replace the constant `'Genève'` in the SQL query by the variable name. However, the *name* of the variable is useless in itself! What we really need to introduce in the query is the *value* of variable **`city`** instead. To this aim, we use a special form: we surround the name of the variable with "`$`" symbols: **`$city$`**.

The **name** of the variable is `city` while its **value** is denoted by `$city$`. So, we change constant `'Genève'` to `'$city$'`.

```
set city = Genève;
select CustID,Name from CUSTOMER where City = '$city$';
```

**Script 2.5 -** Using a variable to select customers

Symbols **`$`** and **`$`** define the **substitution operator**, that replaces the name of a variable by its value. These symbols are called **variable delimiters**. We will see later that they can be changed quite easily and that the second symbol may even be dropped if desired.

When the script has been executed, we examine its result in the output window. The data of the customer(s) appear in a tabular form but without any clue on where this customer lives. Let us add a title that indicates her city (which is the value of variable `city`) before writing this table.

To write text on the console, we use statement **`write`**. Script 2.6 includes this new statement,

4. A variable has no specific type: its value will be interpreted as a number, word, a name, a date, a message, a database name or even a statement or a fragment thereof, according to the way it is used. Internally, the value of a variable is a character string, except in very special cases.

```
write Our customer(s) in $city$:;
```

that writes the city of the customers before extracting and writing their data. Here again, we use the form **$city$** to inject the value of variable **city** into the title.

```
set city = Genève;
write Our customer(s) in $city$:;
select CustID,Name from CUSTOMER where City = '$city$';
```

**Script 2.6 -** Introducing a title [Chapter-02-03.sql]

The output window now reads as follows:

```
Our customer(s) in Genève:
+--------+-------+
| CustID | Name  |
+--------+-------+
| B332   | MONTI |
+--------+-------+
```

The result is a bit crude but we will study later how to give it a more stylish look.

## 2.11 Variable's way of life

This section discusses some aspects of SQLfast variables that will mainly interest readers familiar with other programming languages.

Using SQLfast variables is very simple. However, they must be used with some care, specially when that are compared with those of other programming languages.

1. First, we observe that the text to write requires no quotes nor double-quotes as would be the case in many other languages. What follows keyword **write** is a simple character string that starts at the first non-space character (here letter **O**) and ends with the last character before terminal**;** (here, punctuation character **:**).[5]

2. The **write** statement just writes this character string in the output window. However, **before their execution**, all SQLfast statements[6] are *pre-processed* by the SQLfast engine. During this operation, all variable references (i.e., all delimited variable names) are replaced by their values. So, statement

   ```
   write Our customer(s) in $city$:;
   ```

   becomes, after substitution, executable statement:

5. Or the last non space character if there is no terminal semi-colon **;**.
6. But one: **label**. In addition, the structured control statements **if**, **for** and **while** and associated secondary statements (**then**, **endfor**, **next**, etc.) have some limitations as far as substitution is concerned.

```
write Our customer(s) in Genève:;
```

3. The way variable values are inserted in the `write` statement differs a little bit from what is usual in some other languages. If we express the write statement as follows (Java style):

   ```
   write "Our customer(s) in " + city + ":";
   ```

   or (*à la* SQL)

   ```
   write "Our customer(s) in " || city || ":";
   ```

   we just write

   ```
   "Our customer(s) in " + city + ":"
   ```

   or

   ```
   "Our customer(s) in " || city || ":"
   ```

   in the output window. Not exactly what we want!

4. If a variable cited in the statement does not exists, its name is not interpreted. However, such a situation is not considered as an error (it even will be very useful in some applications), though it may result from a typing or programming error:

   ```
   write Our customer(s) in $MyTown$:;
   ```

   just writes, because there is no variable named `MyTown`:

   ```
   Our customer(s) in $MyTown$:
   ```

## 2.12 Interactive SQLfast scripts

Successively examining the customers from several cities will quickly prove tedious. Opening the SQLfast program, identifying the place where the variable is set, changing its value, saving the program and executing it, and this, for each city, is neither user-friendly nor particularly secure. To this aim, we introduce a new statement through which the user is *asked* to enter the name of a city.

This is the role of the **`ask`** statement, which replaces the **`set`** statement in Script 2.5:

```
ask city = Enter the name of a city:;
```

When executed, **ask** opens the dialog box shown in Figure 2.8. This box comprises a prompt message (Enter the name of a city:) and an empty entry field, in which the user types the name of a city (e.g., Genève), before clicking on button OK. Now, the script is autonomous and does not require risky modification any longer.

```
ask city = Enter the name of a city:;
write Our customer(s) in $city$:;
select CustID, Name from CUSTOMER where City = '$city$';
```

**Script 2.7 -** Asking the user to enter data

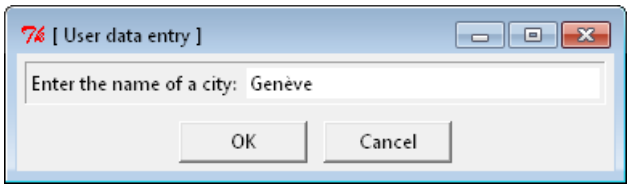



**Figure 2.8 -** The dialog box of the basic *ask* statement

The complete script, with a short comment and the database opening and closing operations, is shown in Script 2.8. It shows that variable **city** is defined once but used twice.

```
-- Examining customers through a variable (user input)

openDB ORDERS.db;
  ask city = Enter the name of a city;
  write Our customers in $city$;
  select CustID, Name from CUSTOMER where City = '$city$';
closeDB;
```

**Script 2.8 -** Customer examination - The complete script [Chapter-04.sql]

We will see later how to cope with undisciplined users who enter erroneous city name, who leave the entry field blank, who click on the Cancel button or who close the box.

### Note

**ask** is a diminutive of **askData**. So, both can be used in SQLfast scripts.

## 2.13 To *quote* or not to *quote*

As we have seen, character strings need not be surrounded by single, double or smart quotes, as is usual in many other languages. To write the name of the most famous English author in the output window, we execute the statement

```
write William Shakespeare;
```

and **not**

```
write 'William Shakespeare';
```

nor

```
write "William Shakespeare";
```

because, in the two latter cases, the quotes would be part of the line written. However, when we write SQL statements, we must obey the SQL syntax rules. To extract the data of the customer from Genève, we must write:

```
select CustID,Name from CUSTOMER where City = 'Genève';
```

where constant `Genève` must be written between single quotes, that is, `'Genève'`. Now, the question is, when we specify such constants through variables, *when* and *where* do we have to insert these quotes: in the value stored in the variable or directly in the SQL statement?

Both techniques are valid, but one and only one must be adopted[7]. Script 2.9 shows an example of each approach.

```
-- Quoted constants in SQL statements
set city = 'Genève';
select CustID, Name from CUSTOMER where City = $city$;

set city = Genève;
select CustID, Name from CUSTOMER where City = '$city$';
```

**Script 2.9 -** Two ways to quote SQL constants

## 2.14 SQLfast sessions

When the first script starts its execution, a new blue line is written in the output window:

```
[Start session - 2022-10-05 - 10:23:36]
```

It means, as we can guess, that *a new session has started*. When SQLfast is stopped, the current session closes. We can manually close the current session through menu **Scripts** > **Close session**. So, what is a session?

7. These quotes can also be entered by the user through the dialogue box but this way is not particularly elegant!

An SQLfast session is a suite of activities, such as script execution, during which some data and states are preserved: for example, the variables created are persistent and keep their values, files opened are left open, schema window and data windows are left open as well until intentionally closed.

When a session is closed, all the user variables are deleted, the current database is closed (if left open), the output files are closed, the schema window and the data windows are closed.

From then on, we can use SQLfast as if it has just been started. In particular, the next script we execute starts a new session.

## 2.15 Summary

This chapter has taught us the following facts.

- SQLfast scripts are created as pure text files with extension ***.sql**.
- SQLfast statements must be terminated by a **semi-colon** (unless ...). Missing semi-columns are a major source of script errors.
- To **execute a new script**, we must write it in the script area of the main window then click on button **Run script**.
- To **execute an existing script**, we select and load it from the **Scripts** folder through button **Select script** then we click on button **Run script**.
- To **execute an existing script** without loading it in the script are, we use button **Select & Run script**. The contents of the script area is unchanged.
- The **data written** by a script appear in the output window.
- We must open a database before using its contents (statement **openDB**). It is more polite to close the database with statement **closeDB**. However, this operation is not mandatory with the default parameter settings.
- The execution of an SQL **select** query results in the writing of the extracted data in a tabular form in the output window.
- Statement **write** displays a character string on the console.
- A **variable** is a small memory in which a value can stored.
- Statement **set** is used to store a value in a variable.
- Statement **ask** opens a dialog box comprising a label (a prompt text) and an entry field. When the user types a value in the entry field and clicks on button `OK`, this value is stored in a variable, which can be further used in the script.
- The **substitution operator** replaces the name of a variable that appears in a statement by its value. It is denoted by surrounding the variable name with symbols **$** and **$** (**$city$** denotes the value of variable **city**).

# Chapter 3

# Creating a database

**Objective**: So far, we have worked on existing database ORDERS.db. Before going back to it, we will learn how to create a new database, to declare new tables and to insert data into them.

**Keywords**: create a database, create a table, insert data, commit updates, calling a script, temporary table, in memory databases, upper/lower case

**Statements and functions**: createDB, createOrReplaceDB, create table, insert, drop table, commitDB, execSQL, fastExec, write-a, write-b

**System variables and parameters**: -

## 3.1 How to create a new database

The SQL standards do not include a recommended syntax for creating a new database. To compensate for this, SQLfast provides such a function through statement **`createDB`**. In the following example, we ask SQLfast to create a new database with name SALES.db.

```
createDB SALES.db;
```

Of course, if a database with this name already exists, this statement should fail. And, indeed, it (fortunately) fails! Which translates, first, into the polite but unambiguous message "Database SALES.db already exists", and secondly, into the sudden, premature and therefore frustrating termination of the script.

There are several means to prevent the program from behaving in such a brutal way. The simplest one consists in using the **`createOrReplaceDB`** statement, the effect of which is to create a new database if it does not exist yet, otherwise, to delete the existing one (therefore loosing its content) then to create the new one:

```
createOrReplaceDB SALES.db;
```

Once the database has been created (or *recreated*), we declare a new table CUST through the standard SQL query **`create table`** (Script 3.1). The table comprises two columns that describe customers, named CNum and Name, the type of which is `char` and that cannot be set to null (`not null`) and a third column Volume, declared of type `integer`. The first column is the *primary key* of the table (`primary key`), preventing two rows with the same value of this column to be inserted.

```
createOrReplaceDB SALES.db;
  create table CUST(CNum   char(6)  not null primary key,
                    Name   char(12) not null,
                    Volume integer  not null);
```

**Script 3.1 -** Creating a database with a first table . . .

Now, we can insert data in table CUST. Just to show (or recall) how this works, we add three SQL **`insert`** statements (Script 3.2).

```
createOrReplaceDB SALES.db;
  create table CUST(CNum   char(6)  not null primary key,
                    Name   char(12) not null,
                    Volume integer  not null);
  insert into CUST values('B112','HANSENNE',123);
  insert into CUST values('C123','MERCIER',234);
  insert into CUST values('C400','FERARD',456);
```

**Script 3.2 -** . . . then, inserting some data in this table

So, we have a new database, comprising one table, which, in turn, contains three data rows (with three column values each). To check if everything went as expected, we write the data of the new table in the output window[1] and we close the database (Script 3.3). The result is as expected:

```
+------+----------+--------+
| CNum | Name     | Volume |
+------+----------+--------+
| B112 | HANSENNE | 123    |
| C123 | MERCIER  | 234    |
| C400 | FERARD   | 456    |
+------+----------+--------+
```

```
createOrReplaceDB SALES.db;
  create table CUST(CNum   char(6)  not null primary key,
                    Name   char(12) not null,
                    Volume integer  not null);
  insert into CUST values('B112','HANSENNE',123);
  insert into CUST values('C123','MERCIER',234);
  insert into CUST values('C400','FERARD',456);
  select * from CUST;
closeDB;
```

**Script 3.3 -** Creating and checking the content of table CUST [Chapter-04-01.sql]

When adding table CUST to the current database, we must be sure that this table doesn't exist already. This is obvious in our example but we must be cautious when adding a table to an existing database. Two ways to cope with this situation[2]:

1. We first delete the table

   ```
   drop table if exists CUST;
   ```

2. We use a special variant of the **`create table`** SQL statement:

   ```
   create table if not exists CUST (CNum ... );
   ```

## 3.2 How (not) to loose data

However, to check if everything **REALLY** went as expected, we try another experiment. After the database has been created, checked and closed, we open it again and we examine the content of table `CUST` (Script 3.4).

When we examine the result in the output window, we have a **horrible surprise**: the table appears to be empty the second time we open it! As if the DBMS forgot to record them in the database!

1. In SQL statement `select * from CUST`, symbol "**`*`**" stands for *the list of all the column names of table `CUST`*.
2. There is a third technique, that we will describe later on, consisting in explicitly checking whether this table exists.

```
+------+----------+--------+
| CNum | Name     | Volume |
+------+----------+--------+
| B112 | HANSENNE | 123    |
| C123 | MERCIER  | 234    |
| C400 | FERARD   | 456    |
+------+----------+--------+

+------+------+--------+
| CNum | Name | Volume |
+------+------+--------+
+------+------+--------+
```

Actually, it didn't forget. It just decided not to record our data. Because we didn't tell it that we **really** wanted them to be recorded in the database.

```
createOrReplaceDB SALES.db;
  create table CUST(CNum   char(6)  not null primary key,
                    Name   char(12) not null,
                    Volume integer  not null);
  insert into CUST values('B112','HANSENNE',123);
  insert into CUST values('C123','MERCIER',234);
  insert into CUST values('C400','FERARD',456);
  select * from CUST;
closeDB;
-- Now, we open the database again ...
openDB SALES.db;
  select * from CUST;
closeDB;
```

**Script 3.4 -** *Double-checking* the content of table CUST: an unfortunate initiative! [Chapter-04-02.sql]

The problem is well known and is quite easy to address once we understand how the DBMS works. For security reason (our data could be used by thousands of users during a long period), the DBMS asks us to **confirm** that all the data modifications we have carried out in the script are **definitive**. Should we forget to confirm, it would cancel these modifications[3].

To soften its paranoia, we merely add a *confirmation* statement, **`commitDB`**. Hence the correct creation program of Script 3.5.

3. The rules of several popular DBMS, including SQLite, are slightly different. In particular, any DDL statement (`create table` for instance) entails an *auto-commit*. All the modifications that precede the execution of this statement are automatically confirmed. The behavior of SQLite is detailed in Appendix 3.

If we can (or most often *must*) confirm data modifications, why couldn't we **cancel** them intentionally? Yes, we can! Through the **`abortDB`** statement. Its effect is to abandon all the modifications we have carried out on the data since the last *commit*, the last *abort* or since the start of the script execution. However, it's too early to discuss its application.

Scripts that load data in actual databases can be very huge. For example, the script of a statistics database comprised 3,500,000 statements (`create table` and `insert`). It produced a 275 MB database in 7 minutes, with a decent speed of 8,333 insert operations per second[4]. Much higher speed can be obtained through **`fastExec`** statement, specially designed for bulk data loading. As usual, the price to pay is lower control on error management!

```
createOrReplaceDB SALES.db;
  create table CUST(CNum char(6)  not null primary key,
                    Name char(12) not null,
                    Volume integer not null);
  insert into CUST values('B112','HANSENNE');
  insert into CUST values('C123','MERCIER');
  insert into CUST values('C400','FERARD');
  select * from CUST;
  commitDB;
closeDB;
```

**Script 3.5 -** Finally, a fully correct creation program [Chapter-04-03.sql]

## 3.3 Creating and loading the ORDERS database

As we now know how[5] to create a database, we can go back to the ORDERS database to examine how we can create it and populate it with some representative data.

We plan to develop two scripts, one to create the tables (ORDERS-create.sql, Script 3.6) and the other one to load the data (ORDERS-insert.sql, Script 3.7).

It is nice to separate these operations, since this allows different data samples to be loaded in the same database structure, or to incrementally load data in successive batches.

4. *Technical note*. Intel i7, 2.8 GHz, 16 GB RAM, HDD 7200 rpm. No auto-commit, statement-level primary and foreign key validation enabled, 45 checkpoints through `commitDB`. Faster data loading can be obtained through special fine tuning, but this is beyond the scope of this tutorial.
5. Sorry!

## 3.4 Calling secondary scripts (first look)

Let's consider that these two scripts are most often executed in sequence, notably when we experiment various cruelties on this toy database. To simplify the reloading of the database, we intend to merge both scripts into a new script, called, say, ORDERS-create-insert.sql, that creates the database and loads the data.

Writing an ORDERS-create-insert.sql script could be done by merely copying both source scripts into the third one.

```
createOrReplaceDB ORDERS.db;
create table CUSTOMER (
        CustID      char(10)     not null,
        Name        char(32)     not null,
        Address     char(60)     not null,
        City        char(30)     not null,
        Cat         char(2),
        Account     decimal(9,2) not null,
        constraint PKCUST primary key (CustID));
create table PRODUCT (
        ProdID      char(15)     not null,
        Description char(60)     not null,
        Price       integer      not null,
        QonHand     integer      not null,
        constraint PKPROD primary key (ProdID));
create table CUSTORDER (
        OrdID       char(12)     not null,
        CustID      char(10)     not null,
        DateOrd     date         not null,
        constraint PKORD primary key (OrdID),
        constraint FKORDCUST foreign key (CustID)
           references CUSTOMER on delete no action);
create table DETAIL (
        OrdID       char(12)     not null,
        ProdID      char(15)     not null,
        Qord        integer      not null,
        constraint PKDET primary key (OrdID,ProdID),
        constraint FKDETORD foreign key (OrdID)
           references CUSTORDER on delete cascade,
        constraint FKDETPROD foreign key (ProdID)
           references PRODUCT on delete cascade);
commitDB;
closeDB;
```

**Script 3.6 -** Creation script for the ORDERS database [ORDERS-create.sql]

However, duplicating these scripts would not be particularly elegant (suppose the second one includes several thousands `insert` statements) nor secure (suppose you change the first one and forget to update the second one accordingly).

It would be better to ***reuse*** the two source scripts, not by duplicating their content but by *calling them*, that is, by asking SQLfast to *execute* them, *from within* script ORDERS-create-insert.sql. The latter is shown in Script 3.8. It includes two statements only: [6]

```
openDB ORDERS.db;
insert into CUSTOMER values
   ('B112','HANSENNE','23, r. Dumont','Poitiers','C1',1250);
insert into CUSTOMER  values
   ('C123','MERCIER','25, r. Lemaître','Namur','C1',-2300);
. . .
insert into CUSTOMER  values
   ('D063','MERCIER','201, bvd du Nord','Toulouse',null,-2250);
insert into CUSTOMER  values
   ('F400','JACOB','78, ch. du Moulin','Bruxelles','C2',0);
insert into PRODUCT values
   ('CS262','RAFT. PINE 200x6x2', 75,45);
insert into PRODUCT values
   ('CS264','RAFT. PINE 200x6x4', 120,2690);
. . .
insert into PRODUCT values
   ('PH222','PL. BEECH 200x20x2',230,782);
insert into PRODUCT values
   ('PS222','PL. PINE 200x20x2', 185,1220);
insert into CUSTORDER values ('30178','K111','2013-12-21');
insert into CUSTORDER values ('30179','C400','2013-12-22');
. . .
insert into CUSTORDER values ('30186','C400','2014-01-02');
insert into CUSTORDER values ('30188','B512','2014-01-03');
insert into DETAIL values ('30178','CS464',25);
insert into DETAIL values ('30179','PA60',20);
. . .
insert into DETAIL values ('30188','CS464',180);
insert into DETAIL values ('30188','PA45',22);
commitDB;
closeDB;
```

**Script 3.7 -** Data loading script for the ORDERS database [ORDERS-insert.sql]

6. In script ORDERS-create-insert.sql, the **`execSQL`** statements appear a bit different from what is shown here. Actually, the names of the called scripts are prefixed by path "Chapter-03/", since the default script directory has been set to D:/SQLfast. This can be changed easily for real applications.

- **execSQL** ORDERS-create.sql;
  asking for the execution of ORDERS-create.sql,
- **execSQL** ORDERS-insert.sql;
  asking for the execution of ORDERS-insert.sql.

The effect of these **execSQL** statements is simple. When the first execSQL statement is encountered, the *calling* script (here, ORDERS-create-insert.sql) suspends its execution and asks for the execution of secondary script ORDERS-create.sql (which is the *called* script). When the latter execution is completed, the calling script execution resumes.

The **execSQL** statement would deserve more development but we will leave it to a further section.

```
-- creating and loading ORDERS database
  execSQL ORDERS-create.sql;
  execSQL ORDERS-insert.sql;
```

**Script 3.8 -** A short script asking for the execution of two secondary scripts [ORDERS-create-insert.sql]

## 3.5 The schema of the ORDERS.db database

As the ORDERS.db database is used as the support of many SQLfast examples in this tutorial, it would be nice to draw its schema to avoid *decrypting* the detail of the DDL statements in Script 3.6 when we want to know the correct spelling of table and column names.

SQLfast includes a function, *Show schema*, that opens an independent window in which the main data structures of a database are shown (see Figure 1.4). In the **Expert** level, we open it through the menu:

**Databases** > **Show schema**

A database schema can also be represented graphically, as in Figure 3.1. The graphical conventions are those of the DB-MAIN database CASE tool[7]. They are fairly intuitive:

- Each table is described in a rectangular box showing its name, the names of the columns and the keys,

7. DB-MAIN is a database CASE tool that supports the creation, maintenance and evolution of databases. It can be freely downloaded from the DB-MAIN website [https://www.db-main.eu].

- optional (*nullable*) columns are indicated through cardinality constraint [0-1], stating that there is from 0 to 1 value for each CUSTOMER row; the default cardinality constraint, [1-1] (i.e., `not null`), is implicit,

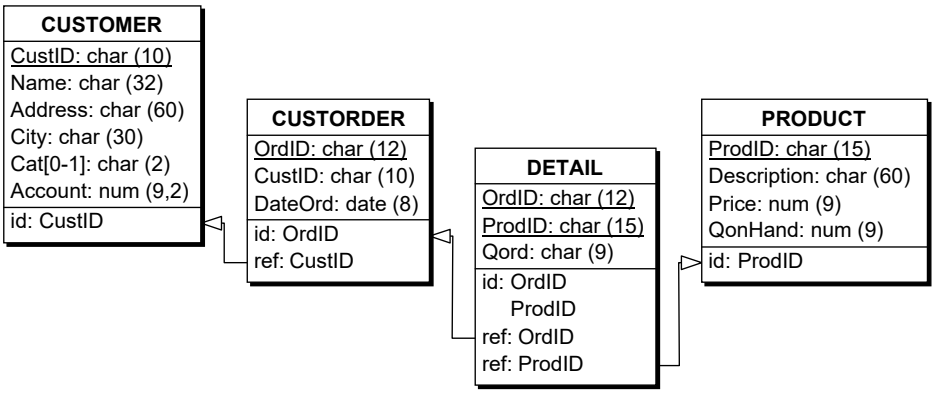



**Figure 3.1 -** Schema of the ORDERS database

- primary keys are denoted by clause **`id`**, as in "id: CustID" for table CUSTOMER (in addition, column names are underline, just to be sure!),
- foreign keys are denoted by clause **`ref`**, as in "ref: CustID" for the foreign key in CUSTORDER that references table CUSTOMER.

## 3.6 A new skill: writing blank lines

A document with the standard content of the ORDERS database can be obtained quite easily with the material we have learned so far (Script 3.9).

Statement **`write-b`** seems to be new but it is just a variant of the **`write`** statement used so far. Suffix "**`-b`**", appended to some output statements, stands for "write a blank line before", so that:

```
write-b Table CUSTOMER
```

is a handy shorthand for[8]:

```
write;
write Table CUSTOMER;
```

A blank line ***after*** would be written by suffix "**`-a`**" , which is equivalent to:

```
write Table CUSTOMER;
write;
```

8. A `write` statement without argument just writes a line *with nothing in it*, that is, a blank line.

Writing a blank line both *before* **and** *after* requires suffix "**-ba**"[9]. Therefore,

```
write-ba Table CUSTOMER;
```

is equivalent to:

```
write;
write Table CUSTOMER;
write;
```

```
-- Write the data of the ORDERS database
openDB ORDERS.db;
  write-b Table CUSTOMER;
  select * from CUSTOMER order by CustID;
  write-b Table PRODUCT;
  select * from PRODUCT order by ProdID;
  write-b Table CUSTORDER;
  select * from CUSTORDER order by OrdID;
  write-b Table DETAIL;
  select * from DETAIL order by OrdID, ProdID;
closeDB;
```

**Script 3.9 -** *Dumping* (i.e., *writing all*) the content of database ORDERS.db [script ORDERS-write-all.sql]

## 3.7 Temporary tables

If a table is just useful to store intermediate results or to load data for a one shot exploitation, it could be best to create it temporarily, as a *temporary table*:

```
create temporary table CUST(...);
```

This table will disappear when the database is closed.

## 3.8 *In memory* databases

A database is a persistent repository of data. Therefore, it is most often stored as a file or as a set of files on some external memory, such as a hard disk or a solid-state memory.

However, some applications do not require their structured data being preserved when they finish. They load data in a database, they process these data, then they

9. For the absent-minded, suffix "**-ab**" is also accepted.

discard them after having generated some results. This is quite frequent for databases that support training, prototyping or for small applications the input data of which are stored in *create and load* SQL scripts, such as those we have used in this chapter.

The **in memory databases** are particularly fitted to these applications. An *in memory* database is stored in RAM and not on a disk. Since its usage requires no disk access, its processing may be faster, specially for large databases. On the other hand, its size is limited to the RAM space available.

In SQLfast, an *in memory* database has the special name **`InMemory`**, and is created through statement **`createDB`** `InMemory;` (Script 3.10). Tables must be created (**`create table`**) and data loaded (**`insert into`**) before being processed.

When data processing is finished, the database is closed implicitly or explicitly (**`closeDB`**). There is no need to commit data modifications since the data are lost at database closing.

For obvious reasons, **`openDB`**, **`createOrReplaceDB`** and **`deleteDB`** cannot be used with an *in memory* database.

```
createDB InMemory;
   create table SAMPLE ( ... );
   insert into SAMPLE values( ... );
   select * from SAMPLE where ...;
closeDB;
```

**Script 3.10 -** Using an *in memory* database

## 3.9 Upper case or lower case?

Some parts of SQLfast statements can be written in *upper case* or in *lower case*. Here are the rules.

- The SQL keywords such as `select`, `from`, `where`, `insert`, `not null`, `primary key` can be written in upper or lower case.
- The keywords of other SQLfast statements, such as `openDB`, `commitDB`, `write` or `ask` can be written in upper or lower case.
- The names of the objects of the database schema (tables, columns) can be written in upper or lower case.
- In data values, upper case and lower case characters are different.
- In variable names, upper case and lower case characters are different.

For example:

- `openDB` can be written `opendb`, `OpenDb` or `OPENDB`
- Statement

  ```
  select CustID, Name from CUSTOMER where City = 'Genève'
  ```

  can be written

  ```
  SELECT CustID, Name FROM CUSTOMER WHERE City = 'Genève'
  ```

  and also

  ```
  select CUSTID, NAME from Customer where CITY = 'Genève'
  ```

- However, statement

  ```
  select CustID, Name from CUSTOMER where City = 'GENÈVE'
  ```

  is not equivalent
- Similarly, statements

  ```
  set city = 'Genève'
  ```

  and

  ```
  set City = 'Genève'
  ```

  define two distinct variables.

## 3.10 Summary

In this chapter we have learned the following facts.

- Statement **`createDB`** creates a new database. It fails if the database already exists.
- Statement **`createOrReplaceDB`** is used to create a new database or to replace an existing database.
- SQL query **`create table`** defines a new table. Variant **`create table if not exists`** copes with situations in which we are not sure the table does not exist yet.
- SQL query **`insert`** stores a new **`row`** in a table. Integrity constraints `not null`, `primary key` and `foreign key` must be satisfied by the new row, otherwise the insertion will be rejected.
- To avoid the loss of data modifications, statement **`commitDB`** must be executed before closing the database.
- A script can call for the execution of another script, through statement **`execSQL`**.
- It may be handy to split a long script into homogeneous parts, each stored in an autonomous script. Such is the case for a *create and load* script.

- Tables usually are permanent in a database. However, a table declared **temporary** disappears with its content when the database is closed.
- Normal databases are stored on a permanent support, such as a hard disk. Temporary databases can be stored in RAM. They are created by statement **`createDB InMemory`**. They are lost when they are closed.
- SQLfast keywords and schema object names can be written in upper case or in lower case. However, data values and variable names are *case sensitive*.

# Chapter 4

# Updating data

**Objective**: Variables have been used to let users customize (specialists would say *parametrize*) queries, notably by inserting values in the **`where`** part of the SQL **`select`** query. In this section we learn other useful ways to exploit the power of variables, notably to update a database.

**Keywords**: data modification, quoted values, substitution operator, dynamic statements, dialog box layout

**Statements and functions**: ask, ask-u, insert, update, delete, §

**System variables and parameters**: -

## 4.1 Extending user-driven data selection

In Script 2.7, the user specified the city of the customers about which data are to be extracted. Script 4.1 extends this operation by selecting customers who *live in a given city* (variable `city`) **and** who *are of a definite category* (variable `cat`).

The dialog box for user data entry (Figure 4.1) now has two data fields with two labels (`Name of the city` and `Category`) that are specified at the right of the **`ask`** statement, separated by symbol `|`.

This symbol is not a mere separator. It also tells the SQLfast engine where to place the second field: *just **below** the previous one*.

```
ask city,cat = Name of the city | Category;
select CustID,Name from CUSTOMER
where  City = '$city$' and Cat = '$cat$';
```

**Script 4.1 -** Asking users to enter two values to select customers [Chapter-05-01.sql]

When the box is closed through button **OK**, the `select` query extracts the two selected `CUSTOMER` rows (`B112` and `F010`) and writes them in the output window.

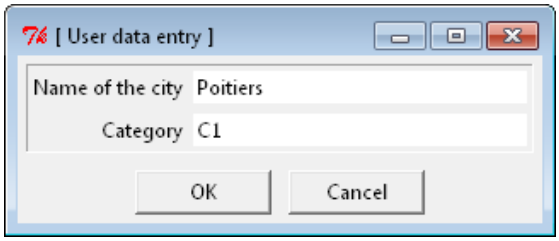



**Figure 4.1 -** The 2-field dialog box for customer *city* and *category* data entry

Let us try two more experiments:

1. We execute this script once again, but, instead of typing values in the fields, we leave them blank and we click on button **Cancel**. Surprise: *the same two rows are displayed in the output window*!
2. Same manipulation, but we click on button **OK** instead. Now an *empty result appears in the output window*.

This surprising behavior is quite natural once we understand how variables and data entry dialog boxes work. Let us first remember that a variable is created the first time it is assigned a value, e.g., through a **`set`** or an **`ask`** statement.

- A question immediately arises: *when does a variable disappear*? Simple: **never**, or at least not before the current session is closed or SQLfast is stopped. In other words, a variable and its value survive the script that created them. When we start another script, it can access the variable created before it. So, the second time we start script ORDERS-05-01.sql, variables **`city`** and **`cat`** still exist with their former values **`Poitiers`** and **`C1`**.
- When the dialog box opens, we have two actions to perform: (1) we enter values in the entry fields or we leave them blank, and (2) we click of button **OK** or on button **Cancel**.

– If we click on **OK**, the values of the entry fields *are assigned to the variables*, be they valid, invalid or blank. The result returned by the **`select`** query depends on these values (blank means *no row selected*).
– If we click on **Cancel**, the *variables are left unchanged*, whatever the values of the entry fields. The result of the **`select`** query depends on the values of the variables *before* the opening of the dialog box.

We can now admit that the results of the two experiments are quite normal!

So far, we have used user data entry to specify constants that are injected into **`select`** queries. We will study other usages of data entry and variable insertion in SQLfast statements.

## 4.2 Data modification statements: a short reminder

SQL provides three families of queries to **modify** the content of a database: **`insert`**, **`update`** and **`delete`**. We briefly recall how they work through some representative examples before studying how to write SQLfast data modification scripts.

### *a) Inserting new rows*

The simplest form of the **`insert`** query has been used in Section 3.3. In fact, the `insert` query comes in two variants, that are illustrated in Script 4.2 (the names of the target columns are optional as shown in the first two statements):

- inserting a *single row* of values specified by constants
- inserting a *set of rows* extracted from other tables of the database (table CUS is supposed to be a temporary table created in database ORDERS.db).

```
insert into CUS(CNum,Name,Volume)
  values('B195','SMITH',159);
insert into CUS
  values('B195','SMITH',159);
insert into CUST(CNum,Name,Volume)
  select CustID,Name,Account from CUSTOMER where CAT = 'C2';
```

**Script 4.2 -** The two variants of the **insert** SQL query

### *b) Modifying column values of existing rows*

The **`update`** SQL query applies a list of column modifications to a set of selected rows in a table. Each modification mentions the name of a column (left side) and an expression[1] defining the new value (right side).

```
update CUSTOMER
set    Account = Account + 500,
       CAT = 'C2'
where Account > 100;
```

**Script 4.3 -** Updating a set of rows

### c) *Deleting existing rows*

The **delete** query erases from a table the set of rows that satisfy a selection condition (Script 4.4).

```
delete from CUSTOMER where CustID = 'C400';
delete from DETAIL where ProdID = 'PA45';
```

**Script 4.4 -** Deleting a set of rows

These queries seem fairly intuitive but some caution will be required when using them in context. Indeed, data are submitted to constraints that ensure their integrity (hence the name *integrity constraint*). Modifying the content of the database may lead to incorrect data. One of the most important role of the DBMS is to enforce these constraints in order to prevent modification queries from corrupting the data. Three constraints are of particular importance:

- **not null** *columns*: in `insert` and `update` queries these columns must be assigned non null values.
- **primary key** *and* **unique** *constraint*[2]: these constraints ensure the uniqueness of values; they could be violated by `insert` and `update` queries attempting to introduce duplicate values.
- **foreign key**: for any row in the referencing table (DETAIL for instance), the value of the foreign key (ProdID) must designate an existing row in the referenced table (PRODUCT). Several operations could violate this constraint: inserting a DETAIL row, modifying column ProdID of DETAIL rows, modifying column ProdID of a PRODUCT row and deleting PRODUCT rows.

Any attempt to violate one of these rules will entail the rejection of the operation or compensation actions (e.g., deleting the DETAIL rows depending on the deleted PRODUCT row). We will see later how to cope with constraint violation. For now, in

1. This expression can be, or include, a `select-from-where` query that returns a single value, that is, one row of one column.
2. The `unique` constraint has the same effect as a primary key. The declaration `unique(Name,Address,City)` on table CUSTOMER would force any two rows to have distinct values for at least one of these columns.

case of violation, we have to accept either to let the script stop after displaying an error message or to validate the data before modifying them.

This closes the reminder on SQL data modification queries.

## 4.3 Inserting rows into a table

SQLfast allows rows to be inserted, without using SQL, by clicking on button **Insert a row** in data windows. This opens a dialog box with one entry field per column, in which the user types the values of the row to insert. By clicking on button **OK**, the row is automatically inserted. In *Expert* interface, the insertion can also be carried out in two steps:

– asking the user to enter the values of the future row (statement **ask**)

– executing an SQL **insert** query to store the data in the table.

Script 4.5 shows the use of the **ask** statement for letting the user provide six values that are then used to insert the row. The dialog box of the **ask** statement is shown in Figure 4.2. For this box, we have added a short message explaining the goal of the dialog: `Enter customer data`. This message is specified in the **ask** statement between **[**square brackets**]** before the list of field labels. In addition, mandatory fields are indicated by symbol "*".

```
ask cid,nam,adr,cit,cat,acc = [Enter customer data]
    CustID*:|Name*:|Address*:|City*:|Cat :|Account*:;
insert into CUSTOMER
    values('$cid$','$nam$','$adr$','$cit$','$cat$',$acc$);
```

**Script 4.5 -** Inserting a new row in table CUSTOMER [Chapter-05-02.sql]

## 4.4 About single quotes in SQL values

The way a value is inserted into the **insert** query, through substitution operator **$nam$**, will fail when the value includes single quotes. Indeed, the latter are also used to delimit character string values, so that the SQL engine will be completely confused and will report a syntax error.

To avoid this confusion, the SQL syntax dictates that single quotes that are part of a value be doubled. For example, value **O'Hara** must be written **O''Hara** in any SQL query in which it appears.

To this aim, SQLfast provides another substitution operator, **§**`nam`**§**, that behaves similarly to `$nam$`, except that all single quotes are doubled. So, a more secure **insert** query would be:

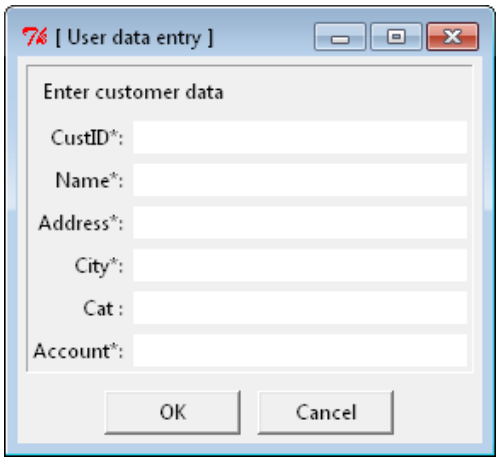



**Figure 4.2 -** The dialog box for collecting the data required to create a new row in table CUSTOMER

```
insert into CUSTOMER
values('§cid§','§nam§','§adr§','§cit§','§cat§',$acc$);
```

## 4.5 Updating rows of a table

To update a row or a set of rows, we must specify the selection condition that identifies this row (the **where** clause) and the new values of the columns to modify (the **set** clause). Script 4.6 shows the use of the **ask** and **update** statements for updating the city and address of a customer identified by his customer ID.

The dialog box is shown in Figure 4.3. Though this script does the job, it doesn't do it in a particularly user-friendly way. We will develop later a better way to update data.

```
ask cid,adr,city = [Update customer data]
    Customer ID:|New address:|New city:;
update CUSTOMER
set    Address = '§adr§',
       City    = '§city§'
where  CustID  = '§cid§';
```

**Script 4.6 -** Selecting and updating the data of a customer [Chapter-05-03.sql]

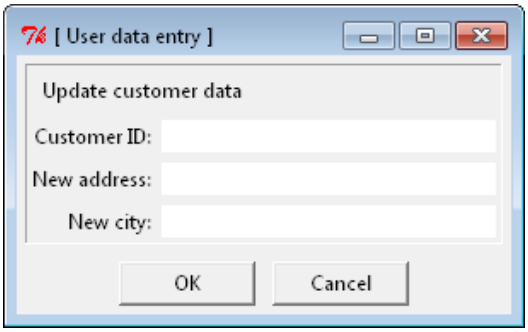



**Figure 4.3 -** Collecting the data required to update a row of table CUSTOMER

## 4.6 Variables in statement customization

Variables can be used to build larger parts of an SQL query. In the example of Script 4.7, the user is asked to specify a table and an SQL condition (Figure 4.4). From these fragments, the script builds and executes the complete SQL query[3]:

```
delete from DETAIL where ProdID = 'PA60';
```

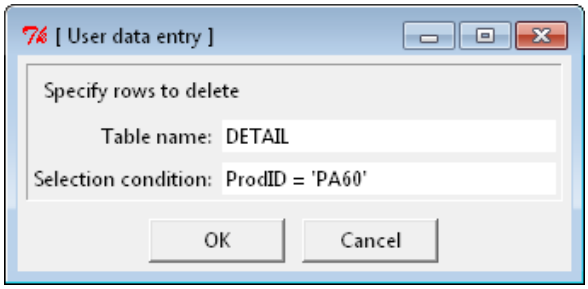



**Figure 4.4 -** Getting fragments of a *delete* query

## 4.7 Variables in full statement specification

This application is the ultimate form of statement customization: the user does not provide fragments of a statement but a full statement. In the example of Script 4.8

3. This kind of script must be used with much caution and by trusted users only. Indeed, it may lead to *SQL code injection attack*, in which a malicious user tries to counterfeit a query to execute illegal actions.

and Figure 4.5, the user asks for deleting all the data of table `DETAIL`. This short script acts as a primitive *interactive SQL interpreter.*

```
ask table,condition = [Specify rows to delete]
    Table name:|Selection condition:;
delete from $table$ where $condition$;
```

**Script 4.7 -** The user chooses the table and the selection condition of a delete query [Chapter-05-04.sql]

```
ask stat = [User-defined statement] SQLfast statement:;
$stat$;
```

**Script 4.8 -** Executing an SQLfast statement given by the user [Chapter-05-05.sql]

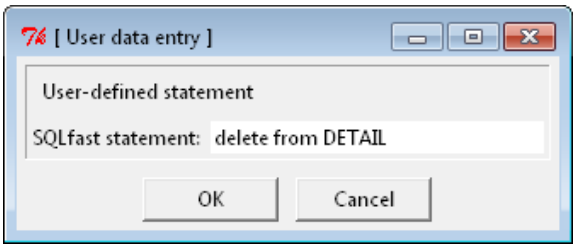



**Figure 4.5 -** Getting a full SQLfast statement

### Note

This little script can be used to execute (almost!) any SQLfast statement. For example, try to enter each of the following character strings:

```
set C = Poitiers
write Customers from $C$:
select * from CUSTOMER where City = '§C§'
```

## 4.8 Improving data entry box layout

When collecting a large number of fields, a box can appear as a quite long column of fields that may prove difficult to use. In addition, one sometimes wants to group data fields in logical columns: *administrative data*, *contact data*, *statistics*, *optional data*,

etc. Figure 4.6 shows a variant of the dialog box of Figure 4.2 in which the fields are displayed in two equal length columns.

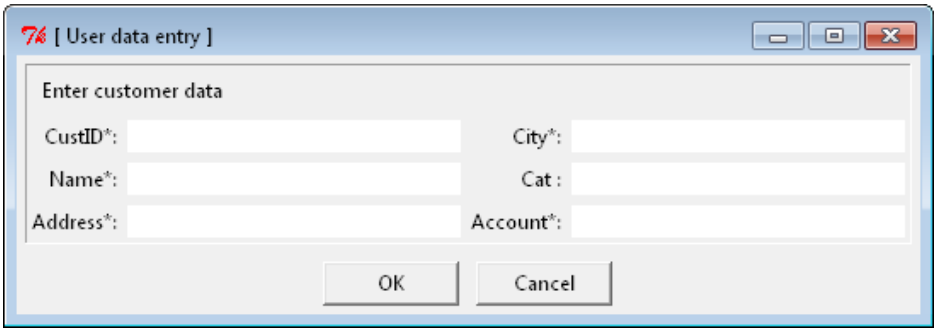



**Figure 4.6 -** Two-column data entry box

Defining a multicolumn data entry box cannot be simpler: a **column break** is specified by two vertical bars ||. In Script 4.9, this break symbol is inserted between labels Address and City. It stops the first field column and starts a second column.

Layouts with non-equal length columns and with more than two columns can be defined in the same way.

```
ask cid,nam,adr,cit,cat,acc = [Enter customer data]
    CustID*:|Name*:|Address*:||City*:|Cat :|Account*:;
```

**Script 4.9 -** Defining a multicolumn data entry box [Chapter-05-06.sql]

The last script as well as its former version are fine as long as valid data are entered through the dialog box. However, if incorrect values are provided, an SQL error is displayed and the script stops. This behavior is not particular user-friendly, to say the least!

So far, we do not know enough of the SQLfast language to be able to cope with such events in an appropriate way but those who are impatient to understand how to write more robust scripts, can examine the script in file Chapter-05-06bis.sql in directory Chapter-05. We will explain it later on.

## 4.9 Dialog box for updating values

In some circumstances, we want the user to ***change*** initial values instead of ***entering*** new ones. In other words, the entry box submits current values to the user and invites him to change some of them. To this aim, we use a variant of the **ask** state-

ment, namely **ask-u** or **askData-u** (**u** for *update*). Its use is illustrated in Script 4.10 and Figure 4.7.

Take note that this script anticipates a bit on database extraction statement **extract**, that will be studied in Chapter 8. For now, we only need to know that this statement stores in variables `adr` and `city` the address and city of the customer specified by the user through statement **ask** (C400 in this example).

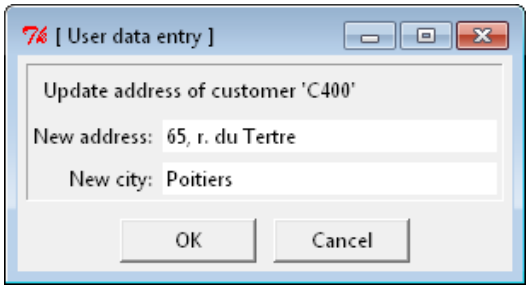



**Figure 4.7 -** Data entry box with initial values

```
openDB ORDERS.db;
  ask cid = Customer ID;
  extract adr,city = select ADDRESS,CITY
                     from CUSTOMER where CustID = '$cid$';
  ask-u adr,city = [Update address of customer '$cid$']
                   New address:|New city:;
  update CUSTOMER
  set    Address = '$adr$',
         City    = '$city$'
  where  CustID  = '$cid$';
  commitDB;
closeDB;
```

**Script 4.10 -** Updating existing values [Chapter-05-07.sql])

## 4.10 Summary

We have learned some interesting facts in this chapter.

- How to get data from the user to **insert**, **update** and **delete** rows in a table.
- A variable keeps its value even when the script which created it is finished.
- A data entry box can comprise several data entry fields (separated with | symbols).

- A data entry box can include a message field (between square brackets), informing, e.g., on the goal of the box.
- The entry fields (and their labels) can be laid out in one or several columns (column break | |).
- If the value of a variable must be inserted as a constant in an SQL query, we should use the substitution operator **§ §** instead of **$ $** to double single quotes.
- The **ask-u** variant of the **ask** statement allows users to change values already displayed in the entry fields.
- An data entry box can be used to collect SQLfast statement fragments or even full statements.

# Chapter 5

# SQL in a nutshell

This chapter is a short reminder of the main concepts and query types of the SQL-92 language.

## 5.1 Introduction

The primary goal of this book is to introduce the reader to some important aspects of database programming through the SQLfast language. It is not to train them in the SQL language and in all its subtleties. So, the reader is expected to have taken some introductory lessons or readings in SQL, for example in the countless SQL tutorials available on the internet.[1]

However, it may be useful to recall, very briefly, some of the basic principles of SQL. This is the goal of this chapter.

SQL (Structured Query Language) is the standard language used to create, manage and query relational databases. It was first published in 1974 and its first ANSI standard was finalized in 1986.

SQL is a very rich language. A precise description of all the variants of its statements would require hundreds of pages. In this chapter we will modestly recall, through representative examples, some of the most important forms and features of SQL-92 (also called SQL2) and SQL-1999 among those that will be used in this tutorial.

1. Or, for French speaking readers in the reference book *Bases de données, Dunod 2022*!

## 5.2 Creating and modifying the schema of a database

A database comprises a set of tables. There is no standard SQL query to create and to delete a database (though there are SQLfast statements to do so).

SQL query **`create table`** is used to create a new table (Script 5.1 and 5.2). This query specifies the **name** of the table (mandatory), includes the description of its **columns** (mandatory), the specification of **primary** and **unique keys**, the specification of **foreign keys** and **check** predicates.

The description of a column includes its **name** (mandatory), its **data type** (mandatory), its **`not null`** constraint, its **default value** and its **check** predicate. The main standard data types are shown in Figure 5.1.

A table is deleted by query drop table (Script 5.3).

```
create table CUSTOMER (
       CustID      char(10)     not null,
       Name        varchar(32)  not null,
       Address     varchar(60)  not null,
       City        varchar(30)  not null,
       Cat         char(2),
       Account     decimal(9,2) not null,
       primary key (CustID));
create table CUSTORDER (
       OrdID       char(12)     not null,
       CustID      char(10)     not null,
       DateOrd     date         not null,
       constraint PKORD primary key (OrdID),
       constraint FKORDCUST foreign key (CustID)
          references CUSTOMER on delete cascade);
```

**Script 5.1 -** Creating tables

```
create table if not exists INVOICE (...);
```

**Script 5.2 -** Creating a table if it does not exist yet

```
drop table INVOICE;
drop table if exists INVOICE;
```

**Script 5.3 -** Deleting a table (two variants)

A *uniqueness* constraint is declared by a *primary key* or by a *unique key* (through a **`unique`** predicate) (Script 5.4).

```
primary key (ItemID) ...
unique (ProdID) ...
```

**Script 5.4 -** A table can have one primary key (ItemID) but several unique keys (ProdID)

| | | | |
|---|---|---|---|
| *numeric* | *exact* | | smallint, integer, decimal, numeric |
| | *approximate* | | real, float, double |
| *string of* | *bits* | *fixed length* | bit |
| | | *variable length* | bit varying |
| | | *undefined length* | BLOB |
| | *characters* | *fixed length* | char |
| | | *variable length* | character varying (or varchar) |
| | | *undefined length* | CLOB |
| *temporal* | *date* | | date |
| | *instant* | | time |
| | *instant date* | | timestamp |
| | *interval* | | interval |
| *boolean* | | | boolean |

**Figure 5.1 -** The datatypes of SQL2

A *foreign key* is a list of columns that references the primary key or a unique key of another table or its own table (Script 5.5). The way the SQL engine reacts when a data modification is asked on the target table of foreign key is defined by the **on delete** clause for the **delete** query and **on update** for the **update** query (5.6).

```
foreign key (ItemID) references ITEM
foreign key (ProductID) references ITEM(ProdID)
```

**Script 5.5 -** A foreign key references a primary key (ItemID) or a unique key (ProdID) of the target table

```
foreign key (CustID) references CUSTOMER
        on delete no action
        on update cascade
```

**Script 5.6 -** How does the SQL engine react when a delete or an update is performed in the target table?

A **check** predicate defines a constraint that must be met whenever a modification applies on the data of the table. In case of violation, the action is rejected (Script 5.7).

```
check (Cat is null or Cat in ('B1','B2','C1','C2'));
```

**Script 5.7 -** Check predicate defining an integrity constraint

An index is a secondary data structure associated with a table. It is defined by a list of column names or expressions. An index is created by a **create index** query and deleted by a **drop index** query (Script 5.8). It allows a fast access to the rows of the table that have definite values of the columns or expressions of the list. An index can also accelerate **select** queries that include an interval condition (**between**, **>**, **like** 'B%'), a **group by** clause or an **order by** clause. Most DBMS automatically associate an index to primary keys, and, quite often, to unique keys as well. In these cases, they need not be explicitly created.

```
create index XCUSTCITY on CUSTOMER(City);
create index if not exists XCUSTCITY on CUSTOMER(City);
drop index XCUSTCITY;
drop index if exists XCUSTCITY;
```

**Script 5.8 -** Creating and deleting an index

The schema of an existing database can be modified by adding and deleting a table, by adding or deleting an index. In addition, specific query **alter table** can be used to modify certain components of a table.

One can add or remove a column, add or remove a uniqueness constraint, add or remove a foreign key, change the *not null* status of a column as well as its length, its type and its default value. The name of a table or of a column can be changed.

Some variants are shown in Script 5.9

```
alter table PRODUCT add column Weight smallint
alter table PRODUCT drop column Weight;
alter table CUSTOMER add primary key (CliID);
alter table CUSTOMER add unique (Name,Address,City);
alter table CUSTORDER add foreign key (CustID)
            references CUSTOMER;
alter table CUSTOMER alter CAT not null;
```

**Script 5.9 -** Modifying the schema of an existing database

When an object is necessary for specific processing but can be discarded when the session is closed, it can be created *temporarily*, as shown in Script 5.10.

```
create temporary table INVOICE ...
create temporary view ...
create temporary index ...;
create temporary trigger ...;
```

**Script 5.10 -** Creating temporary objects

## 5.3 Single table extraction queries

A typical data extraction query is shown in Script 5.11. The **from** clause indicates from which tables the data are extracted (here, CUSTOMER). The **where** clause (optional) specifies the condition that identifies the rows from which the data are extracted (here, those where City = 'Toulouse'). The **select** clause indicates the data to extract (here, the values of columns CustID and Name). When all the columns are mentioned, the shorthand ***** can be used (**select * from** CUSTOMER). The **order by** clause (optional) specifies the order in which the data will be delivered.

```
select CustID, Name
from   CUSTOMER
where  City = 'Toulouse'
order by Name;
```

**Script 5.11 -** A typical data extraction query

SQL offers a wide variety of row selection conditions that appear in the **where** clause. Script 5.12 shows some of them. Most are self-explaining. To check whether a column value is *null*, we **must** use condition **is null** (and its opposite, is not null) and never **= null**, because *null* cannot be compared to anything, itself included!

Operator **like** deserves some explanation. **like** links two expressions. The left side expression, which is a character string (here, column Address), is analyzed to check whether its value matches the right side expression. The latter is a character string comprising plain characters (here N, e, u, v, e) and, possibly, special characters **%** and _. These two characters are place-holders which match any character substring (**%**) or any single character (_). In our example, the condition is satisfied when the value of **Address** is formed as follows: an arbitrary character string (**%**) + characters **Neuve** + any character string (**%**). In short, the value of column **Address** includes the word **Neuve**.

```
City in ('Toulouse','Namur','Breda')
Account between 0 and 1000
Address like '%Neuve%'
Cat is null
City = 'Toulouse' and Account < 0
```

**Script 5.12 -** Some selection conditions

The **select** clause can include, besides column names, constants and computing expressions (Script 5.13). Each element of the **select** list can be assigned a local name, called *column alias* (**as** VAT_Value).

```
select ProdID, 0.21*Price*QonHand as VAT_Value
from   PRODUCT
where  QonHand > 500;
```

**Script 5.13 -** Computing expression and column alias in the select list

In the **select** clause, the modifier **distinct** eliminates duplicate rows in the result (Script 5.14).

```
select distinct City
from   CUSTOMER;
```

**Script 5.14 -** Eliminating duplicates

## 5.4 Subqueries

A subquery is a complete **select** query that appears in the **where** clause of a parent **select** query. Its purpose is to designate a set of data in the database that will be used to evaluate a condition in a selection condition. In Script 5.15, subquery (**select** CustID **from** CUSTOMER **where** City = 'Genève') designates ID's of the customers of Genève. The parent query extracts data on the orders placed by these customers. Note the **in** condition, that has been shown in Script 5.12, in which the right side was a list of constants.

Script 5.16 illustrates another use of subqueries. Function **exists** takes as argument an expression that returns a set of rows. If this set is not empty, the **exists** evaluates to **true**, otherwise it evaluates to **false**.

```
select OrdID,DateOrd
from   CUSTORDER
where  CustID in (select CustID
                  from   CUSTOMER
                  where  City = 'Genève');
```

**Script 5.15 -** A subquery (1)

The subquery designates the rows of **DETAIL** of a *certain product*.[2] Which product? The one which is *current* (that is, which is under examination) in the parent query, and which is denoted by *table alias* **P**. If this set of DETAIL rows is not empty, then the current PRODUCT row is selected, otherwise, it is ignored. In summary, this query extracts data of products referenced by at least one detail.

```
select ProdID,Description
from   PRODUCT as P
where  exists (select *
               from   DETAIL
               where  ProdID = P.ProdID);
```

**Script 5.16 -** A subquery (2)

Script 5.17 returns the same data by counting the numbers of DETAIL rows depending on the current PRODUCT row.

```
select ProdID,Description
from   PRODUCT as P
where  (select count(*)
        from   DETAIL
        where  ProdID = P.ProdID) > 0;
```

**Script 5.17 -** A subquery (3)

## 5.5 SQL functions

The expressions that appear in the `select` list and in the conditions of the `where` clause can be simple column names or constants, but they also can be written as more or less complex numeric, character or time expressions.

Numeric operators and functions include the arithmetic operators (+, -, /, *), absolute value, modulo, round, sign, power, etc.

2. Symbol ***** in the **`select`** list stands for "*all the columns*".

Character string operator and functions include upper and lower case, concatenation (| |), substring, leading and trailing characters removing (trim), type changing (cast), etc.

Time functions include date and time formatting, time arithmetics, conversion, component extraction (**extract**), etc. Several temporal system variables are available: **current_date**, **current_time**, **current_timestamp** (= data + time), etc.

Script 5.18 shows some of them in **select** expression. It is recommended to name expressions by assigning to each of them a *column alias*. It is possible to give names that do not conform to the SQL name syntax, in which case this name must be enclosed between double-quotes.

```
select (Price - 10) * 2 + Bonus as Total
select lower(Address) as lowAddress,
       trim(Address)||' '||City as "Full address",
       cast(Date as char(12)) as StrDate
select current_date,current_time,
       extract(year from DateOrd) as Year
```

**Script 5.18 -** Numeric, character and time expressions with column aliases

The **case** functions will be used to specify conditional values. In Script 5.19, the query writes the value of CustID, the word `Good`, `Average` or `Bad` according to the value of Cat, then the value of Account.

```
select CustID, case when Cat like 'B_' then 'Good'
                    when Cat like 'C_' then 'Average'
                    else 'Bad'
               end as Category, Account
```

**Script 5.19 -** The **case** function

The **coalesce** function will be particularly useful. It converts potential *null* values into some conventional *normal* value:

```
select CustID, Name,
       coalesce(Cat,'no value') as "Cat"
from   CUSTOMER;
```

**Script 5.20 -** The **coalesce** function

## 5.6 Aggregate (statistical) functions

The **select** list can include aggregate (or statistical) functions that each returns one value for the set of selected rows. Script 5.21 gives the number of customers in Poitiers (**count**) together with the greater of their Account values (**max**) and their sum (**sum**). SQL provides other functions: **count**(Cat) (number of *not null* values of Cat), **count**(**distinct** City) (number of distinct values of City), **min** and **avg** (minimum value and average value). Special function group_concat will be described later.

```
select count(*) as Nbr, max(Account) as Highest,
       sum(Account) as Total
from   CUSTOMER
where  City = 'Poitiers';
```

**Script 5.21 -** This query returns one row comprising three statistical values

## 5.7 Joining tables

Results extracted from more than one table can be built by joining two or more tables. Script 5.22 *joins* tables CUSTOMER and CUSTORDER by returning rows formed by one row in each table for which the value of CustID is equal. In practical terms, the result set comprises all the rows of CUSTORDER, augmented by column values from the corresponding row in table CUSTOMER. Condition CUSTORDER.CustID = CUSTOMER.CustID is the *join condition*. The elements of the **select** list can come from both tables. To avoid ambiguity, column names CustID is prefixed with the name of the table they belong to. Table aliases can be used instead.

```
select OrdID, CUSTOMER.CustID, DateOrd, Name, City
from   CUSTORDER, CUSTOMER
where  CUSTORDER.CustID = CUSTOMER.CustID;
```

**Script 5.22 -** Joining tables CUSTOMER and CUSTORDER on CustID

Besides the join condition, the **where** clause may include other selection conditions as well (Script 5.23). To shorten the condition expressions, using table aliases (here **D** and **P**) is recommended. Computing expressions and aggregate functions can also appear in the **select** list.

```
select 'Total order 30184 = ', sum(Qord*Price) as Total
from   DETAIL as D, PRODUCT as P
where  OrdID = '30184'
and    D.ProdID = P.ProdID;
```

**Script 5.23 -** Computing the total amount of order # 30184

A table can be joined with itself. In Script 5.24 table **PERSON**, which comprises columns PersID, Name and Spouse (yes, a spouse is a person), is joined with itself to produce couples of persons.

```
select P1.Name, P2.Name
from   PERSON as P1, PERSON as P2
where  P1.Spouse = P2.PersID;
```

**Script 5.24 -** Self-join: extracting couples of persons

SQL also provides alternative syntaxes to specify joins (Script 5.25). They do not bring any additional power to the language. Their use as substitutes for the regular syntax is a matter of taste!

```
select *
from   CUSTOMER natural join CUSTORDER
where  ...

select *
from   CUSTOMER join CUSTORDER using (CustID)
where  ...

select *
from   CUSTOMER as C join CUSTORDER as M
       on (C.CustID = M.CustID)
where  ...
```

**Script 5.25 -** Alternative syntaxes for the join of Script 5.22

When we join two tables, the rows of a table that have no matching row in the other table do not appear in the result set. They can be added through an **outer join** (Script 5.26). This operator acts as a standard join but also includes the non matching rows of the *left side* table (here CUSTOMER) to the result set. The missing columns are set to *null*. There are three variants: *left outer join*, *right outer join* and *full outer join*.

```
select OrdID, C.CustID, DateOrd, Name, City
from   CUSTOMER as C left outer join CUSTORDER as M
       on (M.CustID = C.CustID);
```

**Script 5.26 -** Outer join

## 5.8 Set operators

The result sets of two **select-from** queries can be combined through the standard set-theoretic operators union (**union**), difference (**except**) and intersection (**intersect**). Script 5.27 returns the IDs and the cities of the persons who are not customers with an account level greater then zero.

```
select PersID,City
from   PERSON
  except
select CustID as PersID,City
from   CUSTOMER where Account > 0
order by City,PersID;
```

**Script 5.27 -** Difference operator (**except**)

The result of a *union*, *intersection* or *difference* of two tables is a set or rows, with no duplicates, even if the source tables include duplicates. To keep these duplicates in the result set, we must use variants of these operators: **union all**, **intersect all** and **except all**.

## 5.9 Data grouping

Data grouping consists in partitioning the rows of a table in subsets (groups) with the same values of a column, a set of columns or, more generally, with the same values of expressions. Data grouping is a special form of the **select-from** query. It includes a **group by** clause, that specifies the grouping criterion, and its **select** list is limited to constants, elements of the **group by** criterion and aggregate functions.

Script 5.28 returns, for each group of rows with the same value of City, the *city name*, the *number* of customer in this city and the *sum* of their accounts. The groups can also be selected through the **having** clause. Here, only cities with more than one customer will appear in the result.

```
select City,count(*) as Nbr,sum(Acount) as Total
from   CUSTORDER
group by City
having Nbr > 1;
```

**Script 5.28 -** Simple data grouping query

Script 5.29 gives, for each customer who has ordered products for more than 100 €, the total amount of their orders. Here, the **`group by`** clause applies on the join of three tables.

The aggregate functions of Section 5.6 can be used here. They apply to each group of rows considered independently. Most DBMSs offer an additional aggregate function, `group_concat`, that provides the concatenation of values from the elements of the group. Script 5.30 is an example of application of this function, as it is implemented in SQLite. Figure 5.2 shows the result of the execution of this query.

```
select M.CustID, sum(Qord*Price) as Total
from   CUSTORDER as M, DETAIL as D, PRODUCT as P
where  M.OrdID = D.OrdID
and    D.ProdID = P.ProdID
group by M.CustID
having Total >= 100;
```

**Script 5.29 -** Grouping data from the join of several tables

```
select City, group_concat(CustID,',') as Customers
from   CUSTOMER
group by City
order by City;
```

**Script 5.30 -** Application of the *group_concat* function

```
+-----------+--------------------------+
| City      | Customers                |
+-----------+--------------------------+
| Bruxelles | F400                     |
| Genève    | B332                     |
| Lille     | K111                     |
| Namur     | C123,S127,B062,L422      |
| Paris     | S712                     |
| Poitiers  | B112,F010,C400           |
| Toulouse  | B512,C003,K729,F011,D063 |
+-----------+--------------------------+
```

**Figure 5.2 -** Result from the execution of Script A5.30

Unfortunately, the syntax of the **group_concat** function varies among DBMS but most of them include a variant of this function: SQLite with group_concat, MySQL with group_concat, Oracle with listagg, DB2 through XML function xmlagg, SQL Server through XML functions, PostgreSQL with string_agg. Some experts distrust this function, arguing that it may not be deterministic, notably due to the random ordering of the fragments when no specific order is specified.

The full group_concat, as understood by MySQL, includes 5 parameters: the expression specifying the values to concatenate, whether duplicate values are allowed or not, the value separator, the order in which these values appear in the result and the direction of this order (ascending, descending). SQLite understands a subset of these parameters only, but SQLfast provides function group_concat2, that simulates the MySQL variant.

## 5.10 Generalization of the select and from clauses

The **select** clause specifies the *select list*, that is, a list of expressions that each returns a simple value (a number, a character string, a date) for each row. Among these expressions, we can use a **select-from** subquery, provided it returns **one** simple value.

Script 5.31 extracts a list of customer IDs together with the number of orders they have placed.

```
select CustID, (select count(*) from CUSTORDER
                where  CustID = C.CustID) as NbOrders
from   CUSTOMER as C
where  NbOrders > 0;
```

**Script 5.31 -** Subquery in the **select** clause

Similarly, each element of the **from** list must be a table. However, since a **select-from** query returns a table, it can be used in this **from** list in lieu of any table name. Scripts 5.32 and 5.33 shows examples of this form.

```
select CityName,Population
from   (select City, count(*) from CUSTOMER
        group by City) as Cities(CityName,Population)
where  Population > 2;
```

**Script 5.32 -** Subquery in the **from** clause

```
select CustID, Name
from ((select CustID,Name,City from CUSTOMER)
       except
      (select CustID,Name,City from VIP_CUSTOMER)
       union
      (select CustID,Name,City from PROSPECT))
where City = 'Poitiers';
```

**Script 5.33 -** Set expression as table reference in the **from** clause

## 5.11 Data modification

Statement **insert** is used to insert a new row, or several new rows in a table. These rows are made up of lists of expressions, most often mere constants (Script 5.34). The new data must satisfy the integrity constraints declared on the table, otherwise the operation is cancelled.

```
insert into DETAIL values ('30185','PA45',12);

insert into DETAIL values ('30185','PA45',12),
                          ('30185','PA60',25),
                          ('30185','PH222',100);
```

**Script 5.34 -** Inserting rows in a table

With the second variant of the **insert** statement, the data are not (only) constants, but come from other tables through a **select-from** statement (Script 5.35).

```
insert into CUSTOMER_TOULOUSE
     select CustID, Name, Address
     from   CUSTOMER
     where  City = 'Toulouse';
```

**Script 5.35 -** Inserting rows comprising data from other tables

The **delete** statement is used to erase rows from a table. It includes a **where** clause to specify the rows to delete (Script 5.36).

```
delete from  CUSTOMER where CustID = 'K111';

delete from  DETAIL
where ProdID in (select ProdID
                 from   PRODUCT
                 where  QonHand <= 0);
```

**Script 5.36 -** Deleting rows

The **update** statement will be used to modify the value of columns of selected rows. It includes a **where** clause to specify the rows to modify (Script 5.37).

```
update CUSTOMER
set    Address = '29, av. de la Magne',
       City = 'Niort'
where  CustID = 'F011';

update PRODUCT
set QonHand = QonHand - (select coalesce(sum(Qord),0)
                         from   DETAIL
                         where  ProdID = PRODUCT.ProdID)
where QonHand > 200;
```

**Script 5.37 -** Updating rows

## 5.12 SQL views

A view is a virtual table the content of which is evaluated when this table is queried. A view is defined by a **select-from** query that tells how to compute its content (Script 5.38). The signature of a view specifies its name and the name of its columns.

```
create view ORDER_FULL(OrdId,CustId,CustName,CustCity,DateOrd)
as  select OrdID,O.CustID,Name,City,DateOrd
    from   CUSTOMER C,CUSTORDER O
    where  O.CustID = C.CustID;
```

**Script 5.38 -** Definition of a view

Theoretically, the data of a view can be modified if the SQL engine is able to translate the modifications into equivalent operations on the actual tables. Practically, most DBMS enforce strong limitations on these modifications. If needed, one can define the effect of view modifications by **instead of** triggers associated with this view.

## 5.13 Stored procedures and triggers

A **stored** (or SQL) **procedure** is a procedure created as an object of the schema of a database. Its body comprises SQL statements but also, quite often, variables and procedural statements (*assignment*, *if-then*, *while* statements). A stored procedure can be invoked by any program which has a connection with the database, by triggers or by other stored procedures. Stored procedures are not standardized[3] and their syntax and semantics are implementor-dependent.

A **trigger** is a procedure that is activated when a definite event occurs, such as a data modification operation. The definition of the trigger typically specifies (Script 5.39)

- the nature of the event (**`insert`**, **`update`**, **`delete`**),
- when the trigger fires (**`before`**, **`after`** or **`instead of`** the modification),
- whether this execution occurs once for the statement that triggered it (**`for each statement`**) or once for each row affected by the modification (**`for each row`**),
- the (optional) condition which must be satisfied for the trigger to execute (**`when`** clause),
- and the body that comprises the procedure itself (**`begin end`**).

During the execution of the procedure of the trigger, the current row is given two aliases, **old** and **new**. **old** denotes the state of the row just before the modification and **new**, the state of the row just after the modification. For an `update` operation, both states exist, for an `insert` operation, only the **new** state exists while only **old** exists for a `delete` operation.

One of the frequent goals of a trigger is to cancel the source modification operation when integrity constraints are not, or will not be, satisfied.

Triggers are not standardized and their syntax and semantics are implementor-dependent.

```
create trigger DELETE_ORDER
after update of Cat on CUSTOMER
for each row
when (new.Account < 0)
begin
  if (old.Cat is not null and new.Cat is null)
  then delete from CUSTORDER where CustID = old.CustID;
  endif;
end;
```

**Script 5.39 -** Definition of a trigger

3. Though a different but somewhat similar construct is defined in the standard, the ***module***.

An interesting variety of trigger is the so-called *recursive trigger*. Its body includes modification queries that will fires the trigger itself.

Table EMPLOYEE(EmpID,Manager) stores the data of employees (their Id's and the Id of their manager, who is a employee). TEAM is a working table in which one stores the data of an employee and of all the employees of whom he is the direct or indirect manager. If we insert a row in table TEAM we would like to also insert the rows of all his subordinates as they are described in table EMPLOYEE. This is done by trigger TEAM_EMP (Script 5.40).

```
create trigger TEAM_EMP
after insert on TEAM
for each row
begin
  insert into TEAM
      select * from PERSON
      where  Manager = new.EmpID;
end;
```

**Script 5.40 -** Recursive trigger

## 5.14 About SQLite

SQLite follows the SQL2 (SQL-92) standard fairly faithfully with some features from SQL3 (SQL-1999). However, some variations must be taken into account.

### Data types

The data type system of SQLite is proprietary (the so-called *affinity typing*) and does not follow the standard but it is compatible with the usual data types (Figure 5.1). Therefore, standard SQL DDL scripts are allowed to define an SQLite schema.

*Reference*: http://www.sqlite.org/datatype3.html

### Set operators

Operators **`except all`** and **`intersect all`** are not allowed but are easy to simulate.

### Alter table

This statement is limited to two operations: renaming a table, adding a new column and renaming a column.

### View

SQLite views are not updatable. Data modification in a view can be carried out through **instead of** triggers.

### SQL stored procedures

There is no stored procedure in SQLite. However, they can be simulated by special patterns of triggers.

### Triggers

Only the **for each row** scope is available. The body of a trigger is a sequence of SQL queries. There is no procedural statements but a limited *if-then-else* structure can be defined through the **where** clause of these queries.

Aborting an operation is obtained by a special version of the **select** query using SQL function **raise** (Script 5.41).

```
create trigger TR_PROD_1
before update of QonHand on PRODUCT
for each row
when new.QonHand <= 0
begin
    select raise(ABORT, 'No more product available')
    where old.QonHand = 0;
    select raise(ABORT, 'Quantity impossible to supply')
    where new.QonHand < 0;
end;
```

**Script 5.41 -** Aborting an operation through a **select** query

## 5.15 Further reading

- https://en.wikipedia.org/wiki/SQL
- Joe Celko, *SQL for Smarties, Advanced SQL Programming* (Fifth Edition), 2015, Morgan Kaufmann
- http://www.sqlite.org/
- *The definitive guide to SQLite*, G. Allen & M. Owens, APress, 2010
- http://www.tutorialspoint.com/sqlite; PDF version: http://www.tutorialspoint.com/sqlite/sqlite_pdf_version.htm

- http://sqlite.awardspace.info/)
- http://www.tutorialspoint.com/sql

# Chapter 6

# More on variables

**Objective**: This chapter studies some advanced usage of variables: time variables, variable/value substitution and computing.

**Keywords**: variable, date, time, timer, elapsed time, variable delimiters, substitution ambiguity, array, quoted values, variable scope

**Statements and functions**: start-timer, create index, drop index, output, parameter delimiters, set-n, substitution, dropVariable, §, compute, execSQL

**System variables and parameters**: date, time, timer

## 6.1 Time variables and statements

What is today's date? What time is it? How long did this operation last? The answers are in three SQLfast variables, **`date`**, **`time`** and **`timer`**, the value of which are the current date, the current time and a duration. Time and duration are measured down to the millisecond.

It is good practice to add current date and time when writing text in the output window, as in the result of Script 6.1, shown here below:

```
Date = 25/03/2022
Time = 9:5:29.623
```

```
+--------+---------+
| CustID | Name    |
+--------+---------+
| K729   | NEUMAN  |
| D063   | MERCIER |
+--------+---------+
```

```
write Date = $date$;
write Time = $time$;
select CustID, Name from CUSTOMER where CAT is null;
```

**Script 6.1 -** Writing date and time in the output file

## 6.2 Measuring elapsed time

These variables could also be used to measure elapsed time, but SQLfast includes a much better tool for this, the *timer*.

The timer is available through a third variable, named **`timer`**. This variable is set to `0` when the SQLfast engine is started. It can also be reset through statement **`start-timer`**. It yields the time elapsed since the last reset in seconds and milliseconds. As any variable, it keeps its (evolving) value when a script terminates, so that the timer can be reset in a script and used in another one.

The timer allows a script to measure execution time or waiting time. Script 6.2 reports the time spent by the user to enter the data for a new customer, then the execution time of an **`insert`** query.

```
openDB ORDERS.db;
  start-timer;
  ask cid,nam,adr,cit,cat,acc = [Enter customer data]
      CustID|Name|Address|City|Cat|Account;
  write User input time = $timer$;
  start-timer;
  insert into CUSTOMER
      values('§cid§','§nam§','§adr§','§cit§','§cat§',§acc§);
  write Insertion time = $timer$;
closeDB;
```

**Script 6.2 -** Measuring the time spent by the user to enter data and by the SQL engine to insert a row [Chapter-06-01.sql]

Script 6.3 compares the execution time of the same query when an index is available or not.

```
openDB ORDERS.db;
  drop index if exists IDXCITY;
  start-timer;
  select * from CUSTOMER where City = 'Poitiers';
  write Execution time without index = $timer$;
  create index IDXCITY on CUSTOMER(City);
  start-timer;
  select * from CUSTOMER where City = 'Poitiers';
  write Execution time with index = $timer$;
closeDB;
```

**Script 6.3 -** Measuring SQL execution time - First trial [Chapter-06-02.sql]

The way the **select** statement behaves is not really appropriate to measure SQL query execution time. Indeed, the latter also includes the writing time in the output window, which can be significant. It would be nice to *disable writing* in the output window when we evaluate database performance. This can be done very simply. We just tell SQLfast engine to switch the output **off** and **on** with statement **output**. Script 6.4 shows how to use this feature. Output is disabled when executing **select** statements and enabled when writing execution time.

```
Execution time without index = 0.16;
Execution time with index = 0.002;
```

```
openDB ORDERS.db;
  drop index if exists IDXCITY;
  start-timer;
  output off;
  select * from CUSTOMER where City = 'Poitiers';
  output on;
  write Execution time without index = $timer$;
  create index IDXCITY on CUSTOMER(City);
  start-timer;
  output off;
  select * from CUSTOMER where City = 'Poitiers';
  output on;
  write Execution time with index = $timer$;
closeDB;
```

**Script 6.4 -** Measuring SQL execution time - Disabling output writing [Chapter-06-03.sql]

### Note

Actually, measuring query execution time is a bit more complex than this illustration lets it believe. For instance, if the size of table CUSTOMER is not two high (let's say, 100 MB), there is a high probability that the first execution of the query has brought in RAM (in the OS and/or DBMS *cache*) all the rows of the table. Same for index creation, that requires reading the whole table contents. When the second query is run, most rows still are in RAM, which artificially speeds up its execution, whether the index is used or not. However, this is another issue, that will be discussed in a further chapter.

## 6.3 Oh! Wait a minute!

While **`time`**, **`date`** and **`timer`** variables are used to *observe* the passing of time, statement **`wait`** *creates* time, in some sense. With one parameter (**`wait`** `500`), it suspends script execution during the specified number of millisecond. With two parameters (**`wait`** `200,500`), it suspends the execution a random number of milliseconds in the specified range. Script 6.5 stores three rows in table MEASURE with the value of the timer increased by a random number of milliseconds.

```
ask min,max = [Give range of wait time] Minimum:|Maximum:;
start-timer;
wait $min$,$max$;
insert into MEASURE values($timer$);
wait $min$,$max$;
insert into MEASURE values($timer$);
wait $min$,$max$;
insert into MEASURE values($timer$);
```

**Script 6.5 -** Storing increasing random numbers [Chapter-06-03bis]

We will see in Section 10.5 (Numeric expressions) that SQLfast function **`random`**, that provides a sequence of random numbers, may be a more comfortable way to store random numbers in a database.

## 6.4 Variable delimiters - Reminder

Before going into further detail about *variable/value substitution*, let us recall the bases of this operator. In assignment statement,

```
set city = Poitiers;
```

the word **city** is the name of a variable and character string **Poitiers** is a constant that becomes the current value of the variable. We cannot change the name of a variable but we can change its value.

When we want to introduce the value of a variable into a message or, more generally into an expression, mentioning its name does not provide the expected result, since we merely introduce ... the name of this variable! For instance, the statement:

```
write The customers of city;
```

where **city** is the name of a variable, just writes `The customers of` **city** in the output window, which probably is not what we intended to do. To write the message `The customers of` **Poitiers**, we must indicate that we want the SQLfast engine to *replace* the name of the variable with its value, an operation called *variable substitution* that is specified by *variable delimiters*. The correct expression is therefore,

```
write The customers of $city$;
```

## 6.5 We can change the delimiters

If the default pair of delimiters **$ $** does not fit our taste, we can change it with statement **parameter**, used as follows:

```
parameter delimiters = [ ];
```

For now on, the substitution operation must be denoted by `[city]` instead of `$city$`.

Script 6.6 illustrates various styles for the substitution operation. It shows that each delimiter is any character string and that the second delimiter can by empty, (styles 4 and 5). Style 5 is the standard SQL convention to inject variable values in queries in application programs.[1]

This script will write the following lines in the output window:

```
[1] A + B / C
[2] 123 + 45 / 6.78
[3] 123 + 45 / 6.78
[4] 123 + 45 / 6.78
[5] 123 + 45 / 6.78
[6] 123 + 45 / 6.78
```

1. When included into an application program, written in, say, COBOL, C, or Java (SQLJ) an SQL query can refer to program variables by prefixing their name by a semi-colon `':'`, to avoid confusion with schema object names and SQL keywords. So, we will find such statements as "`select * from DETAIL where OrdID =` **:OID;**" in application programs, where **:OID** means *the value of variable* ***OID***.

```
set A = 123;
set B = 45;
set C = 6.78;
write [1] A + B / C;
write [2] $A$ + $B$ / $C$;
parameter delimiters = [ ];
write [3] [A] + [B] / [C];
parameter delimiters = ';
write [4] 'A + 'B / 'C;
parameter delimiters = :;
write [5] :A + :B / :C;
parameter delimiters = value-of( );
write [6] value-of(A) + value-of(B) / value-of(C);
```

**Script 6.6 -** Experimenting with variable delimiters [Chapter-06-04.sql]

Please be conscious that, despite what the result may suggest, this script *does not perform any computation*! We must wait until Section 6.8 to study the way SQLfast carries out computations.

Restoring the default delimiters cannot be simpler[2]:

```
parameter delimiters = default;
```

## 6.6 Sometimes variable substitution must be disabled

***Warning***. This section is a bit more technical and can be skipped in a first reading.

In the examples above, the delimiters have been changed to make expressions more concise or more readable or to make them comply with the convention of other languages, such as SQL in style [5]. In some circumstances, however, we would like to change the substitution mechanism of SQLfast.

Let us suppose that we want to build a library of formulae. These formulae will be stored in variables `form-01`, `form-02`, etc. (or, better, in a temporary SQL table), through **`set`** statements. Most of these formulae will include variable names, such as `A`, `B` and `C`. So, we expect our script to include statements such as:

```
set form-01 = $A$ + $B$ / $C$;
set form-02 = ($A$ + $B$) / $C$;
```

2. The default delimiters are specified in parameter file **SQLfast.ini** as follows:
   `delimiters = $ $`

However, we do not want the names of these variables being replaced by their values **now**, when **set** statements are executed, but later, whenever the formulae are used to perform actual computations.

So, we must prevent the SQLfast engine from carrying out substitution in the **set** statements. To do so, we use a variant of the **set** statement, **set-n** (with **n**, as ***n****o substitution*) which disables variable substitution before its execution.

Script 6.7 shows a fragment that stores formulae `$A$ + $B$ / $C$` and `($A$ + $B$) / $C$` in variables `form-01` and `form-02` respectively. Thanks to the use of **set-n**, these variables now contain the formulae unchanged, with their variable delimiters preserved. When the contents of these variables are written in the output window:

```
write $form-01$;
```

variable `form-01` is first replaced by its value:

```
write $A$ + $B$ / $C$;
```

then variables `A`, `B` and `C` are replaced by their values:

```
write 123 + 45 / 6.78;
```

Therefore, this script writes the following lines in the output window:

```
123 + 45 / 6.78
(123 + 45) / 6.78
```

```
set A = 0;
set B = 0;
set C = 0;

set-n form-01 = $A$ + $B$ / $C$;
set-n form-02 = ($A$ + $B$) / $C$;

set A = 123;
set B = 45;
set C = 6.78;

write $form-01$;
write $form-02$;
```

**Script 6.7 -** Inhibiting variable substitution in **set** statements [Chapter-06-05.sql]

Should we have used simple **set** statements, the substitution would have been performed before assignment execution, so that the script would have written these silly lines:

```
0 + 0 / 0
(0 + 0) / 0
```

There is no **-n** modifier for the other SQLfast statements. Instead, we can use the **substitution** statement, which disables/enables variable substitution. Script 6.8 is a variant of Script 6.7, in which substitution has been used.

```
substitution off;
set form-01 = $A$ + $B$ / $C$;
set form-02 = ($A$ + $B$) / $C$;
substitution on;
set A = 123;
set B = 45;
set C = 6.78;
write $form-01$;
write $form-02$;
```

**Script 6.8 -** Disabling variable substitution through **substitution** statement [Chapter-06-05bis.sql]

Note that there are other ways to inhibit variable substitution.

– By changing the variable delimiters, we let **set** statements assign their values without substitution. The former delimiters are then restored to allow normal variable substitution when appropriate.

– If a variable is unknown when the statement is preprocessed, no substitution is performed for it. If the variable already exists, we can delete it with statement `dropVariable`:

```
dropVariable A;
```

## Note

Statement preprocessing consists in applying variable substitution before execution. It is time to precise how this works:

first, *for each existing variable* **v***, the engine replaces all instances of* **$v$** *by the value of* **v***. If, during this run, at least one substitution has been carried out, the process is performed again, and so on until all possible substitutions have been carried out.*

then, *for each existing variable* **v***, the engine replaces all instances of* **§v§** *by the value of* **v***. This process is applied only once.*

## 6.7 Beware substitution ambiguity

***Warning***. Same remark: this section can be skipped in a first reading.

This substitution operator is very powerful but, in some extreme cases, it must be used with care, since it may provide unexpected results. Let us consider the (stupid but valid) example of Script 6.9, in which variable **in** is empty.

The message written in the output window depends on the order in which the variables are replaced:

- processing of **$in$** first (result $AB$ since **in** is empty) then of **$A$** (failure), then of **$B$** (failure), then of **$AB$**: the result is **3**;
- processing of **$A$** first (result 1in$B$), then **$B$** (result 1in2), then **$in$** (failure), then **$AB$** (failure): the result is **1in2**.

```
set in = ;
set A  = 1;
set B  = 2;
set AB = 3;
write $A$in$B$;
```

**Script 6.9 -** A tricky script [Chapter-06-06.sql]

Since there is no way to force the order in which the variables will be processed, the result is uncertain.

The second example of Script 6.10 shows how surprising variable substitution can be.

```
set-n A = $B$;
set-n B = $A$;
write $A$;
```

**Script 6.10 -** The most scary script [ORDERS-06-07.sql]

Designers who are in such an exploratory mood as to write such a script have to pay the price! However, absent-minded script developers can unintentionally write invalid scripts that are prone to fail. Such is the case of the following sequence, that should run forever, or at least as long as there is enough RAM:

```
set text = introduction
set message = This is an $text$;
```

The missing **;** at the end of the first statement creates the following single **recursive** (it includes a reference to itself) statement:

```
set text = introduction set message = This is an $text$;
```

Fortunately, such invalid pattern will not freeze the script because the SQLfast engine limits to 100 the number of times each variable is replaced in a statement.[3]

## 6.8 Numeric computation

We know how to store a value in a variable but, sometimes, we need to combine values to produce new results, that is, to *compute* these results. There are several ways to do so. The most straightforward of them is the use of statement **compute**, as in the following example:

```
compute A = $B$ + 1;
```

This statement asks SQLfast to add **1** to the value of **B** and to store the result in variable **A**. Note the use of expression **$B$** to denote the value of variable **B**. We don't add **1** to **B**, but we add **1** to *the value of* **B**.

The **compute** statement comprises two parts, the name of the *target variable* (**A**) and the *computation* (**$B$ + 1**). The computation part can be made up of several variables and constants and combine arithmetic and other mathematical operators through the use of parentheses, as in Script 6.11, which writes:

```
78, 1623.6
```

in the output window.

We observe that the first result is an integer number (the actual result is 78.5) while the second one is a real number, with one decimal. The rule is simple: if all the arguments of a computation are integer numbers, so is the result. If at least one of them is a real number (such as the value of variable **C**) the result is a real number.

```
set A = 123;
set B = 34;
set C = 1.25;
compute AB2 = ($A$ + $B$)/2;
compute ABC = ($A$ * $B$)/(2 * $C$);
write $AB2$, $ABC$;
```

**Script 6.11 -** Numeric computations [Chapter-06-08.sql]

To force a computation to return a real number, whatever the type of the arguments, we can write:

3. Message: *Aborting infinite looping in statement preparation.*

```
AB2 = ($A$ + $B$)/2.0
```

or

```
AB2 = 1.0 * ($A$ + $B$)/2
```

The syntax and evaluation rules of computation expressions will be described in some detail in Chapter 10. Let us just say now that a computation expression, once variables have been replaced by their values, must be a **valid SQL expression**, such as those that appear in the *select list* and in the *where clause* of a **`select`** query.

Some basic functions are also available, such as *absolute value* (**`abs`**), *maximum* and *minimum* of a set of numbers (**`max`** and **`min`**) and *rounding* (**`round`**). Additional functions can be built as *User-defined Functions* (UDF) and through *functions libraries*, callable by statement **`function`**, that we will study later (Chapter 22).

## 6.9 Character string computation

Computation can apply on character strings as well. We mention some of them:

- convert a string in uppercase (**`upper`**) or in lowercase (**`lower`**),
- string concatenation (||),
- length of a string (**`length`**),
- delete leading spaces (**`ltrim`**), delete trailing spaces (**`rtrim`**), same for leading and trailing spaces (**`trim`**),
- replace a substring in a string (**`replace`**),
- find a substring in a string (**`instr`**)
- extract a substring from a string (**`substr`**)
- etc.

Script 6.12 shows some examples of string manipulation functions.

```
set A = 235, Regent street, London;
set B = Regent;
set C = King;

compute I = instr('$A$','Regent');
compute J = replace('$A$','$B$','Queen');
compute K = upper('$B$')||'S and '||upper('$C$')||'S';

write $I$;
write $J$;
write $K$;
```

**Script 6.12 -** Character string computations [Chapter-06-09.sql]

It writes the following result in the output window:

```
6
235, Queen street, London
REGENTS and KINGS
```

These string manipulation functions and operators will be studied in more detail in Chapter 10. Special string manipulation functions can be developed as *User-defined Functions* (UDF) and through functions libraries, notably **LStr**, callable by statement **function**, as mentioned above.

Let us note that string concatenation already exists in SQLfast. The following two statements are equivalent:

```
compute C = $A$||$B$;
set C = $A$$B$;
```

## 6.10 Set or compute?

What if the computing expression reduces to a single value? In such case, is the **compute** statement strictly equivalent to a simple **set** statement? For instance, we can guess that the following two statements have the same effect:

```
set X = 123.5;
compute X = 123.5;
```

Indeed, they store the same value in variable **X**. However, this is not necessary true for character string values. For example, the following statements are not equivalent:

```
set X = Owner's file;
compute X = 'Owner's file';
```

While the first statement will store value `Owner's file` in variable **X, *the second one will fail***. Indeed, it must comply with the SQL syntax, with which it does not, since it includes a single-quote. Following the SQL format for character string constants, it should be written as follows:

```
compute X = 'Owner''s file';
```

To deal with quotes and doubled quotes, we can also use three tricks: (1) using double-quotes instead of single-quotes to surround string values, (2) using substitution operator § § and (3) applying function doubleQuote() that doubles all single-quotes of its argument.

Script 6.13 shows the behavior of SQLfast in different configurations.

```
set X = Owner's file;
write 1. $X$;
write 2. §X§;
set X = Owner''s file;
write 3. $X$;
compute X = 'Owner''s file';
write 4. $X$;
compute X = "Owner's file";
write 5. $X$;
compute X = "Owner''s file";
write 6. $X$;
compute X = doubleQuote('Owner''s file');
write 7. $X$;
compute X = doubleQuote("Owner's file");
write 8. $X$;
```

**Script 6.13 -** Dealing with quotes and double quotes

It produces the following result:

```
1. Owner's file
2. Owner''s file
3. Owner''s file
4. Owner's file
5. Owner's file
6. Owner''s file
7. Owner''s file
8. Owner''s file
```

## 6.11 Multi-variables assignment

A sequence of similar **set** statements, such as the following:

```
set A = 1;
set B = 2;
set C = 3;
```

can be written in a more concise form as

```
set A,B,C = 1,2,3;
```

If a value includes, or may include, commas, it must be enclosed between curly brackets. For example, the sequence

```
set A = 1;
set B = Hello, world;
```

```
set C = 3;
```

will be written

```
set A,B,C = 1,{Hello, world},3;
```

## 6.12 Simulating an array

SQLfast offers simple variables but no arrays. In most cases, a temporary SQL table can be used instead. However, if absolutely needed, arrays can be simulated quite easily, as we will suggest in Appendix 1, Application 1.12.

The principle is straightforward. Let us imagine an array called STUDENT, in which we intend to store the names of a group of students. The names will be stored in the successive cells of the array, generally denoted by expressions STUDENT[1], STUDENT[2] to STUDENT[100]. If N is a variable containing a cell number, say, **46**, then the student name in this cell will be designated by expression STUDENT[$N$].

Some examples:

```
set N = 46;
set STUDENT[$N$] = Dominique;
set NAME = $STUDENT[$N$]$
write $NAME$
```

This technique simulates an array through a series of individual variables, each representing a cell of the array. In the examples above, expression `STUDENT[$N$]` is developed into `STUDENT[46]`, while `$STUDENT[$N$]$` is first developed into `$STUDENT[46]$` then into `Dominique`.

In Chapter 8, we will study another way to create and process arrays, through *SQLfast lists*.

## 6.13 Variables and SQLfast procedures

The question is, when a variable **X** has been created in script A that calls script B, does B know this variable? Conversely, can a variable created in B be used in A?

The answer definitely is *yes* for both, as shown in the experiment of Script 6.14. Script **Proc-A.sql** knows variables **X** and **Y,** Script **Proc-B.sql** knows variables **X**, **Y** and **Z** and the main script, once both external scripts have been executed, also knows variables **X**, **Y** and **Z**. The execution of the main script writes this sequence in the output window:

```
Proc-A: 1,2
Proc-B: 1,2,3
Main:   1,2,3
```

```
-- Main script
set X = 1;
execSQL Proc-A.sql;
execSQL Proc-B.sql;
write Main: $X$,$Y$,$Z$;
```

```
-- Proc-A.sql
set Y = 2;
write Proc-A: $X$,$Y$;
```

```
-- Proc-B.sql
set Z = 3;
write Proc-B: $X$,$Y$,$Z$;
```

**Script 6.14 -** Understanding variables in external scripts [ORDERS-06-10.sql, Proc-A.sql, Proc-B.sql]

### Conclusion

Once a variable has been created, it can be used and modified by all the scripts that are executed. It disappears when the current session is closed or when SQLfast stops. In other (technical) words, SQLfast variables are *dynamic* and *global*.

## 6.14 What is a variable value after all?

Simple: just a character string. If a variable exists and contains *nothing*, its value is an empty character string. Even numbers, time, date and timer values are character strings. What makes the difference when processing a value is the context in which this value is used. For example, values **`London`** and **`123.4`** are fine to display in the output window, but only the second one is appropriate for numeric computation.

There is however an exception for variables in which *binary*, that is, *non character values*, such as an image, a video clip or an executable program, have been stored. If the contents of a binary file or a binary value from the database has been stored in a variable, the latter gets the **binary** status. A binary variable cannot appear in a substitution operator.

## 6.15 Summary

This chapter has taught us important facts.

- SQLfast provides three time related system variables: **`time`**, **`date`** and **`timer`**. They can be used but not modified. The **`timer`** variable allows measuring elapsed time. It can be reset through statement **`start-timer`**.
- The variable delimiters that define the substitution operator can be changed from the default pair **`$ $`** to any pair of character strings through statement **`parameter delimiters`**. The second character string of the pair can be empty.
- Statement **`set-n`** is equivalent to **`set`** except that no variable substitution is performed before execution.
- Statement **`substitution`** can be used to disable or enable variable substitution for all succeeding statements.
- Statement **`dropVariable`** can be used to delete a variable.
- Variable substitution may, in some tricky patterns, lead to indeterminacy or to infinite looping. The SQLfast engine as able to detect such misbehavior and to stop substitution.
- In the multi-variables variant of the **`set`** statement, more than one target variables can be specified. The same number of values must be specified at the right of sign `=`.
- Statement **`compute`** is used to assign the result of a numeric or character string computing expression to a variable. Such expression, after variable substitution have been applied, must comply with the syntax of SQL expressions.
- A large set of numeric and character string functions is available, either built-in or through User-defined Functions and SQLfast libraries.
- SQLfast variables are dynamic and global. They exist from the time they were created to the closure of the current session or of the SQLfast system. In particular, they persist beyond the script execution that created them and can be used by further scripts.

# Chapter 7

# Taking decisions

**Objective**: This chapter studies various ways to write a script that is able to perform alternative tasks according to data values from the user or from the database.

**Keywords**: alternative control structure, elementary if statement, block if statement, condition, loop, displaying a message

**Statements and functions**: if, endif, else, goto, label, showMessage

**System variables and parameters**: DIALOGbutton

## 7.1 Alternative execution of a script

So far, a script has been a simple sequence of statements that are to be executed from the first one to the last one. Such scripts are called *sequential* or *linear*. The only way for them to diverge from this sequential execution is to crash, that is, to stop before the last statement!

In many situations, there is a need to change the way the statements are executed depending on some conditions at run time. Script 7.1 asks the user whether table DETAIL must be emptied, and if the user agrees, by typing character `1`, SQL statement `delete from` DETAIL is executed. If the user answers *no*, by typing character `0`, this statement is skipped and the table is left unchanged.

```
openDB ORDERS.db;
ask del = Delete all DETAIL rows (1/0);
if ($del$ = 1) delete from DETAIL;
```

**Script 7.1 -** An elementary **if** statement

The **if** statement has two parts. The first one is a **condition** (`$del$ = 1`) expressed between parentheses. The second one is the **statement** to be executed if the condition evaluates to *true* (**delete** `from DETAIL`). This statement constitutes the *true* branch of the **if** statement.

To submit more than one statement to the condition, we use an **if-endif** block structure, as in script 7.2. If the condition evaluates to *true*, i.e., if the value of **del** is **1**, all the statements between statements **if** and **endif**, are executed[1]. This sequence is the *true branch* of the **if** structure. In this script, **if** `(...);` and **endif**`;` are statements on their own and are terminated by character **;**.

Leading spaces in the *true* branch are used to make the script more readable but they have no meaning at run time.

```
openDB ORDERS.db;
ask del = Delete all DETAIL rows (1/0);
if ($del$ = 1);
   select * from DETAIL;
   delete from DETAIL;
   select * from DETAIL;
endif;
closeDB;
```

**Script 7.2 -** Multi statement **if-endif** block structure [Chapter-07-01.sql]

If the user answers **1** to the dialogue box, the output window shows the following result:

```
+-------+--------+------+
| OrdID | ProdID | Qord |
+-------+--------+------+
| 30178 | CS464  | 25   |
| 30179 | PA60   | 20   |
| 30179 | CS262  | 60   |
| 30182 | PA60   | 30   |
| 30184 | CS464  | 120  |
| 30184 | PA45   | 20   |
| 30185 | PA60   | 15   |
| 30185 | PS222  | 600  |
| 30185 | CS464  | 260  |
| 30186 | PA45   | 3    |
| 30188 | PA60   | 70   |
```

1. Needless to worry: no rows will be harmed in this script, since the updates are not committed.

```
| 30188 | PH222  | 92   |
| 30188 | CS464  | 180  |
| 30188 | PA45   | 22   |
+-------+--------+------+

+-------+--------+------+
| OrdID | ProdID | Qord |
+-------+--------+------+
+-------+--------+------+
```

## 7.2 Skipping statements

Script 7.2 can be written in another way. It is surely less elegant but it will allow us to introduce new concepts.

Script 7.3 includes two new statements, **goto** and **label**. Statement **label** SKIP-DELETE (line 2) merely defines a point in the script, between two statements, just as does a *bookmark* between two pages in a book. A *label* is attached to this point (SKIP-DELETE) so that it can be referred to from elsewhere. Its execution does nothing (anyway, nobody reads bookmarks).

The second new statement is also quite simple: **goto** SKIP-DELETE (line 1). When executed, it just does what it tells: the execution continues at the point labelled SKIP-DELETE. If the user enters value **0**, condition ($del$ = 0) evaluates to *true* and statement **goto** SKIP-DELETE is executed. Its effect is to jump to label SKIP-DELETE, skipping the next four statements, after which the statements that follow the label are executed. If, on the contrary, another value (say **1**) is entered by the user, then statement **goto** SKIP-DELETE is not executed and the select-delete-select sequence is executed, deleting the rows of table DETAIL.

```
   openDB ORDERS.db;
   ask del = Delete all DETAIL rows (1/0);
   if ($del$ = 0) goto SKIP-DELETE;                          [1]
      select * from DETAIL;
      delete from DETAIL;
      select * from DETAIL;
label SKIP-DELETE;                                           [2]
   closeDB;
```

**Script 7.3 -** Skip statements with a **goto** statement [Chapter-07-02.sql]

## 7.3 *If-else-endif* block structure

Now, let us go back to the **if-endif** construct. It allows an interesting variant in which the *True branch* (the statements to execute when the condition evaluates to *True*) is complemented by a *False branch* (the statements to execute when the condition evaluates to *False*) introduced by statement **else**.

In Script 7.4, the *True branch* is reduced to statement `showMessage The value is 0` and the *False branch* comprises statement **`showMessage`** `The value is 1`. Statement `showMessage` suspends the execution of the script and opens a message box.

If the *condition* is evaluated to *true*, the *true branch* is executed while if the *condition* is evaluated to *False*, the *False branch* is executed. The **`if`** statement, the *True branch*, the **`else`** statement, the *False branch* and the **`endif`** statement form an **`if-else-endif`** block structure.

```
ask A = Enter 0 or 1;
if ($A$ = 0);
   showMessage The value is 0;
else;
   showMessage The value is 1;
endif;
```

**Script 7.4 -** A full *if-else-endif* construct

The way Script 7.4 analyzes the answer of the user is sketchy since it relies on the user entering 0 or 1 only. What if -1 or 2 were entered instead? Script 7.5 is better at reporting on the value entered, warning the user when the value entered is invalid. It shows that **`if`** statements can appear in the *True* and *False* branches of a parent **`if`** structure.

```
ask A = Enter number 0 or 1;
if ($A$ = 0);
   showMessage The value is 0;
else;
   if ($A$ = 1);
      showMessage The value is 1;
   else;
      showMessage Value $A$ is invalid;
   endif;
endif;
```

**Script 7.5 -** A more rigorous analysis of values entered [Chapter-07-03.sql]

This structure is fine for two values but it will quickly become unreadable when more values are to be processed. Let us examine solutions for three values, that can be extended to an arbitrary value sets.

Script 7.6 uses a symmetric structure to analyze each possible value through a similar sequence. The remaining case is that of invalid values of **`A`**. Despite its simplicity, it exhibits a serious drawback: whatever the value of **`A`**, all the **`if`** statements are executed. Though this is a minor problem in this simple script, it may become more serious in more complex situations, notably because this structure is error prone and inefficient.

Script 7.7 is still another way to process these values in a symmetric way. It combines **if** and **goto** statements. Here, when the value has been processed, all the following statements are skipped.

But wait, we are not done yet! Script 7.8 is a quite interesting variant, that uses no **if-endif** block (except to check the validity of the value) and in which only the statements devoted to the current value are processed. This structure is both clear and efficient, specially when many values must be processed.

```
ask A = Enter a number between 0 and 2;
if ($A$ = 0) showMessage The value is 0;
if ($A$ = 1) showMessage The value is 1;
if ($A$ = 2) showMessage The value is 2;
if ($A$ not between 0 and 2) showMessage Value $A$ is invalid;
```

**Script 7.6 -** A symmetric but inefficient way to cope with several values of **A**

```
ask A = Enter a number between 0 and 2;
if ($A$ = 0);
   showMessage The value is 0;
   goto END-TEST;
endif;
if ($A$ = 1);
   showMessage The value is 1;
   goto END-TEST;
endif;
if ($A$ = 2);
   showMessage The value is 2;
   goto END-TEST;
endif;
showMessage Value $A$ is invalid;
label END-TEST;
```

**Script 7.7 -** A general symmetric structure to cope with several values of **A** [Chapter-07-03bis.sql]

The idea is to build a series of similar value processing sequences each headed by a significant label (e.g., **label** Process1), and terminated by a jump to the end of the script (**label** END-TEST). Jumping to the appropriate label is done by statement **goto** Process$A$, that, when preprocessed, becomes **goto** Process0, **goto** Process1 or **goto** Process2.

A last remark: symmetric scripts 7.7 and 7.8 both jump to the exit label **END-TEST** when the current value has been processed. Indeed, in many case, further

processing is needed before the end of the script. If this is not the case, then we can replace all **goto** END-TEST by statement **stop**, and get rid of the ending label.

```
ask A = Enter a number between 0 and 2;
if ($A$ not between 0 and 2);
   showMessage The value is invalid;
   goto END-TEST;
endif;
goto Process$A$;
label Process0;
   showMessage The value is 0;
   goto END-TEST;
label Process1;
   showMessage The value is 1;
   goto END-TEST;
label Process2;
   showMessage The value is 2;
   goto END-TEST;
label END-TEST;
```

**Script 7.8 -** A clear and efficient structure to process many values [Chapter-07-03bis.sql]

Generally, using **goto** statements is not recommended since they tend to make the script confusing[2]. They are justified here because *all of them* are used only to *escape* the normal flow of execution in a clean and systematic way.

## 7.4 Better controlling user values

Though they have been improved by addressing the case of invalid values, Scripts 7.5, 7.6, 7.7 and 7.8 still are insecure. Indeed, what happens when the user enters, say, `qwerty` or `2016-11-29` instead of 0, 1, 2 or another valid number?

Script 7.5 simply terminates with an **error message** declaring that condition **qwerty = 0** *cannot be evaluated* (we will see in the next section why this condition makes no sense for SQLfast).

We can make the script more robust in several ways. The first one checks whether the value of **A** is a numeric value before processing it with function **isNumeric()**, which tells whether its argument is a number or a numeric character string. This is shown in Script 7.9

2. Intricate programs with many forward and backward goto's are qualified *spaghetti-like*!

Another technique consists in considering the value of **A**, not as a *number*, but as a *character string*. This is what Script 7.10 does. Now, condition **'$A$' = '0'** becomes the valid condition **'qwerty' = '0'** after substitution, which evaluates to *false*.

```
ask A = Enter number 0 or 1;
if (not isNumeric('$A$'));
   showMessage Value $A$ is not numeric;
else;
   if ($A$ = 0);
      showMessage The value is 0;
   else;
      if ($A$ = 1);
         showMessage The value is 1;
      else;
         showMessage Value $A$ is invalid;
      endif;
   endif;
endif;
```

**Script 7.9 -** A robust version of Script 7.5 in which non numeric values are rejected

```
ask A = Enter number 0 or 1;
if ('$A$' = '0');
   showMessage The value is 0;
else;
   if ('$A$' = '1');
      showMessage The value is 1;
   else;
      showMessage Value $A$ is invalid;
   endif;
endif;
```

**Script 7.10 -** A robust version of Script 7.5 in which numbers are seen as character strings [Chapter-07-03ter.sql]

## 7.5 Conditions

Let us have a look at the way conditions must be written. Before statement **if** ($A$ = 0) is executed, it is first transformed through variable/value substitution that replaces expression **$A$** by the value of variable **A**.

If we suppose that variable **A** contains value **1**, this statement becomes **if** (1 = 0), so that the *True* branch will be skipped.

We also have observed that, when the user enters value **qwerty**, the `if` statement is transformed into `if (qwerty = 0)`, which is considered invalid by the SQLfast engine.

If, on the contrary, the `if` statement is written `if ('$A$' = '0')`, it becomes, after substitution, respectively, `if ('1' = '0')` and `if ('qwerty' = '0')`, which are two valid conditions.

The rule is simple: the *condition* part of an `if` statement, once variable/value substitution has been applied, must follow the **SQL syntax** of conditional expressions.

So, the following SQLfast `if` statements are valid:

```
if ($A$ < $B$);
if ($A$ = 2 * $B$ + 1);
if ('$answer$' = 'Y' and $N$ < 100);
if (not ('SQLdiag' = 'OK') or $I$ > $max$);
if (upper('$cat$') like 'B%');
if ('$city$' in ('London','Paris','Madrid'));
if ($age$ between 18 and 65);
```

Since SQLfast conditions actually are SQL conditions, we must address the situations in which character strings may include single-quotes, a point that has been discussed in Section 4.5. For example, if variable `name` contains value `O'Hara`, the statement

```
if ('$name$' = 'O''Hara')
```

will fail because it will be transformed into the invalid form

```
if ('O'Hara' = 'O''Hara')
```

It must be writen with the special substitution operator `§ §`:

```
if ('§name§' = 'O''Hara')
```

## 7.6 If, goto, labels and variables

All SQLfast statements may include variable references, but one, the `label` statement. This is due to the fact that, before a script is executed, the SQlfast engine parses it in order to extract all its labels and to store them in a dictionary. Therefore, all the labels of a script must exist before the script is executed.

So, the following script is **not valid**:

```
   set lab = DELETE;
   if ($X$ > $Y$) goto DELETE;
   update ...
label $lab$;
   delete ...
```

while the following is **correct**:

```
   set lab = DELETE;
   if ($X$ > $Y$) goto $lab$;
   update ...
label DELETE;
   delete ...
```

## 7.7 Complement: more on *goto* and *label* statements

**Warning**.

*This section includes technical material that **is not required** to write standard scripts. It is intended to advanced readers and should be skipped in a first reading.*

In this section, we study some advanced uses of the **goto-label** construct to develop *loops*, that is, sequences of statements that are repeated for each element of a series of values or while a condition is satisfied. In standard scripts, loops are specified through **for-endfor** and **while-endwhile** block statements that will be studied in Chapters 8 and 9.

### 7.7.1 Looping and counting

The combination of statements **if**, **goto** and **label** opens the door to a new script structure, the *loop*. Script 7.11 includes a **goto NEXT** statement that sends the execution *back* in the script, allowing statements to be executed more than once. It defines a **loop**, that is, a sequence that is executed *while a certain condition is met* (`$N$ <= 10`), or, equivalently, until the converse condition is satisfied (here `$N$ = 11`). Its goal is not particularly interesting: writing numbers from 1 to 10 in the output window!

```
   set N = 1;
label NEXT;
   if ($N$ <= 10);
      write $N$;
      compute N = $N$ + 1;
      goto NEXT;
   endif;
```

**Script 7.11 -** Counting [Chapter-07-04.sql]

Script 7.12 is a variant of this counting loop. It uses an elementary **if** statement but requires two **labels** and two **goto** statements.

```
   set N = 1;
label NEXT;
   if ($N$ = 11) goto STOP;
   write $N$;
   compute N = $N$ + 1;
   goto NEXT;
label STOP;
```

**Script 7.12 -** Another way to count [Chapter-07-04.sql]

In most cases, a loop is made up of four parts: the *useful* statement (**write** `$N$`), the continuation (`$N$ <= 10`) or exit (`$N$ = 11`) condition, the initialization (**set** `N = 1`) and the advancing statements (**compute** `N = $N$ + 1`). Generally, there are several ways to build a loop, Scripts 7.11 and 7.12 just being two of them.

Note that the scripts developed above will generally be written more intuitively and more concisely as follows:

```
for N = [1,10] write $N$;
```

**Script 7.13 -** The third and best version!

We will examine this form in the next chapters.

### 7.7.2 Looping again

The **looping** structure we played with in the previous section will allow us to write very powerful scripts, that begin to look like *real programs*. Let us imagine a script that asks the user the ID of a customer, displays the data of this customer and keep executing this *ask-and-display* process until the user decides to stop it.

This *iterative* script could look like Script 7.14. Label `NEXT` (line 2) starts the body of the loop, in which

- the user is asked to enter a value of CustID (or **X** to stop iterations) (line 3)
- if the value entered is not **X** (line 4), then the data of the customer are written to the output window (line 5) and the process continues (line 6).

If the value entered is **X**, then the condition evaluates to *false*, statements (lines 5 and 6) are skipped and the loop terminates.

```
openDB ORDERS.db;
label NEXT;                                                    [2]
   ask C = [Enter customer number or X to exit] CustID;        [3]
   if ('$C$' <> 'X');                                          [4]
      select * from CUSTOMER where CustID = '$C$';             [5]
      goto NEXT;                                               [6]
   endif;
closeDB;
```

**Script 7.14 -** Consultation of customer data [Chapter-07-05.sql]

### 7.7.3 Reading the buttons of the dialogue box

Script 7.15 is a copy of Script 7.14 in which the condition evaluates system variable **`DIALOGbutton`**, whose value is the name of the button clicked on when the user closed the dialogue box.

```
openDB ORDERS.db;
label NEXT;
   ask C = [Enter customer number or Cancel to exit] CustID;
   if ('$DIALOGbutton$' = 'OK');
      select * from CUSTOMER where CustID = '$C$';
      goto NEXT;
   endif;
closeDB;
```

**Script 7.15 -** Consultation of customer data - Using the DIALOGbutton system variable [Chapter-07-06.sql]

In the next chapters, we will learn to write such loops in a much more elegant way, as suggested in Script 7.16.

```
ask C = [Enter customer number or Cancel to exit] CustID;
while ('$DIALOGbutton$' = 'OK');
   select * from CUSTOMER where CustID = '$C$';
   ask C = [Enter customer number or Cancel to exit] CustID;
endwhile;
```

**Script 7.16 -** A better (but premature) way to examine customer data

## 7.8 Summary

This short chapter provides important new concepts related to script structure.

- The **`if`** statement allows the conditional execution of a statement or of a series of statements.
- There are three variants of the **`if`** statement, elementary **`if`** statement, **`if-endif`** block and **`if-else-endif`** block.
- The elementary **`if`** statement comprises a *condition* and a *true* branch made up of one statement (which can itself be an elementary **`if`** statement). The *true* branch is executed if the *condition* evaluates to *true*.
- The **`if-endif`** block comprises a *condition*, a *true* branch formed of a series of statements (possibly other **`if`** statements and blocks) and closing statement **`endif`**. The *true* branch is executed if the *condition* evaluates to *true*.
- The **`if-else-endif`** block comprises a *condition*, a *true* branch formed of a series of statements (possibly other **`if`** statements and blocks), an **else** statement, a *false* branch formed of a series of statements (possibly other **`if`** statements and blocks) and closing statement **`endif`**. The *true* branch is executed if the *condition* evaluates to *true*, otherwise the *false* branch is executed.
- The *condition* part of an **`if`** statement, once variable/value substitution has been applied, must follow the SQL syntax of conditional expressions. In particular, character string values that may include single-quotes must be carefully treated.
- A **`label`** statement gives a name to a point in the script. This point can be referred to from a **`goto`** statement.
- When a **`goto`** statement is executed, the script execution continues at (the statements that follows) the label specified in this statement.
- **`goto`** statements must be used carefully and parsimoniously since their undisciplined usage produces complex and insecure scripts.
- **`label`** and **`goto`** statements can be used to build *loops*. However, SQLfast offers specific loop statements that must be preferred except in very special situations.
- **`label`** statements cannot include substitution operators.

# Chapter 8

# Reading data

**Objective**: This chapter studies more advanced scripts, in which rows returned from the database through a **`select`** query are processed one at a time. Processing is based on various forms of **loops**.

**Keywords**: indexed select, empty result set, null value, SQL for loop, elementary for loop, block for loop, embedded loops, array, list

**Statements and functions**: extract, #, for, endfor, group_concat, itemLen, item, itemIndex

**System variables and parameters**: SQLdiag

## 8.1 Can we process individual rows?

We have learned to get data from the database through just one **`select`** query.

This query is evaluated and its result, a set of rows, is written in a tabular form in the output window. In general, the result set comprises several rows. It is fine to show the contents of a table, but there is no means to process these data in the script: the link from the database and the output window is direct and *uninterruptible*. The way the data are displayed in the output window can be controlled to some extent through the *format variables*, as will be described in Chapter 16, but the data themselves cannot be transformed. The following example statement writes the data of the customers of Poitiers in the output window:

```
select CustID,Name from CUSTOMER where City = 'Poitiers';
```

To *examine* and *process* the data coming from the database, we must use a new SQLfast statement, **`extract`**, that stores column values of a selected row into target variables, as in the following example:

```
extract adr,cit = select Address,City
                  from   CUSTOMER where CustID = '$cid$';
write Customer $cid$: $adr$ in $cit$;
```

that writes:

```
Customer C400: 65, r. du Tertre in Poitiers
```

The SQLfast engine executes the query, then extracts the values of columns **Address** and **City**, and finally stores them in target variables **`adr`** and **`cit`**.

Once the data have been stored into those variables, we can use them in whatever way we want. However, the **`extract`** statement applies to SQL queries that return **one row only**. If the **`where`** clause happens to select several rows, only the first one will be returned, the other ones being ignored.

Let us, for instance, consider the following query, which, for `Cat = 'B2'`, returns four rows from our example database.

```
select Address,City from CUSTOMER where Cat = 'B2';
```

Clearly, writing:

```
extract cid,cit = select CustID,City
                  from CUSTOMER where Cat = 'B2';
write Customer $cid$ from $cit$;
```

does not help, since only the first row will be read and displayed:

```
Customer B332 from Genève
```

Before examining how we can process a series of rows, let us first have a look at two simple scenarios: extracting one (or a definite number of) row(s) from a sequence and receiving no row.

## 8.2 Forcing a query to return one row only

When the selection condition is such that several rows may be returned, we can tell the SQLfast engine to return one of them by specifying its position in the result set.

If we want to extract data from row number **8**, we just write:

```
select CustID,City
from   CUSTOMER order by CustID #8;
```

or

```
extract cid,cit = select CustID,City
                  from   CUSTOMER order by CustID #8;
```

This position can also be provided by a variable:

```
set pos = 8;
extract cid,cit = select CustID,City
                  from   CUSTOMER order by CustID #$pos$;
```

This form of statement is called **indexed select** and **indexed extract** statements. It should be clear that **indexed select** or **extract** statements make no sense without an **`order by`** clause!

## 8.3 Can we force a query to returns *two* rows?

Yes, we can. And even three rows! Indeed, the **`#`**`8` clause of the *indexed select* statement, which specifies **one** position, can be generalized to an *interval* of positions such as in **`#[2,6]`**, which means *from 2 to 6* (the square brackets specify a sequence of elements, here numbers 2, 3, 4, 5, 6). Script 8.1 writes data from the first **N** rows of the CUSTOMER table. As we can guess, the interval extension is not valid for the **`extract`** statement.

```
ask N = How many rows?;
select CustID,City from CUSTOMER order by CustID #[1,$N$];
```

**Script 8.1 - select** statement indexed by an interval [Chapter-08-01.sql]

Before execution, the interval is adjusted to make it comply with the actual size of the result set. Table of Figure 8.1 shows how the various forms of interval are transformed, if needed, for table CUSTOMER, which contains 16 rows. Symbol - means that no row will be returned and **`error`** denotes a syntax error.

| source interval | actual interval |
|---|---|
| [2,6] | [2,6] |
| [2,2] | [2,2] |
| [0,50] | [1,16] |
| [18,22] | - |
| [-2,3] | [1,3] |
| [10,2] | - |
| [10] | error |

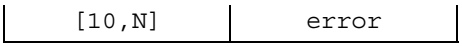



**Figure 8.1 -** Adjustment of the interval of an **indexed select** statement

## 8.4 Queries that return no row

The result set of a query may be empty. For example, there is no customer with **`CustID = 'X999'`** or there is less than 20 customers, so that row **`#20`** does not exist. In these cases, *the query returns no row.*

This means that it can be dangerous to use the contents of the target variables, as we did above, without checking first whether the query returned a row.

There is two ways to check this. The first one relies on the behavior of the **`extract`** statement, which empties the target variables if no row has been found. So, checking the value of variable **`adr`** (for example) can tell us whether a row was found or not. Script 8.2 shows how this information can be used.

```
extract adr,cit = select Address,City
                  from CUSTOMER where CustID = 'X999';
if ('$adr$' = '') write No such customer;
```

**Script 8.2 -** Checking data extraction failure through target variable contents

The second way uses system variable **SQLdiag**, that informs on how the execution of the last query completed. If a row has been found, its value is **OK**, otherwise its value is **NONE**.[1] This is illustrated in Script 8.3.

```
extract adr,cit = select Address,City
                  from CUSTOMER where CustID = 'X999';
if ('$SQLdiag$' = 'NONE') write No such customer;
```

**Script 8.3 -** Checking data extraction failure through examination of SQL diagnostic

However, these techniques are not always equivalent. We will refine them in Section 8.9.

## 8.5 Looping in the database

Now, let us go back to our primary objective: processing individual rows from a multirow result set.

1. There are other code values that will be examined later, such as **`SYNTAX`** for syntax errors.

For instance, we would like to write, in a personalized format, the Id and the city of the customers of category B2.

SQLfast includes a very powerful construct, the **for** statement, through which we can apply a statement to each row of a result set. It work as shown in Script 8.4.

This script fragment is made up of one statement, itself comprising two parts, the loop header and the loop body:

```
for cid,cit = [select CustID,City from CUSTOMER where Cat='B2']
   write Customer $cid$ from $cit$;
```

**Script 8.4 -** Processing a series of CUSTOMER rows - Using an elementary **for** loop

- the **loop header** specifies an SQL **select** query (`select CustID,City from CUSTOMER where ...`) and a list of target variables that receive the values returned by the query (`cid,cit`).
- the **loop body** is made up of *one* statement (`write Customer $cid$ from $cit$`); any statement is valid, except a **label** statement.

The SQL query returns a **sequence of rows** (square brackets denote a *sequence* of elements), each comprising two values. For each of these rows, the values are assigned to the two target variables then the body statement is executed. And so on, until all the rows of the result set have been processed.

This form of the **for** loop, called elementary **for** loop, allows single statement body only. When several statements must be applied to each row, we will use the **for-endfor** block statement instead. Script 8.5 extends Script 8.4 by adding statements that count the rows processed.

```
set I = 0;
for cid,cit = [select CustID,City from CUSTOMER where Cat='B2'];
   write Customer $cid$ from $cit$;
   compute I = $I$ + 1;
endfor;
write-b $I$ customers retrieved;
```

**Script 8.5 -** Processing a series of CUSTOMER rows - Using a **for-endfor** block [Chapter-08-02.sql]

The loop body is delimited by the *loop header* (**for**) and the (new) **endfor** statement and can now comprise more than one statement. Note that, in this structure, the loop header becomes a full statement in its own right, and is therefore terminated by a semi-colon symbol. The same applies to the **endfor** statement.

## 8.6 Embedded loops

Could the body of a **for** loop include another **for** statement? Sure. This is shown in Script 8.6. The external loop examines the customers of *Poitiers*. For each of them, the inner loop examines his orders, then for each order, writes the customer Id and the order Id.

```
for cid = [select CustID from CUSTOMER where City = 'Poitiers']
  for oid = [select OrdID from CUSTORDER where CustID = '$cid$']
    write Customer $cid$ placed order $oid$;
```

**Script 8.6 -** A single SQLfast statement composed of two embedded loops [Chapter-08-03.sql]

We notice that there is a tight link between both loops, through variable **cid**, set by the outer loop and used in the **where** clause of the inner loop.

Now, let us write a script that selects the customers that have at least one order, then, for each of them, lists these orders together with their total amount. This script must generate the following list.

```
Customer B512 from Toulouse
         placed order 30188 on 2014-01-03 for 69720 €

Customer C400 from Poitiers
         placed order 30179 on 2013-12-22 for 6400 €
         placed order 30184 on 2013-12-23 for 28500 €
         placed order 30186 on 2014-01-02 for 315 €

Customer F011 from Toulouse
         placed order 30185 on 2014-01-02 for 169625 €

Customer K111 from Lille
         placed order 30178 on 2013-12-21 for 5500 €

Customer S127 from Namur
         placed order 30182 on 2013-12-23 for 2850 €
```

Clearly, we need a **for-endfor** block loop that allows multi-statement body. A first **for-endfor** loop examines the customers who have placed at least one order:

```
for cid,cit = [<the customers who have at least one order>];
   write-b Customer $cid$ from $cit$;
   <process the orders of customer $cid$>;
endfor;
```

A second, inner loop extracts the orders of the current customer (Script 8.7).

```
for cid,cit = [<the customers who have at least one order>];
   write-b Customer $cid$ from $cit$;
   for oid,dat,am = [<the orders of customer $cid$>]
       write @t placed order $oid$ on $dat$ for $am$ €;
   endfor;
endfor;
```

**Script 8.7 -** Skeleton of the *customer-order data extraction* script

Now, we define the set of customers through an SQL **select** query:

```
<the customers who have at least one order>
  ≡ select CustID,CITY from CUSTOMER
    where  CustID in (select CustID from CUSTORDER)
```

Same exercise for the orders of the current customer:

```
<the orders of customer $cid$>
  ≡ select O.OrdID,DateOrd,sum(Qord*Price)
    from   CUSTORDER O,DETAIL D,PRODUCT P
    where  CustID = '$cid$'
    and    D.OrdID = O.OrdID
    and    D.ProdID = P.ProdID
    group by O.OrdID,DateOrd
```

By introducing these queries into the skeleton, we get Script 8.8[2]:

```
for cid,cit = [select CustID,City from CUSTOMER
               where CustID in (select CustID from CUSTORDER)];
   write-b Customer $cid$ from $cit$;
   for oid,dat,am = [select O.OrdID,DateOrd,sum(Qord*Price)
                     from   CUSTORDER O,DETAIL D,PRODUCT P
                     where  CustID = '$cid$'
                     and    D.OrdID = O.OrdID
                     and    D.ProdID = P.ProdID
                     group by O.OrdID,DateOrd];
       write @t placed order $oid$ on $dat$ for $am$ €;
   endfor;
endfor;
```

**Script 8.8 -** The full script for *customer-order data extraction* [Chapter-08-04.sql]

2. Since the body of the inner loop includes one statement only, it could have been written as an elementary **for** loop.

## 8.7 A more elegant script

**Warning**. *This section is optional and can be skipped in a first reading.*

The SQL **select** queries being fairly large, they make the double-loop structure of script 8.8 a bit difficult to read. We will use the *variable substitution* mechanism of SQLfast to simplify this script. First, we define two variables containing each a data description as an SQL query (Script 8.9). Note that the verb **select** is not part of the query, as will be explained in Chapter 9.

```
set Customers = CustID,City from CUSTOMER
                where  CustID in
                       (select CustID from CUSTORDER);
set-n Orders =  O.OrdID,DateOrd,sum(Qord*Price)
                from   CUSTORDER O,DETAIL D,PRODUCT P
                where  CustID = '$cid$'
                and    D.OrdID = O.OrdID
                and    D.ProdID = P.ProdID
                group by O.OrdID,DateOrd;
```

**Script 8.9 -** Encapsulating the queries into variables

Since the second query includes a substitution operator, we use a **set-n** statement instead of **set** to preserve this operator. Then, we inject these variables in the script in place of the queries (Script 8.7).

```
set Customers = CustID,CITY from CUSTOMER ...;
set-n Orders =  O.OrdID,DateOrd,sum(Qord*Price) ...;

for cid,cit = [select $Customers$];
   write-b Customer $cid$ from $cit$;
   for oid,dat,am = [select $Orders$]
       write @t placed order $oid$ on $dat$ for $am$ €;
   endfor;
endfor;
```

**Script 8.10 -** Another version of the full script for *customer-order data extraction* [Chapter-08-05.sql]

## 8.8 SQLfast arrays revisited

SQLfast does not include the concept of *array*, that quite often is a poor implementation of sets. Indeed, relational tables and the SQL language provide a much powerful and flexible technique to represent and manipulate sets, multisets, either ordered or unordered. However, arrays may be convenient in some circumstances, as

illustrated in Chapter 6, where they were simulated through appropriate name patterns.

Actually, SQLfast provides a concept that may also be used to simulate arrays, namely *list* character strings.

A *list* is a mere character strings built as the concatenation of elements separated by definite symbols:

```
Paris;London;New York;Genève
122;0;0.55;-451
C1;;B1;B2;
```

Lists are easy to build (statement `compute` remove the leading separator):

```
set numbers = ;
for I = [1,10] set numbers = $numbers$;$I$;
compute numbers = substr('$numbers$',2);
```

which gives:

```
numbers = 1;2;3;4;5;6;7;8;9;10
```

SQL aggregate function **`group_concat`** can also be used to build lists from a database:[3]

```
extract cust = select group_concat(CustID,';')
               from   CUSTOMER where City = 'Toulouse';
```

which gives:

```
cust = B512;C003;K729;F011;D063
```

They are also quite easy to process, though three specific functions `item`, `itemLen` and `itemIndex`:

- `itemLen(s,sep)`: considering string `s` as a list of elements separated by string `sep`, returns the number of elements in `s`
- `item(s,i,sep)`: considering string `s` as a list of elements separated by string `sep`, returns the `i`th elements of `s`
- `itemIndex(s,e,sep)`: considering string `s` as a list of elements separated by string `sep`, returns the position of first element `e`. Returns `0` if `e` is absent.
- `itemLenS(s)`, `itemS(s,i)`, `itemIndexS(s,e)`: same as functions above with the default separator `sep = ;`.

Some examples:

```
compute cusL = itemLenS('$cust$');
```

3. `group_concat` is a very powerful SQL function that will be detailed later.

```
compute cus3 = itemS('$cust$',3);
compute cusX = itemIndexS('$cust$','F011');
write $cusL$, $cus3$, $cusX$;
```

which gives:

```
5, K729, 4
```

## 8.9 Empty sets, null values and statistics

**Warning**. *This section may appear a bit tricky for a first reading. It may be so, but it describes a really important issue that must be coped with in many database application programs, whatever the programming language they are written in.*

Checking either *empty* result or system variable **`SQLdiag`** is quite handy to cope with empty result sets but some care must be paid when *statistical values* or *nullable column values* are asked for. We have seen in Section 8.4 that the two fragments of Script 8.2 and Script 8.3 are equivalent. They both extract the address and city of customer number `X999` and report on whether this customer does not exist in the database.

If we want to compute the *sum of account amounts* of the customers who live in a definite city, we can formulate the query as one of the fragments of Script 8.11. Let us suppose that this city is Madrid, and that we have no customer there. Surprisingly, despite an apparent similarity, this case is quite different from that of Section 8.4.

Indeed, the query returns a **non empty result**, which is a set containing a *null* value. The SQL engine tells us: *OK, I have a value for you, but it is* ***undefined*** *since it is calculated on an empty set*. So, the **`SQLdiag`** code is **`'OK'`**, but this *null* value is stored into the target variable, which therefore becomes empty. Here, checking **`SQLdiag`** is useless, so that only checking the emptiness of the target variable is effective.

```
-- Fragment 1 (doesn't work)
extract total = select sum(Account)
                from CUSTOMER where City = 'Madrid';
if ('$SQLdiag$' = 'NONE') write No customers in Madrid [1];

-- Fragment 2 (works fine)
extract total = select sum(Account)
                from CUSTOMER where City = 'Madrid';
if ('$total$' = '') write No customers in Madrid [2];
```

**Script 8.11 -** These fragments are not equivalent for (seemingly) empty sets. Only the second one is valid.

### SQL rules for empty result sets

When asked to extract the values of one **select** element **E** and to store it in variable **V** (i.e., extract **V** = select **E** from ...) the behavior of SQL depends on the nature of **E**. There are four simple rules:

– **E** denotes a *not null* value: *not null* column (**Name** for example) or a nullable column with a not null value (**Cat** = B1 for example):
  - a row was found: **V** is not empty and **SQLdiag** = OK;
  - no row was found: **V** is empty and **SQLdiag** = NONE;

– **E** denotes a *nullable* column and this column is *null* (**Cat** = *null* for example):
  - a row was found: **V** is empty and **SQLdiag** = OK;
  - no row was found: **V** is empty and **SQLdiag** = NONE;

– **E** denotes *statistical function* **count()**:[4]
  - a row was found: **V** > 0 and **SQLdiag** = OK;
  - no row was found: **V** = 0 and **SQLdiag** = OK;

– **E** denotes any other *statistical function* (**sum()** for example):
  - a row was found: **V** is not empty and **SQLdiag** = OK;
  - no row was found: **V** is empty and **SQLdiag** = OK;

If the *select list*[5] comprises more than one element, we must combine these rules. To check that an SQL query returns values from an empty set (so to speak), we distinguish five cases according to the composition of the select list.

1. The *select list* **includes no** *statistical functions* (count, sum, avg, max, min) and **includes** at least one *non nullable element*. Example:

```
select Name,Cat from CUSTOMER
where  CustID = 'X999';
```

   **Action**: check either SQLdiag or the emptiness of a target variable associated to a *non nullable* column.

2. The *select list* **includes no** *statistical functions* but **includes** only *nullable elements*. Example:

```
select Cat from CUSTOMER
where  CustID = 'X999';
```

   **Action**: check SQLdiag.

3. The *select list* **includes** *statistical functions* only. Example:

```
select sum(Account),count(*) from CUSTOMER
```

4. The result of statistical functions applied to an empty set is **0** for **count** and ***null*** for **sum**, **avg**, **max** and **min**, which translates into an empty string in target variables.
5. The list of elements that appear between **select** and **from** SQL words.

```
where  City = 'Madrid';
```

**Action**: check the value of any target variable (`SQLdiag` is always `OK`).

4. The *select list* **includes** *statistical functions* and **includes** at least one *non nullable element*. Example:

```
select City,sum(Account) from CUSTOMER
where  CustID = 'X999';
```

**Action**: check the value of a target variable associated to a *non nullable* column (`SQLdiag` is always `OK`).

5. The *select list* **includes** *statistical functions* and **includes** *nullable elements* only. Example:

```
select Cat,sum(Account) from CUSTOMER
where Cat = 'X9';
```

**Action**: check the value of a statistical target variable (`SQLdiag` is always `OK`).

Script 8.12 includes examples of these patterns. When executed on our example database, it produces the following results:

```
Case 1 - SQLdiag='NONE', Name='', Cat=''
Case 2 - SQLdiag='NONE', Cat=''
Case 3 - SQLdiag='OK', sum='', count='0'
Case 4 - SQLdiag='OK', City='', sum=''
Case 5 - SQLdiag='OK', Cat='', sum=''
Case 6 - SQLdiag='OK', Cat='', sum='-2250.0'
Case 7 - SQLdiag='OK', Cat='', sum=''
```

All this may seem a bit complicated but remember that SQL is very powerful and very precise, two invaluable qualities that have their price! Clearly, *null* values may make things tricky and sometimes produce non intuitive behaviors. It is better to avoid them when possible, at least in base tables stored in the database[6].

## 8.10 Summary

This chapter presents the following facts about data extraction from a database.

- The **result set** of an SQL **`select`** query is the set of rows returned by this query. A result set can be empty.

6. Unfortunately, avoiding nullable columns in the database tables wouldn't discard null values. Indeed, SQL itself may generate null values in some result sets as shown in Script 8.11.

- The **select** statement writes data extracted from the database but does not allow the script to examine and process individual rows from the result set.

```
-- column values absent from the database
set cid = X999;
set cit = Madrid;
set cat = X9;
extract N,C = select Name,Cat from CUSTOMER
              where CustID = '$cid$';
write Case 1 - SQLdiag='$SQLdiag$', Name='$N$', Cat='$C$';
extract C   = select Cat from CUSTOMER where CustID = '$cid$';
write Case 2 - SQLdiag='$SQLdiag$', Cat='$C$';
extract S,C = select sum(Account),count(*) from CUSTOMER
              where City = '$cit$';
write Case 3 - SQLdiag='$SQLdiag$', sum='$S$', count='$C$';
extract C,S = select City,sum(Account) from CUSTOMER
              where CustID = '$cid$';
write Case 4 - SQLdiag='$SQLdiag$', City='$C$', sum='$S$';
extract C,S = select Cat,sum(Account) from CUSTOMER
              where Cat = '$cat$';
write Case 5 - SQLdiag='$SQLdiag$', Cat='$C$', sum='$S$';
extract C,S = select Cat,sum(Account) from CUSTOMER
              where Cat is null;
write Case 6 - SQLdiag='$SQLdiag$', Cat='$C$', sum='$S$';
delete from CUSTOMER where Cat is null;
extract C,S = select Cat,sum(Account) from CUSTOMER
              where Cat is null;
write Case 7 - SQLdiag='$SQLdiag$', Cat='$C$', sum='$S$';
```

**Script 8.12 -** Experimenting with empty sets [Chapter-08-06.sql]

- Statement **extract** selects one row from a result set and stores the elements of this row in target variables. These variables are empty if there is no row in the result set. In addition, system variable **SQLdiag** indicates whether the result set is empty (value **NONE**) or not (value **OK**).
- Expression **#n** (where **n** denotes an integer number > 0) can be appended to a **select** or **extract** statement to specify the rank **n** of the row to extract from the result set.
- Expression **#[n1,n2]** (where **n1** and **n2** denote two positive integer number) can be appended to a **select** statement to specify the *closed* or *semi-open* interval of the ranks of the rows to extract from the result set.
- Statement **for** defines an elementary loop composed of a *loop header* and a *loop body*. The *loop header* specifies a **select** query and a series of target variables.

The body comprises one SQLfast statement. For each row of the result set, its elements are assigned to the target variable and the body is executed.

- The **`for-endfor`** block loop is an extension of the elementary **`for`** statement to multi-statement body. The end of the body is indicated by statement **`endfor`**.
- The body of a **`for`** loop can be another **`for`** loop, a structure called *embedded loop*.
- When **long queries** are integrated into scripts, the latter can be made more readable if we store the queries in variables, then we replace these queries by these variables in the script. *Warning*: the word **`select`** must be part of the loop header and not of the query stored in the variable.
- If the select list of an **`extract`** statement is a **`statistical function`**, the result set is always *non empty* even if no row satisfies the **`where`** condition. (`SQLdiag` = `OK`). In this case, one must check the returned value of this function, which is `0` for function **`count()`** and an empty string for the other statistical functions.
- If the select list of an **`extract`** statement is made up of several elements (nullable columns, non nullable columns, statistical functions), checking whether the set of rows from which the returned values are computed is empty or not must be done according to strict rules.

# Chapter 9

# Looping - The final chapter

**Objective**: This chapter studies the various forms of **loops** and some other statement that can be used to modify the behavior of loops.

**Keywords**: *SQL* for loop, *range* for loop, *file* for loop, *list* for loop, *while* loop, loop modifier

**Statements and functions**: for, endfor, while, next, exit, goto, (True), openTextFile, nextTextLine, closeTextFile

**System variables and parameters**: SQLdiag, FILEdiag

## 9.1 Introduction

Chapter 8 gave us a first insight on the concept of loop. A loop appears as a sequence of operation (the *loop body*) that must be executed for each element of a sequence of rows, defined by the *loop header.*

In this chapter, we will examine four other forms of loop: the **range `for` loop**, the **file `for` loop**, the **list `for` loop** and the **`while` loop**. We will also study two loop modifiers: **`next`** and **`exit`**.

## 9.2 SQL `for` loop

To make this chapter cover all the loop patterns, we first recall the principles of the **`for`** loop studied in Chapter 8. Since it works on sequences of rows extracted by SQL queries, we will call it **SQL for loop**. Script 9.1 gives an example of an elementary *SQL for loop* and its expanded version as a **`for-endfor`** block statement. The block **`for-endfor`** construct allows the body part to comprise more than one statement.

We verify that,

- the query is enclosed in square brackets,
- the *select list* of the query has the same length as the *target variable list*.

```
for cid,nam = [select CustID,Name from CUSTOMER where Cat ='B1']
               write $cid$,$nam$;

for cid,nam = [select CustID,Name from CUSTOMER where Cat='B1'];
   write $cid$,$nam$;
endfor;
```

**Script 9.1 -** Elementary SQL **for** loop and its equivalent **for-endfor** block statement [Chapter-09-01.sql]

In Section 8.3, we have seen that a **`select`** statement can be complemented by an index clause (**`#`**), specifying the position of the rows returned in the result set, either as a single rank (**`#8`**), or as an interval (**`#[2,6]`**). This form can also be used in SQL **`for`** loops, as illustrated in Script 9.2.

```
for cid,nam = [select CustID,Name from CUSTOMER
               order by CustID #[2,6]];
   write $cid$,$nam$;
endfor;
```

**Script 9.2 -** SQL **for** loop with an index clause [Chapter-09-02.sql]

## 9.3 Range `for` loop

The range loop is quite similar to the SQL loop, except for the nature of the sequence of elements, which is a **series of numbers**. In the simplest form of the loop, the sequence comprises integer numbers and is defined by its first and last numbers (e.g., **`[0,10]`**). Both scripts of Script 9.3 writes numbers 0, 1, 2, ..., 10 in the output window.

```
for I = [0,10] write $I$;

for I = [0,10];
   write $I$;
endfor;
```

**Script 9.3 -** Elementary range **for** loop and its equivalent **for-endfor** block statement [Chapter-09-03.sql]

In the second form, the sequence is defined by the smallest number, the upper limit and the increment. It is made up of the numbers starting from the smallest one up to the largest number that is not larger than the upper limit. The numbers of the sequence can be integer or real numbers. Script 9.4 will write numbers 2.5, 3.1, 3.7, 4.3 and 4.9 in the output window.

```
set min =  2.5;
set max =  5;
set step = 0.6;
for I = [$min$,$max$,$step$] write $I$;
```

**Script 9.4 -** Range **for** loop, second form [Chapter-09-03.sql]

A range loop can also be used to read rows of selected positions (Script 9.5).

```
for I = [2,10,2];
   extract cid,nam = select CustID,Name from CUSTOMER
                     order by CustID #$I$;
   write Customer #$I$: $cid$, $nam$;
endfor;
```

**Script 9.5 -** Reading data with a range **for** loop [Chapter-09-04.sql]

The form **`[2,10]`** defines a *closed interval*, in which the first and last number belongs to the sequence.

## 9.4 File `for` loop

If an external file is made up of a sequence of lines, a **file `for` loop** can be used to read and process them, one at a time (Script 9.5). A line is any character string followed by a *new line* character. However, this last character is not part of the line.

```
set N = 0;
for line = [file inpuText.txt];
   compute N = $N$ + 1;
   write line $N$: $line$;
endfor;
write The file contains $N$ lines;
```

**Script 9.6 -** Reading and displaying the successive lines of a text file with a file **for** loop

## 9.5 List `for` loop

This particular loop structure is based on *SQLfast lists*, that are studied in detail in Chapter 24. An SQLfast list is a character string built as a sequence of values separated by a special symbol (just like a CSV record). Example:

```
London;Paris;Genève;Toulouse
```

An SQLfast list can be used as a sequence of elements in a loop (Script 9.7).

```
for city = [items London;Paris;Genève;Toulouse] write $city$;

for city = [items London;Paris;Genève;Toulouse]
           select * from CUSTOMER where City = '§city§';
```

**Script 9.7 -** Two representative uses of a List **for** loop

## 9.6 `While` loop

A **`while`** loop executes its body as long as a definite condition is satisfied. This body may be executed zero, one or several times. In Script 9.8, the user is asked to enter a number in range [1,5]. If (and while) the number entered does not fall in this range, or if the user refuses to answer (by clicking on button **`Cancel`**), the dialog box stubbornly appears again.[1] This is an example of *elementary* **`while`** loop, that has a single statement body.

1. *Beware*: this script does not prevent the user to enter no value or a non-numeric value. See Script 7.9 that addresses this problem, at least partially.

```
ask N = [Enter a number from 1 to 5] Number:;
while ($N$ not between 1 and 5 or '$DIALOGbutton$' <> 'OK')
   ask N = [/bError! Enter a number from 1 to 5] Number:;
showMessage At last, a valid number: $N$;
```

**Script 9.8 -** Forcing the user to enter a valid number through a **while** loop [Chapter-09-05.sql]

We notice that this script requires two instances of statement **ask**, one to prepare the first iteration of the loop and the other one to prepare the next iteration. Section 9.9 will show that we can avoid that.

Similarly to **for** loops, the **while-endwhile** block allows for multiple statement body. Script 9.9 includes two equivalent code snippets that measure the number of iterations executed in one second.

Script 9.10 has the same structure as Script 9.8. It allows the user to update the data of a series of customers. For each of them, the script asks the user to identify her and to enter new values for columns Address and City, then it updates the row of this customer. The process is stopped when the user does not fill the Customer ID field or when he clicks on button **Cancel**.

```
set I = 0;
start-timer;
while ($timer$ <= 1.0) compute I = $I$ + 1;
write number of iterations in 1 second: $I$;

set I = 0;
start-timer;
while ($timer$ <= 1.0);
   compute I = $I$ + 1;
endwhile;
write Number of iterations in 1 second: $I$;
```

**Script 9.9 -** Elementary **while** loop and its equivalent **while-endwhile** block statement (*btw*, to measure the performance of SQLfast!) [Chapter-09-06.sql]

```
ask cid,adr,city = [Update customer data]
    Customer ID:|New address:|New city:;
while ('$cid$' <> '' and '$DIALOGbutton$' <> 'Cancel');
   update CUSTOMER
    set    Address = '$adr$',
           City    = '$city$'
    where  CustID  = '$cid$';
   ask cid,adr,city = [Update customer data]
       Customer ID:|New address:|New city:;
endwhile;
```

**Script 9.10 -** Updating customer data through a while loop [Chapter-09-07.sql]

## 9.7 Forced loop iteration

In the body of a **for** or **while** block statement, **next** forces the loop to stop the current iteration (skipping the remaining statements) and to execute the next one. Script 9.11 shows one way to compute the multiples of a number entered by the user. This is not the simplest way to proceed, but it demonstrates the use of the **next** statement. The loop generates the successive numbers from 2 to 50. When expression **N % d** (symbol **%** represents function modulo) evaluates to 0, then **d** is an exact divisor of **N**, or **N** is a multiple of **d**. In this case a message is written to announce that the next multiple of **d** has been found. Otherwise, the next value of **N** is generated (**next**).

```
ask d = Enter divisor (2-50);
for N = [1,50];
   if ($N$ % $d$ > 0) next;
   write Next multiple of $d$: $N$;
endfor;
```

**Script 9.11 -** Finding multiples of a number

## 9.8 Forced loop exit

In the body of a **for** or **while** block statement, **exit** forces the termination the loop. Script 9.12 is a first example of it.

## 9.9 While (True) loop

When the condition of a **while** loop is replaced by constant **True**, the loop executes its body forever. To stop these iterations, we must use **exit** statements, generally placed in True or False branches of **if** statements. Script 9.12 is another version of Script 9.8, which asks the user to enter a number between 1 and 5. There is only one way to terminate the loop: entering a valid number **and** clicking on button **OK**. We observe that this variant uses one **ask** statement only.

```
while (True);
   ask N = [Enter a number from 1 to 5] Number:;
   if ($N$ between 1 and 5 and '$DIALOGbutton$' = 'OK') exit;
   showMessage Invalid data and/or button;
endwhile;
```

**Script 9.12 -** Forcing the user to enter a valid number through a **while** loop - version 2. [Chapter-09-08.sql]

The **while (True)** loop is particularly fitted to complex loops with several exit points or in which parts of the body must be skipped for more than one reason. Script 9.13 is an improved version of Script 9.10. It asks the user to identify a customer and to enter new values for columns Address and City, then it updates the row of this customer. Before updating the data, four conditions are evaluated:

- two on *process ending*: through button **Cancel** and through empty identification field,
- two on *data validation*: the customer must exist and address and city data must be non empty.

### Note

Constant **True** can only be used in this statement.

## 9.10 Correct loop structure

Russian dolls comprise a head and a body. Each of them, except the smallest one, can contain a smaller doll. We generally agree on the following obvious rule: *when the head of a doll is placed into another doll, its body is placed in the same doll*.

**for** and **while** loops obey a similar rule.[2]

2. If you find this analogy a bit weird, consider that *Russian dolls* obey the same rules as **for** and **while** loops.

```
openDB ORDERS.db;
while (True);
    ask cid,adr,cit = [Update customer data]
                      Customer ID:|New address:|New city:;
 -- Checking ending conditions
    if ('$DIALOGbutton$' = 'Cancel') exit;
    if ('$cid$' = '');
       showMessage OK, but please use the Cancel button to exit;
       exit;
    endif;
 -- Checking data validity
    extract nc = select count(*) from CUSTOMER
                 where  CustID = '$cid$';
    if ($nc$ = 0);
       showMessage Customer $cid$ doesn't exist. Please retry.;
       next;
    endif;
    if ('$adr$' = '' or '$cit$' = '');
       showMessage Data fields cannot be empty. Please retry.;
       next;
    endif;
 -- Updating data
    update CUSTOMER
    set    Address = '$adr$', City = '$cit$'
    where  CustID  = '$cid$';
endwhile;
commitDB;
closeDB;
```

**Script 9.13 -** User-friendly dialogue through **exit** and **next** statements [Chapter-09-09.sql]

Loops must be **embedded**. If a loop is written in the body of another loop, all its components (header, body and closing statement) must appear in this body. In other words, loops cannot overlap. For instance, invalid Script 9.14 will (fortunately) be rejected ("*Statement endwhile closing a ForRange block*").

This property also stands for `True` and `False` branches of **`if-endif`** block statements:

– all the components of a loop must appear in the same branch of **`if`** statements.

– all the components of an **`if`** (block) statement must appear in the same body of **`for`** and **`while`** loops.

```
while ($N$ < 10);
   for I = [1,$N$];
   if (...) exit;
   ...;
endwhile;
   endfor;
```

**Script 9.14 -** Invalid overlapping loops

## 9.11 Variables in loops and if statements

We have seen in Section 4.6 and 4.7 that some parts of an SQLfast statement can be defined as value of variables. This practice is quite usual for constants (`if ($I$ = 0) ...`) but some more structural parts may also be defined in this way. Indeed, we can write:

```
if ($condition$) $statement$;
```

provided variable **`condition`** contains a valid condition and variable **`statement`** has been assigned a syntactically correct statement. However, some complex statements are submitted to specific constraints[3]. The rules for **`while`**, **`for`** and **`if`** statements are the following.

1. ***`if`*** *statements*. The words **`if`**, **`else`** and **`endif`** must explicitly appear in the script. The *parentheses* enclosing the condition must be written explicitly.
2. ***`while`*** *statements*. The words and expression **`while`**, **`while (True)`** and **`endwhile`** must explicitly appear in the script.

   The *parentheses* enclosing the condition must be written explicitly.
3. ***`for`*** *statements*. The words **`for`** and **`endfor`** must explicitly appear in the script. The *square brackets* enclosing the sequence must be written explicitly. In case of an **SQL `for`** statement, the word **`select`** must appear explicitly. If the SQL query is indexed, than the index expression must appear explicitly (**`#$I$`** and **`#[$A$,$B$]`**). In case of a **file `for`** loop, the word **`file`** must appear explicitly. In case of a **range `for`** statement, the sequence expression or parts of it can be defined as variable values.
4. **`next`** and **`exit`** statements. They must explicitly appear in the script.

3. The reason is technical and is related to the way such statements are processed by the SQLfast engine. Loop constructs **`for`** and **`while`** are first analyzed and translated into simpler statements (`if`, `label`, `goto`, etc.). When the translation is complete, then the resulting script is executed. So, this translation is carried out **before** the statements of the script are executed, that is, before variables receive their values. Consequently, the parts of a statement that are defined as variable values cannot be known at translation time, so that such statements cannot be translated into the appropriate executable scripts.

5. A constant (8, C400, Poitiers) can always be defined as the value of a variable.

These constraints are automatically satisfied in practically all useful scripts.

## 9.12 How does a loop work, actually?

**Warning**. This section is reserved to experienced readers interested by the mechanics behind the **`while`** and **`for`** statements. It is useless for plain, and particularly novice, readers. It can (it should!) be skipped in a first reading.

In this section, we will provide some hints on how SQLfast understands and executes the main kinds of loops.

### 9.12.1 The *while (True)* loop

The **`while (True)`** is the simplest loop structure. However, it is able to express all the more complex patterns found in **`for`** and **`while`** block statements. It can be simulated very simply with **`label`** and **`goto`** statements (Script 9.15).

```
while (True);
   execSQL UpdateCustomers.sql;
   if $continue$ = 0 exit;
endwhile;
```

```
label NEXT;
   execSQL UpdateCustomers.sql;
   if $continue$ = 0 goto EXIT;
   goto NEXT;
label EXIT;
```

**Script 9.15 -** Expression of a typical **while (True)** loop with **label** and **goto** statements

### 9.12.2 The *while (condition)* loop

The standard version of the **`while`** loop can be reduced to a **`while (True)`** loop, as shown in Script 9.16.

```
while ($A$ < 10);
   execSQL DisplayInvoices.sql;
endwhile;
```

```
while (True);
   if not ($A$ < 10) exit;
   execSQL DisplayInvoices.sql;
endwhile;
```

**Script 9.16 -** Expression of a typical **while (True)** loop with **label** and **goto** statements

### 9.12.3 The *SQL for* loop

The SQL **for** statement can be expressed by a **while ()** loop. Statement **SQLfor_header** initializes the row extraction with the SQL query and statement **SQLfor_fetch** reads the next row (Script 9.17).

```
for cid = [select CustID from CUSTOMER where City = '$C$'];
   write $cid$;
endfor;
```

```
SQLfor_header select CustID from CUSTOMER where City = '$C$';
while ('$SQLdiag$' = 'OK');
   SQLfor_fetch cid;
   if ('$SQLdiag$' = 'OK');
      write $cid$;
   endif;
endwhile;
```

**Script 9.17 -** Expression of an SQL **for** loop

### 9.12.4 The *file for* loop

The **file for** statement can be expressed by a **while (True)** loop.

The lines are extracted through index **I** starting from 1 and incremented (`file-Tovariable ... #$I$` reads the $I^{th}$ lines of the file) until no line can be found any more (`'$SQLdiag$' <> 'OK'`) (Script 9.18).

```
for line = [file inpuText.txt];
   write $line$;
endfor;
```

```
set I = 1;
while (True);
   fileToVariable line = inpuText.txt #$I$;
   if ('$FILEdiag$' <> 'OK') exit;
   write $line$;
   compute I = $I$ + 1;
endwhile;
```

**Script 9.18 -** Naive expression of a **file for** loop

However, this technique gives poor performance when the file contains more than a few hundreds of lines. A much better approach relies on three file reading statements:

– **`openTextFile`**: opens the specified file

– **`nextTextLine`**: read the next line in the specified file

– **`closeTextFile`**: close the specified file.

This improved translation is shown in Script 9.19. This is the way SQLfast executes **file for** loops.

```
openTextFile inpuText.txt;
while (True);
   nextTextLine line = inpuText.txt;
   if ('$FILEdiag$' <> 'OK') exit;
   write $line$;
endwhile;
closeTextFile inpuText.txt;
```

**Script 9.19 -** An efficient translation of the **file for** loop

### 9.12.5 The *range for* loop

Similarly, the range **`for`** statement can be expressed by a **`while (True)`** loop. The loop variable **`I`** is incremented (by 1 or by the specified step) until it becomes greater than the maximum value (`$I$ > 10`) (Script 9.20).

```
for I = [1,10];
   write $I$;
endfor;
```

```
set I = 1;
while (True);
   if ($I$ > 10) exit;
   write $I$;
   compute I = $I$ + 1;
endwhile;
```

**Script 9.20 -** Expression of an range **for** loop

## 9.13 Summary

This chapter describes the various kinds of loops available in SQLfast. In particular, it describes the following principles.

- **for** loops are based on a set of one or several target variables and on a sequence of elements specified by an expression into square brackets.
- A loop, the body of which comprises one statement only, can be written as an *elementary* loop.
- The sequence of an *SQL* **for** loop is the (ordered) result set of an SQL query. This query can be indexed through the specification of a position (`#8`) or of an interval of positions (`#[1,8]`).
- The sequence of a *file* **for** loop comprises the lines of a text file. It can be simulated with statements **openTextFile**, **nextTextLine** and **closeTextFile**.
- The sequence of a *range* **for** loop is a suite of successive numbers. If no increment value is specified (`[1,8]`), the default increment (or step) is 1. The values of the suite are integers. If an increment is specified (`[1,11,2]`), the values can be integer or real numbers.
- The **while** loop executes its body while a certain condition is satisfied.
- The **while (True)** loop executes its body forever.
- The execution of a loop can be modified from within its body by statements **next** and **exit**. Statement **next** forces the execution of the next iteration and **exit** stops the execution of the loop.
- Any loop can be rewritten with **while (True)** and **exit** statements.
- **for** blocks, **while** blocks and *True-False* **if** branches cannot overlap.
- If loop statement fragments are defined by *variables*, some parts must nevertheless explicitly appear in the source form of the statements.

# Chapter 10

# SQLfast expressions

**Objective**: This chapter studies SQLfast conditional and computation expressions. It describes the SQLfast built-in functions and operators in five categories: *numeric*, *character string*, *logical*, *temporal*, *file/url*, *web* and *general* expressions. It also mentions the way to extend functions and operators according to specific applications through UDF and SQLfast libraries.

**Keywords**: SQLfast expression, numeric functions, character string functions, logical functions, temporal function, file/url functions, binary value, hexadecimal encoding, base64 encoding, UDF, SQLfast external libraries

**Statements and functions**: compute, numeric functions, character string functions, logical functions, temporal functions, file/web functions, general functions

**System variables and parameters**: -

## 10.1 Introduction

Several SQLfast statements include *expressions* that evaluate as numbers, character strings, date-time points or logical values (*true*, *false*). The main examples are the **`compute`**, **`if`** and **`while`** statements.

- the **`compute`** statement stores the value of an *expression* in a target variable:

```
compute X = ($Y$ - 1)/($Y$ + 1) + 1;
compute ID = upper(trim('§adr§') || ' (main)';
```

- the **if** statement executes the *true* branch or the *false* branch according to the logical value of the conditional *expression*,

```
if ('§adr§' like '%Regent%');
```

- the **while** statement executes its body while its conditional *expression* evaluates to *true*.

```
while ($X$ <= 10);
```

## 10.2 Value of a variable

Basically, the value of a variable is a character string. If this value represents a number, then it can be used in numeric expressions. However, it can be treated as a character string as well. For instance, if the value of variable **X** is **-124**, it can be divided by **2** (giving **-62**) but we can also compute its length (**4**).

## 10.3 Binary values

In some specific cases, a variable may contain a so-called *binary value*, also called *non-printable*. Such values are made up of strings of arbitrary bytes that are not intended to represent texts nor to be *writable* as such. So is the case of images, movies, mp3 records, pdf documents or character strings encoded in unusual coding schemes.

Variables that contain binary values require special care. For instance, they cannot appear between the delimiters of a substitution operator, since the latter would transform the statement into an illegal character string.

There are several secure ways to manipulate binary values in SQLfast. They will be described in detail in Chapter 15.

### A. Transfer from/to files

Binary values are transferred from and to *binary files*, notably through statements **variableToFile** (store the value of a variable in a file) and **BfileToVariable** (load the contents of a file in a binary variable)

### B. *hexadecimal* encoding

An arbitrary binary value (from a file or a variable) can be made printable through ***hexadecimal*** encoding. Each source byte is converted into two hexadecimal digits.

Since each of these digits is represented by a byte, the encoded string is twice as long as its source string.

Through this transformation, the values can be stored and transmitted without causing any problem. Of course, processing and exploiting an encoded value require that they be *decoded* (the inverse of *encoding*) so that they recover their initial form. *Hexadecimal* coding is done by SQLfast statements **`HEXencode`** and **`HEXdecode`** and their equivalent functions.

### C. *base64* encoding

***base64*** encoding follows the same principles as *hexadecimal* encoding, but the conversion is based on a *digit set* of 64 symbols instead of that of 16 symbols for hexadecimal. This coding is more efficient than *hexadecimal*, since the encoded string is about 33% larger as the source one.

*base64* coding is performed by SQLfast statements and functions **`b64encode`** and **`b64decode`**.

## 10.4 Syntax of SQLfast expressions

An SQLfast expression is a formula that includes *variable names* and/or *constants*. In most cases, the variable names must be delimited to indicate that they must be replaced by their value before evaluation (otherwise, this name will just be taken as a constant).

When SQLfast processes a statement it first prepares it by applying the *variable-value* substitution of all known (non binary) variables.

When all variable names of an expression that appears in the statement have been replaced by the variable values, the resulting formula *includes constants only*. Let us suppose that the values of variables **`adr`** and **`Y`** are, respectively, **Regent Street** and **`8`**. Then, after substitution, the statements mentioned above are transformed as follows:

```
compute X = (8 - 1)/(8 + 1) + 1;
compute ID = upper(trim('Regent Street') || ' (main)';
if ('Regent Street' like '%Regent%');
while (8 <= 10);
```

The syntax rule for conditional and computing expressions is quite simple: after substitution, **they must comply with the SQL syntax**.

So, an SQLfast condition is valid if, once preprocessed, it has been transformed into a valid SQL condition. An SQLfast computing expression is valid if, after substitution, it has been transformed into a valid SQL numeric, character string, logical or temporal expression. The value of an SQLfast expression is **the value of its SQL translation** as computed by the SQL engine.

The following sections describe some of the functions and operators available in SQLfast. By design, they all are SQL functions and operators. Some are standard (i.e., specified in the SQL standards), others are specific to the underlying DBMS (in this version, SQLite 3[1]) while all the others are added as UDF functions. This latter category is the most interesting, since it is extendible, as shown in Chapter 22.

## 10.5 Numeric functions

The value of an arithmetic expression is a number. It can make use of the following operators and functions[2]:

`+ - * /`: the four basic binary arithmetic operators. If both terms of an arithmetic operator are integers, the result is an integer. If at least one of them is a real number, the result is a real number. Division by zero returns an empty result.

`-`: unary operator that changes the sign of its argument.

`%`: modulo. Returns the remainder of the division of the left-hand argument by the right-hand argument. The result is a whole number that is an integer or real according to the type of the arguments.

`abs(n)`: Returns the absolute value of the numeric argument `n`.

`round(n)`: Returns the value of expression `n` rounded to the closest whole number. The result is a real number. 4.4 becomes 4.0 and 4.5 becomes 5.0.

`round(n,p)`:

Returns the value of expression `n` rounded to `p` decimals.

`truncate(n)`:

integer which is the integral part of `n`

`int(n)`: the nearest integer of `n`

`random()`: returns a pseudo-random integer between -9223372036854775808 and +9223372036854775807. Can be used to generate random integers between `0` and `M-1`: `abs(random()) % $M$`.

`random_i(n1,n2)`:

pseudo random integer number in range [`n1,n2`]

1. The reader is invited to consult the official description of SQLite core functions at http://www.sqlite.org/lang_corefunc.html (*SQL As Understood By SQLite*). When appropriate, this description has been adopted below.
2. The complete set of functions is described in document SQLfast-Commands.pdf, that can be downloaded from the SQLfast website.

`random_f(n1,n2)`:

pseudo random floating-point number in range [`n1,n2`]

`power(n,e)`:

floating point value of `n` power `p`

`sqrt(n)`: floating point square root of number `n`

`sign2(n)`: sign of `n` (two values: `-1`, `+1`)

`sign3(n)`: sign of `n` (three values: `-1`, `0`, `+1`)

`isNumeric(s)`:

whether (`0`, `1`) string `s` represents a decimal number; `s` can also be a number

`isNotNumeric(s)`:

opposite of `isNumeric(s)`

`isInteger(s)`:

whever (`0`, `1`) string `s` represents a integer value (note: **12.0** is considered non integer); `s` can also be a number

`isNotInteger(s)`:

opposite of `isInteger(s)`

See also functions and operators that return numbers, such as `length()`.

**Examples**

```
compute B = 3.0*($X2$ -1)/($X2$ + 1)
compute E = $X2$ % 5
compute R = abs(random()) % 1000;
```

*See also* Script [Expressions-Numeric.sql]

## 10.6 Character string functions

Main character string related operators and functions:

`||`: Returns a string obtained by concatenation of the left hand and right hand expressions.

`substr(s,i,n)`:

Returns a substring of input string `s` that begins with the `i`-th character and which is `n` character long. The left-most character of `s` has index 1. If `i` is negative then the first character of the substring is found by counting from the right rather than the left. If `n` is negative then the abs(`n`) characters preceding the `i`-th character are returned.

`substr(s,i)`:

Returns a substring of input string `s` that begins with the `i`-th character through the end of the string. If `i` is negative then the first character of the substring is found by counting from the right rather than the left.

`substr2(s,i1,i2)`:

the part of string `s` from position `i1` to position `i2`.

`upper(v)`:

Returns string `v` where all ASCII characters have been replaced by their upper-case version, if any.

`instr(v,s)`:

Finds the first occurrence of string `s` within string `v` and returns the index of its first character, or 0 if there is no instance of s.

`countInst(s,a)`:

number of non overlapping instances of `a` in `s`; `s` is a string or a number

`length(v)`:

If `v` is a character string, returns the number of characters in `v`. If `v` is a numeric value, returns the length of a string representation of `v`.

`like(p,s)`:

Equivalent to SQL predicate `s like p`.

`like(p,s,e)`

Equivalent to SQL predicate `s like p escape e`.

`lower(s)`: Returns string `s` where all ASCII characters have been replaced by their lower-case version, if any.

`ltrim(s)`: Returns string `s` where the leading spaces (including tabs and end-of-line) have been removed.

`ltrim(s,c)`:

Returns string `s` where the leading characters `c` have been removed.

`rtrim(s)`: Returns string `s` where the trailing spaces have been removed.

`rtrim(s,c)`:

Returns string `s` where the trailing characters `c` have been removed.

`trim(s)`: Returns string `s` where the leading and trailing spaces have been removed.

`trim(s,c)`:

Returns string `s` where the leading and trailing characters `c` have been removed.

`deaccent(s)`:

string `s` where all accents have been removed

`simplify(s)`:

same as `lower(deaccent(s))`

`shrink(s)`:

same as `simplify(s)` + remove symbols such as space, `-`, `_`, `'`, etc.

`reduce(s,c)`:

string `s` in which all the sequences of several instances of each character in `c` have been reduced to one instance.

`reduceAll(s)`:

string `s` in which all the sequences of several instances of the same character have been reduced to one instance.

`replace(v,s,r)`:

Returns a string formed by substituting string `r` (*replacement*) for every occurrence of string `s` (*search*) in string `v`. Returns `v` if `s` is empty.

`hex(s)`: hexadecimal utf-8 code point string of the characters of `s`

`doubleQuote(s)`:

string `s` in which all internal single quotes have been doubled; better to express `s` into double-quotes!

`repeat(s,n)`:

string built as `n` times substring `s`

`startWith(s,h)`:

whether string `s` starts with string `h`; all strings start with "

`endsWith(s,n)`:

whether string `s` ends with string `e`; all strings end with "

`frame(s,n,a,p)`:

string `s` framed in a `n`-long string; `a` is the alignment ('<' = left, '>' = right, '^' = center); unused positions are padded with characters `p`; `s` is truncated (right, left, both) if its length is larger than `n`.

`thousandSep(s,sep)`:

the integer part of numeric string `s` is formated by inserting symbol `sep` between 3-digit blocks. `s` is a string or a number.

`b64encode(s,ext)`:

string `s` encoded in *base64*; parameter `ext` denotes either the standard encoding (`'std'`) or the url-safe encoding(`'url'`); this function is a reduced version of statement **`b64encode`**α**`To`**β in which `s` is a character string.

`b64decode(s)`:

*base64* string `s` decoded into its original text; this function is a reduced version of statement **`b64decode`**α**`To`**β in which the result is a character string.

`char2hex(s)`:

string `s` encoded in *hexadecimal*; this function is a reduced version of statement **`HEXencode`**α**`To`**β in which `s` is a character string.

`hex2char(s)`:

*hexadecimal* string `s` decoded into its original text; this function is a reduced version of statement **`HEXdecode`**α**`To`**β in which the result is a character string.

`utf8Compliant(s)`:

whether string `s` may be an **utf8** byte string.

`latinCompliant(s)`:

whether string `s` may be a **Latin** (e.g., cp1252) byte string.

`hexCompliant(s)`:

whether string `s` may be an **hexadecimal** character string.

`base64Compliant(s)`:

whether string `s` may be an **base64** character string.

`hash(s)`:

returns a string of 64 hexadecimal characters resulting from the secure hashing function `sha256`.

`encrypt(s,k)`:

encrypts source data **s** with key **k** then codes it into base64. Null values and empty strings are encrypted as empty strings.

`decrypt(c,k)`:

decrypts *base64* coded encrypted value **c** with key **k**.

### Examples

```
compute K = upper('$Y$')||'S and '||upper('$Z$')||'S'
compute I = instr('$X$','Regent')
compute J = replace('$X$','$Y$','Queen')
```

*See also* Script [Expressions-Character.sql]

## 10.7 Logical functions

The value of a logical expression is `True` or `False`. These values also have a numeric equivalent: integer `0` is interpreted as `False` and integer `1` (more generally any non-zero number) as `True`.

`and, or, not`: standard logical connectors.

`<,<=,=,=>,>,<>`:
: standard comparison operators.

`(not) in`:
: Standard SQL predicate. `True` if the left hand expression is (is not) in the list specified by the right hand side.

`(not) between`:
: Standard SQL predicate. `True` if the left hand expression lies (does not lie) in the interval specified at the right hand side.

`(not) like`:
: Standard SQL predicate. `True` if the left hand expression meets (does not meet) the pattern specified at the right hand side.

`True`
: Logical value *True*. May appear as the condition of a **`while`** statement only (**`while`** `(True)`).

`&`: bit string **`and`** operator

`|`: bit string **`or`** operator

`~`: bit string **`not`** operator

`<<`: bit string left shift operator

`>>`: bit string right shift operator

### Examples

```
compute K = ($I$ <= 100) or ($J$ <= $I$)
if ('$A$' in ('azerty','qwerty','qwertzu'))
if ('keyboard $A$' like '%ertzu%')
if ($I$ between $X$ and $Y$)
if (($I$*$J$ = $Y$) and ($I$ < $J$))
```

*See also* Script [Expressions-Logical.sql]

## 10.8 Temporal functions

These functions manipulate *date*, *time point* and *datetime* values. SQLfast follows the ISO format for date, time and datetime:

- *date*: `YYY-MM-DD`
- *time*: `HH:MM:SS` and `HH:MM:SS.mmm`
- *datetime*: `YYY-MM-DD HH:MM:SS` and `YYY-MM-DD HH:MM:SS.mmm`

SQLfast provides[3] three standard literals[4] that denote the current time at execution time. Their value is a character string in a definite format:

- `current_time`: the current time, in format `hh:mm:ss`.
- `current_date`: the current date in format `yyyy-mm-dd`.
- `current_timestamp`: current date and time in format `yyyy-mm-dd hh:mm:ss`. Can also be obtained through string `'now'` when used as argument `t` in a time function.

It also provides five functions that return time related values. They do not comply with the standard syntax of SQL time functions but they can replace them:

- `time(t,m*)`: returns the character string that represents the date denoted by `t` in format `hh:mm:ss`. `time()` is equivalent to `time('now')`.
- `date(t,m*)`: returns the character string that represents the date denoted by `t` in format `yyyy-mm-dd`. `date()` is equivalent to `date('now')`.
- `datetime(t,m*)`: returns the character string that represents the date and the time denoted by `t` in format `yyyy-mm-dd hh:mm:ss`. `datetime()` is equivalent to `datetime('now')`.
- `strftime(f,t,m*)`: returns the character string that represents excerpts of date and/or the time denoted by `t` in format `f`.
- `julianday(t,m*)`: returns the (real) number of days since year 4714 B.C. Converts *date*, *time* and *datetime* values into a julian day, allowing them to be added and substrated. Synonym: `jd(t)`.

Argument `t` is a character string that denotes a date, a time or a timestamp (date *plus* time).

Symbol `m*` indicates an optional list of modifiers. Each modifier alter the original value of `t` by adding/subtracting interval (start of year, + 5 months, 4th day of week, etc.). The modifiers are the following:

`n days`: add/subtract `n` days
`n hours`: add/subtract `n` hours
`n minutes`: add/subtract `n` minutes

3. From SQLite function library (http://sqlite.org/lang_datefunc.html)!
4. A *literal* in a language is a word that can be used in place of a special value.

`n.m seconds`: add/subtract `n` seconds
`n months`: add/subtract `n` month
`n years`: add/subtract `n` years
`start of month`: replace the day by the 1st of the month
`start of year`: replace the day by the 1st of the year
`start of day`: replace the hour by the 1st of the day
`weekday n`: change the day to the next day of week number `n`
`unixepoch`: conversion of a Julian day number into Unix time
`localtime`: adjust the time string `t` in UTC[5] to local time
`utc`: adjust the time string `t` in local time to UTC

Format `f` of function `strftime` is a character string in which time extractors can be inserted. These extractors are the following:

`%d` day of month: `01-31`
`%f` fractional seconds: `ss.sss`
`%H` hour: `00-24`
`%j` day of year: `001-366`
`%J` Julian day number
`%m` month: `01-12`
`%M` minute: `00-59`
`%s` seconds since 1970-01-01
`%S` seconds: `00-59`
`%w` day of week: `0-6` (sunday = `0`)
`%W` week of year: `00-53`
`%y` year: `00-99`
`%Y` year: `0000-9999`

Other functions:

- `addToDate(d,n)`: adds `n` days to date `d`.
- `nextDate(d)`: adds `1` day to date `d`.
- `prevDate(d)`: subtracts `1` day from date `d`.
- `current_timestamp_full()`: string value of current date and time down to milliseconds in format `YYYY-MM-DD hh:mm:ss.mmm`
- `current_date_string()`: string value of current date in format `YYYY-MM-DD`
- `current_time_string()`: string value of current time in format `hh:mm:ss.mmm`
- `time_string(t)`: string value of time `t` (in seconds) in format `hh:mm:ss.mmm`
- `year(d)`: if `d` is the string expression of a date or datetime, returns the year part as an integer

5. *Universal Coordinated Time*, the modern successor of the Greenwich Mean Time (GMT).

- `month(d)`: if `d` is the string expression of a date or datetime, returns the month part as an integer
- `day_in_month(d)`: if `d` is the string expression of a date or datetime, returns the day part as an integer
- `hour(t)`: if `d` is the string expression of a time or datetime, returns the hour part as an integer
- `minute(t)`: if `d` is the string expression of a time or datetime, returns the minute part as an integer
- `second(t)`: if `d` is the string expression of a time or datetime, returns the second part as an integer
- `millisecond(t)`: if `d` is the string expression of a time or datetime, returns the millisecond part as an integer
- `clock(t)`: if `d` is the string expression of a date, time or datetime, returns the `hh:mm:ss` part; allows missing parts, replaced by `00`
- `isDate(s)`: whever (`0`, `1`) string `s` represents a date or a datetime value in one of the six standard patterns (YYYY-MM-DD, YY-MM-DD, MM-DD-YYYY, MM-DD-YY, DD-MM-YYYY, DD-MM-YY); other separators are allowed: '`/`' and '`.`'.
- `isNotDate(s)`: opposite of `isDate(s)`
- `datePattern(d)`: one of the six standard patterns with which string `d` matches. In case of mismatch, indicates its kind (string: not a string; length: invalid length; pattern: no pattern found).
- `normDate(d,span)`: the normalized form of date `d` (**YYYY-MM-DD**). `d` must be in one of the standard pattern. Returns `''` in case of mismatch. Parameter span is used for 2-digit years (`YY`). It indicates whether the year field is to be interpreted as a *historical* value (`56` means `56 DC`, or `0056`) or as a *present* value (`56` means `1956`). Its values are `'H'` or `'P'`, case insentitive.

### Examples

```
compute Y = date(current_timestamp)
```

```
compute Y = date('now','start of year','5 months','weekday 4'):
```
first Thursday of May for the current year

```
compute Y = round(julianday('2010-01-01') - julianday('now')):
```
how many days from January 1st, 2010?

```
compute Y = strftime('%d/%m/%Y',$Y$):
```
date in format `dd/mm/yyy`.

*See also* Script [Expressions-Time.sql]

## 10.9 File/web functions

These functions are related to file, directory, path and url manipulation and conversion.

- `fileExists(p)`: whether string `p` is the full name of an existing file
- `dirExists(p)`: whether string `p` is the full name of an existing directory
- `urlExists(p)`: whether a website with url `p` exists
- `normalizePath(p)`: `p` where '`\`' replaced with '/'; spaces removed
- `lastPathNode(p)`: last node of path `p` (leading/trailing '/' excluded)
- `pathPrefix(p)`: path `p` without its last node (trailing '/' removed)
- `nodeExtension(p)`: the extension of the last node of path `p`, if any
- `pureNodeName(p)`: the last node of path `p`, minus the extension, if any
- `sendMail(Host,Login,Pw,From,To,Cc,Form,Subj,Body,Attach)`: sends a mail. `Host` is the address of the SMTP server and (`Login`, `Pw`) are logging parameters. `From`, `To` and `Cc` are mail addresses of the recipient(s), of the sender(s) and of cc(s). `Form` is the format of the message ('`plain`' or '`html`'), `Subj` is the subject, `Body` is the body and `Attach` is the list of the addresses of the attached files.

## 10.10 General functions

These functions are applicable to arguments of any type.

`max(v1,v2, ... )`

Returns the element of the list of arguments with the highest value. This list must include at least 2 elements.

`min(v1,v2, ... )`

Returns the element of the list of arguments with the lowest value. This list must include at least 2 elements.

`case when c1 then v1 when c2 then v2 ... else vn end`:

Returns the value in {`v1,v2,...`} corresponding to the first condition in {`c1,c2,...`} that evaluates to `true`. If there is none, returns (optional) value `vn`.

`cast(v as t)`:

Returns value `v` converted into type `t`

`itemLen(s,sep)`:

Considering string `s` as an array of elements separated by string `sep`, returns the number of elements in `s`

`item(s,i,sep)`:

Considering string `s` as an array of elements separated by string `sep`, returns the `i`th elements of `s`

`itemIndex(s,e,sep)`:

Considering string `s` as an array of elements separated by string `sep`, returns the position of first element `e`. Returns `0` if `e` is absent.

`itemLenS(s)`:

Same as `itemLen(s,sep)` with the default separator

`itemS(s,i)`:

Same as `item(s,i,sep)` with the default separator

`itemIndexS(s,e)`:

same as `itemIndex(s,i,sep)` with the default separator

`group_concat2(v,d,o,d,sep)`:

replacement for `group_concat` aggregate function. Values `v` are concatenated, separated by `sep`. Parameter `d` (0, 1) specified the `distinct` modifier. Parameter `o` defines the order key and `d` specifies the order direction (1 = ascending, 2 = descending). Only available in SQL queries.

**Examples**

```
compute Y = max($X1$,$X2$,$X3$,10)
compute Y = abs(min($X1$,$X2$))/2
compute Y = case when $X$ = 0 then 'one'
                 when $X$ = 1 then 'two'
                 when $X$ = 2 then 'three'
                 else 'other' end
```

*See also* Script [Expressions-General.sql]

## 10.11 User defined functions (UDF)

Users of SQLfast can add their own functions to the standard built-in function set described above. They are called *user defined functions* or UDF. They are coded in Python and are collected in optional system module **SQLfastUDFlib.py**. This module comprises the definition of user functions as well as function **`createFunctions`**. The body of this function is a sequence of statements creating the UDF. The detail of these statements is given in Chapter 22.

Script 10.1 shows the contents of module SQLfast_UDF_lib.py that defines new functions `addition` and `multiplication`. Script 10.2 is an illustration of the use of these functions.

```
import sqlite3
def SQLadd(X,Y):
    return X + Y
def SQLmultiply(X,Y):
    return X * Y
def createFunctions(connect):
    connect.create_function("addition",2,SQLadd)
    connect.create_function("multiplication",2,SQLmultiply)
```

**Script 10.1 -** Defining custom functions **addition** and **multiplication**

```
extract P,Q = select Price,QonHand
              from PRODUCT where ProdID = 'PA60';
compute val = addition(100,multiplication($P$,$Q$))
```

**Script 10.2 -** Using UDF in an SQLfast expression

The set of SQL functions can also be extended with a similar mechanism. These functions must be declared in Python module **SQLiteUDFlib.py**.

## 10.12 Functions from external libraries

SQLfast libraries provide additional functions, either in standard data types or in specialized topics. In this version of SQLfast, libraries are Python modules. The standard distribution includes a first bunch of themes such as character string (**LStr**), text (**LText**), file (**LFile**) and web related operations (**LWeb**). A function is called by statement **function** (or, equivalently, **execApp**). Two main differences with UDF: they may not appear in SQLfast expressions but they can return several results.

Some of these libraries and functions are described in Chapter 22. They can be extended quite easily and new libraries can be developed to meet the requirements of specific applications.

### Example

The following example uses functions from string manipulation library **LStr**. It experiments with SQL string format. Variable `source` contains a string as it must appear on the screen, `arg1` is its equivalent as an SQL constant and `arg2` contains its recovered format.

```
set source = Owner's file;
function arg1 = LStr:SQLunquote {$source$};
function arg2 = LStr:SQLquote {$arg1$};
write $source$ - $arg1$ - $arg2$;
```

## 10.13 Summary

This chapter describes the main SQLfast expressions.

- The main SQLfast expression families are *numeric*, *character string*, *logical*, *temporal*, *File/url* and *general*.
- SQLfast expressions obey the syntactic rules of the underlying SQL DBMS.
- Additional functions can be developed to extend SQfast through external Python libraries or through UDF.

# Chapter 11

# Output channels, files and directories

**Objective**: This chapter discusses the way a script can output the texts it produces, for instance through statements `write` and `select`. It describes various media, the so-called *output channels*, on which it can write output data. It also examines SQLfast resources to manage files and directories.

**Keywords**: output channel, output window, user output file, output variable, file name, default folder

**Statements and functions**: outputOpen, outputAppend, write, print, select, selectFile, selectDirectory, dirNames, fileNames, dirFileNames, dirNamesAll, fileNamesAll, dirFileNamesAll, renameFile, copyFile, deleteFile, LFile:pathSplit, LFile:infoFile

**System variables and parameters**: outputDirectory, outputtype, outputmode, dbDirectory, scriptDirectory, outputDirectory, fileDirectory, helpDirectory

## 11.1 The SQLfast output window: the default *output channel*

Let us consider the following elementary script.

```
openDB ORDERS.db;
select CustID, Name from CUSTOMER where City = 'Poitiers';
```

The result of the execution of the **`select`** statement is sent to the **SQLfast output window** (Figure 11.1).

The medium on which the script writes the data it produces is called the *output channel*. In the previous chapters, the SQLfast output window was the *default output channel*, that is, the channel automatically used when no other channel has been specified.

Actually, in addition to this one, SQLfast provides three alternative output channels:

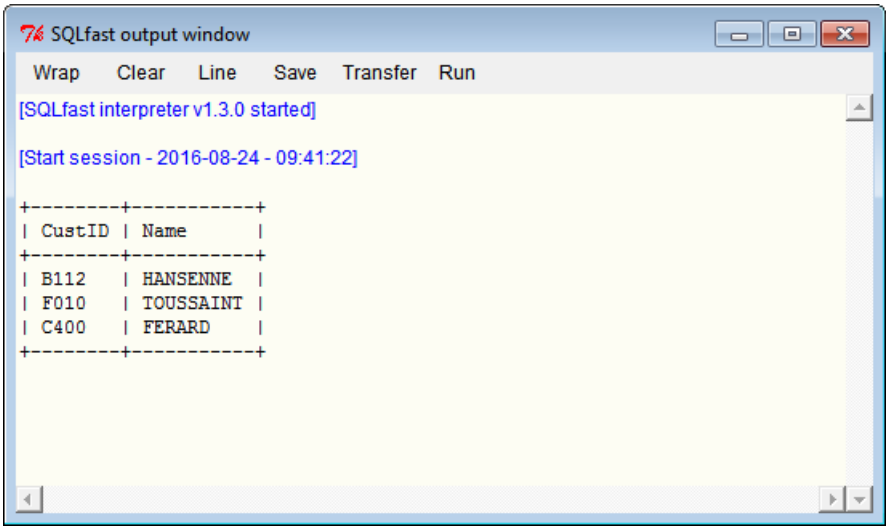



**Figure 11.1 -** The SQLfast output window

– the SQLfast output window (*Output window* for short)
– user output files
– the standard output file
– script variables.

Only one channel can be opened at a time. Opening a channel replaces the current one, which is closed. Whether its content is preserved or not depends on the **mode** with which it will be reopened later on:

- **`write` mode**: the channel is opened but its former content is lost; this mode is selected through statement **`outputOpen`**.
- **`append` mode**: the channel is opened, with its former content preserved. The next text is added to this content; this mode is selected through statement **`outputAppend`**.

## 11.2 The (SQLfast) output window

In the standard configuration, the default channel is the output window. When another output channel is current, going back to the output window is obtained by one of two statements

```
outputOpen window;
outputAppend window;
```

The first one starts with a blank window while the second one preserves the material already written in the window. The output window has six action buttons:

- `Wrap`: changes the way long lines are displayed in the text area. Either such lines are broken and wrapped onto next lines, or they are left unchanged, so that the hidden part can only be seen by expanding the window horizontally or by using the slide bar. This button changes the current mode for the other.
- `Clear`: clears the text area.
- `Line`: adds a blank line at the end of the text area.
- `Save`: copies the content of the output area in a text file.
- ~~`Transfer`: copies the lines selected in the output area in the script area. The former content of the script area is lost.~~ (function removed)
- `Run`: executes the lines selected in the output area. The content of the script area is left unchanged.

## 11.3 User output files

A *user output file* is a text file in which output data are written.

The **`outputOpen`** statement closes the current output channel then creates a new file with the specified name (Script 11.1). If the file already exists, it is first deleted. The output data can be examined and further exploited by any text processor (Figure 11.2).

```
outputOpen display.csv;
```

```
openDB ORDERS.db;
  outputOpen resultset.txt;
    select CustID,Name,City from CUSTOMER
    where  City in ('Poitiers','Toulouse');
  outputAppend window;
```

**Script 11.1 -** Writing data in file result.txt, then going back to the output window [Writing-in-a-file.sql]

The **outputAppend** statement closes the current output channel then opens the file with the specified name. The previous content of the file is preserved. If the file does not exist yet, it is created.

```
outputAppend display.csv;
```

## Note on file names

The name of the output file can be complete, that is, it specifies an absolute address (e.g., **D:/Results/display.csv**). It can also be relative (e.g., **Results/display.csv**). In this case, the file is stored in the *default output folder*, which is, in the standard installation, **SQLfast/Output**. In the latter case, the file will be stored in folder **D:/SQLfast/Output/Results/display.csv**.
However, this default folder can be changed through parameter `outputDirectory` in parameter file **SQLfast.ini**:

```
outputDirectory = D:/Results
```

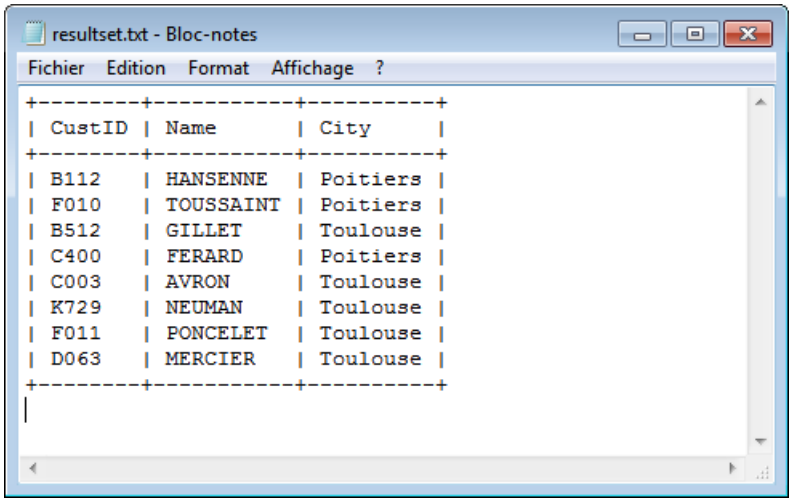



**Figure 11.2 -** The user output file created by script 11.1 in folder D:/SQLfast/Output

## 11.4 Output variables

When the text sent to one of the output channels need not be kept but must just be reused by the current script, it can be stored in a **variable** instead of in a file. The following statements close the current output channel and redirect further output data to script variable **result**:

```
outputOpen result.var;
outputAppend result.var;
```

Suffix **.var** tells the SQLfast engine that **result** is the destination variable of **write** and **select** statements.[1]

As for the other channels, `outputOpen` empties the variable before use while `outputAppend` preserves its former content.

The size of an SQLfast variable is only limited by available RAM. In a standard machine, a variable could contain very large values. Therefore, writing in a variable is not limited to small amounts of data, as would be the case in most programming languages.

## 11.5 Closing an output channel

There is no specific statement to explicitly close an output channel. For file channels (i.e., standard and user files) the corresponding file is closed when the main script terminates or when another channel is selected.

For instance, when generating a text in a file, it is good practice to switch the channel to the output window at the end of the process, as shown in Script 11.1.

## 11.6 Setting the default output channel and output mode

When no output channel has been specified, SQLfast selects the *default channel*. The built-in default channel is the *output window*. However, another default can be chosen in initialization file SQLfast.ini, in which the directive **outputtype** can be set to `window` or `standard`:

```
outputtype = standard
```

The initialization file also allows to set the default opening mode of the output channel, `write` or `append`:

```
outputmode = write
```

1. Needless to insist on avoiding to give a file the suffix **.var**!

The built-in default mode is `write`.

## 11.7 Clearing the output window

There is no such SQLfast statement as `clearWindow`, that would clear the output window. For a good reason: `outputOpen` will do the job:

```
outputOpen window
```

## 11.8 write or print?

SQLfast statements **write** and **print** have the same behavior, except for the medium on which they write their character string: **write** writes on the current output channel while **print** writes in the output window, whatever the current output channel.

```
outputOpen result.sql;
  for cus,nam,cit = [select CustID,Name,City
                     from CUSTOMER where Cat is not null];
     write insert into EMP(CustID,Name,City)
           values ('§cus§','§nam§','§cit§');;
     print -- SQL insert for '$cus$' generated;
  endfor;
outputAppend window;
```

**Script 11.2 -** Mixing *write* and *print* statements in the same script

## 11.9 Example: playing with output channels

Script 11.3 shows how these channels can be used. The data on the customers from *Poitiers* and *London* are first stored in a tabular layout in variable **resultset**. Then, the content of this variable is added (appended) to the current content of external user file **extfile.txt**. Finally, the data generated are written in the output window.

```
openDB ORDERS.db;
outputOpen resultset.var;
  write-b Customers from Poitiers;
  select CustID, Name from CUSTOMER
  where City = 'Poitiers';
  write-b Customers from London;
  select CustID, Name from CUSTOMER where City = 'London';
outputAppend extfile.txt;
  write $resultset$;
outputAppend window;
  write $resultset$;
```

**Script 11.3 -** Writing data on different output channels [Output-channels.sql]

## 11.10 Managing files and directories

Specifying output channels lead us to manipulate files and directories. It is a good opportunity to examine in more detail the resources SQLfast provides us with in file and directory management.

We will first precise how we can (and must) use file and directory names, specially if we want to write *portable* SQLfast scripts.

### 11.10.1 About file and directory names

Let us first recall the rules concerning file and directory names in SQLfast scripts. When the full (or absolute) name of a file or a directory is specified in a script, SQLfast does not interpret it and leaves it as it is:

```
deleteFile D:/SQLfast/Files/result.txt;
```

This *full name* comprises all the nodes of the file or directory path, from the *device name* (**D:**) to the *local name* (**result.txt**). The main drawback of full names is that the scripts in which they appear are not *portable*, in that, they cannot easily be moved to another directory, not to mention to another machine. If we move the SQLfast directory from drive **D:** to drive **E:**, the full name mentioned above is likely to become invalid.

The best practice to write portable scripts is to use *relative names* of files and directories. These names consist of a right side fragment of a full name. For instance, fragments **SQLfast/Files/result.txt**, **Files/result.txt** and **result.txt** all are relative names of file **D:/SQLfast/Files/result.txt**. The right relative name to use depends on the *reference directory*, that is, the full name of the directory that is implicitly added to the left of the relative name of the file to form its full name. For

instance, relative name **Files/result.txt** correctly identifies the file of our example if its reference directory is **D:/SQLfast**.

To simplify the use of relative names, SQLfast provides a set of six registers (system variables) that specify the most useful reference directories. These registers receives defaults values at start time, but their values can be changed in initialization file SQLfast.ini or (with much caution!), dynamically by scripts themselves.

The most important one is **generalDirectory**, which denotes the SQLfast directory, in which all the components of SQLfast are located. It can be given the full name of a directory, which would be bad practice in most case, or, much better, symbolic name **_system**, which denotes the full name of the SQLfast directory, wherever it is located. Its value is automatically computed when the system starts. Except is very special contexts, the initialization file should include the following directive:

```
generalDirectory = _system
```

The other five registers specify respectively

- **dbDirectory**: the default database directory, where database are and will be located when no directory is mentioned
- **scriptDirectory**: the default script directory, where scripts are located when no directory is mentioned
- **outputDirectory**: the default output directory, where the files created by `select` and `write` statements will be located
- **fileDirectory**: the default file directory; suggested directory to store all kinds of files; this register is not used by SQLfast
- **helpDirectory**: the default help and tutorial directory, where help documents (*.help) and tutorials (*.tuto) are located.

Just like generalDirectory, they can be assigned full directory names in the initialization file. However, in most case, it is recommended to give them names relative to the general directory, for example:

```
dbDirectory     = Databases
scriptDirectory = Scripts
outputDirectory = Output
fileDirectory   = Files
helpDirectory   = SQLfastHelp_EN
```

When used in a script, these system variables contain the full names of their respective directories. The following script:

```
write $generalDirectory$;
write $dbDirectory$;
write $scriptDirectory$;
write $outputDirectory$;
write $fileDirectory$;
```

```
write $helpDirectory$;
```

will produce, following our installation example:

```
D:/SQLfast
D:/SQLfast/Databases
D:/SQLfast/Scripts
D:/SQLfast/Output
D:/SQLfast/Files
D:/SQLfast/SQLfastHelp_EN
```

With these settings, the following statements are equivalent:

```
openDB ORDERS.db;
openDB D:/SQLfast/Databases/ORDERS.db;
openDB $dbDirectory$/ORDERS.db;
openDB $generalDirectory$/Databases/ORDERS.db;
```

These system variables can be modified by scripts, but for full names only.

### 11.10.2 Selecting a file or a directory

Statement **selectFile** opens a file selection dialog box. Three parameters must be specified: the full or relative name of the starting directory, a name describing the file type and the file extension (Script 11.4). If button **OK** or, more generally the *positive* button, has been clicked on, the statement returns the full name of the selected file, otherwise, it returns an empty string.

```
selectFile path = Files,Text files,txt;
if ('§path§' <> '') write File §path§ selected;
```

**Script 11.4 -** Selecting a file

Statement **selectDirectory** is used to select a directory. The full or relative name of the starting directory must be specified (Script 11.5). If button **OK** or, more generally the *positive* button, has been clicked on, the statement returns the full name of the selected directory, otherwise, it returns an empty string.

```
selectDirectory path = Scripts;
if ('§path§' <> '') write Directory §path§ selected;
```

**Script 11.5 -** Selecting a directory

### 11.10.3 Examining the content of a directory

SQLfast provides a family of statements to examine the content of a directory. They write in a text file the names of the files and/or directories contained in this directory.

- **dirNames** `Files/names.txt = Scripts;`
  writes in text file Files/names.txt the full names of the direct **subdirectories** of directory Scripts. The names of the file and of the directory may be full names or relative names (as this example). In this latter case, the reference directory is the general directory. These naming rules are also those of all the other statements of this family.
- **fileNames** `Files/names.txt = Scripts,sql;`
  writes in text file Files/names.txt the full names of the **files** with extension sql in directory Scripts. If no extension is specified, all the files are examined.
- **dirFileNames** `Files/names.txt = Scripts,sql;`
  combination of **dirNames** and **fileNames**.
- **dirNamesAll** `Files/names.txt = Scripts;`
- **fileNamesAll** `Files/names.txt = Scripts,sql;`
- **dirFileNamesAll** `Files/names.txt = Scripts,sql;`
  extension of the previous statements, in which the subdirectories are explored recursively. See example in Script 11.6.

```
fileNamesAll Files/sql-names.txt = Scripts;
for name = [file Files/sql_names.txt] write $name$;
```

**Script 11.6 -** Displaying the full name of all the sql scripts in directory Script

### Notes

1. By adding `-s` (for ***s**hort*) and or `-r` (for ***r**oot*) suffixes to the statement keyword, we can modify the result:
   - suffix `-s` reduces file and directory references to their local names instead of their full names (example: **`fileNames`**`-s`)
   - suffix `-r` adds the source directory to the name list (for `dirNames` and `dirFileNames` categories only) (example: **`dirNamesAll`**`-r`)
   - these sufixes can be combined (example: **`dirNamesAll`**`-rs`).
2. Names may include characters that are not recognized by the current platform. They are replaced by character _ and the name itself is prefixed by character `*`.

### 11.10.4 Examining the properties of a file or a directory

The properties of a file and a directory are given by two functions, namely splitPath and infoFile of library LFile.

– Function **LFile:splitPath** returns the components of a path:[2] the full name, the local name, the directory (full name minus the local name) and the extension.

– Function **LFile:infoFile** returns the properties of a file: its size in bytes, its creation time (not reliable in Windows!), its last update time and its last access time.

Script 11.7 illustrates the use of these functions.

```
for name = [file Files/names.txt];
   function ful,pre,dir,fil,ext = LFile:splitPath {$name$};
   if ('§fil§' = '');
      write Directory name: '§ful§','§pre§','§dir§';
   else;
      function siz,cre,upd,acc = LFile:infoFile {$name$};
      write File name: '§ful§','§pre§','§fil§'
      write File properties: $siz$,'$upd$','$acc$'
   endif;
endfor;
```

**Script 11.7 -** Analyzing the files and directories references in names

### 11.10.5 Renaming, copying and deleting files

We can rename, copy and delete files through three specific statements:

– **renameFile** Databases/ORDERS.db.bak,ORDERS.db;

Renames source file Databases/ORDERS.db.bak to target name ORDERS.db. This file must be closed and must exist. However, if suffix **-e** is specified (renameFile-e), does nothing if the source file does not exist. The target name must be a local name. No file with the target file name may exist unless suffix **-r** is specified, (renameFile-r) in which case the target file is replaced if it already exists. These suffixes can be combined in **-er** or **-re**.

**- copyFile Databases/**ORDERS.db,Files/ORDERS.db.bak;

Creates a copy of source file ORDERS.db with name ORDERS.bak. The source file must exist and must be closed. However, if suffix **-e** is specified, does nothing if the source file does not exist. The target file must not exist.

2. This path can also be a url.

However, if suffix **-r** s specified, the target file is replaced if it already exists. These suffixes can be combined in **-er** or **-re**.

**- deleteFile** `Files/result.txt`

Deletes file `Files/result.txt`. This file must exist and must be closed. However, if suffix **-e** is specified (`deleteFile-e`), does nothing if the file does not exist.

## 11.11 Summary

The SQLfast environment provides three **output channels** on which data can be written, such as the result of statements **write** and **select**: the *SQLfast output window*, *user files* and *output script variable*.

- The *default channel* is opened when no other channel has been explicitly opened. The default channel is the *SQLfast output window*, unless another default has been defined in file SQLfast.ini through **outputtype** directive.
- Switching from the current output channel to the SQLfast output window is obtained through statements **outputOpen/outputAppend** `window`.
- Statement **outputOpen** changes the current output channel for another one. The previous content of the latter is lost. Statement **outputAppend** changes the current output channel for another one. The content of the latter is preserved.
- Output data can be written in a text *file* specified by the script. Switching from the current output channel to a user output file is obtained through statements **outputOpen/outputAppend** `display.csv`.
- The output channel can be a *variable*. It is specified by its name, suffixed by fictitious extension **.var**. Switching from the current output channel to variable `result` is obtained through statements **outputOpen/outputAppend** `result.var`.
- File and directory can be interactively selected through statements `selectFile` and `selectDirectory`.
- To examine and analyze the content of a directory, we can use statements `dirNames`, `fileNames`, `dirFileNames`, `dirNamesAll`, `fileNamesAll`, `dirFileNamesAll`. They write in a text file the names of the specified files and directories.
- The properties of a file are extracted through function `LFile:infoFile`.
- Files can be renamed (`renameFile`), copied (`copyFile`) and deleted (`deleteFile`).
- Default directories are specified by system variables dbDirectory for databases, scriptDirectory for scripts, outputDirectory for the results of `select` and `write`

statements, fileDirectory for various external files and helpDirectory for help documents and tutorials. These variables can be set in initialization file SQLfast.ini and dynamically modified by scripts. The system directory, generally the SQLfast directory, is specified by variable generalDirectory.

# Chapter 12

# Dialogues

**Objective**: This chapter collects and discusses various dialogue techniques that have been used in previous chapters and adds some new ones. More complex dialogues will be studied in the next chapters.

**Keywords**: Graphical User Interface, GUI, dialog box, data entry box, text entry box, single selection, multiple selection, label separator

**Statements and functions**: ask, askData, ask-u, askText, askText-u, selectOne, selectOne-u, selectMany, selectMany-u, selectList, selectList-u, selectList2, selectList2-u, set, update, @, |, parameter label-separator, item()

**System variables and parameters**: label-separator, DIALOGbutton, text wrap, stdTextX, stdTextY

## 12.1 The SQLfast GUI

Dialogues define the way information is exchanged between a script and its user. SQLfast provides a limited but powerful set of exchange modes, ranging from simple message boxes to complex data entry and item selection forms and to graphical display of still and animated pictures and drawings. However, SQLfast is not a full application programming language, such a Java, C++ or Python, so that more advanced and sophisticated GUI[1] cannot be built easily, except through SQLfast libraries, external applications (with **`execApp`** or **`function`** statements) or external programs (with **`execProg`** statement).

This chapter collects and discusses some of the dialogue techniques that have been used in previous chapters and adds some new ones: data entry boxes, text entry boxes and item selection boxes. More complex layouts and dialog boxes will be studied in the next chapters.

## 12.2 Standard data entry box

The basic data entry box comprises a message field and a vertical list of data entry fields. A label is associated to each entry field (Figure 12.1). When the box pops up, the user can fill the fields and click on button **OK**, so that the contents of the fields are stored in the target variables. The user also can cancel the data entry process (button **Cancel** or by closing the box), in which case no data are transmitted to the script and the target variables keep their initial values.

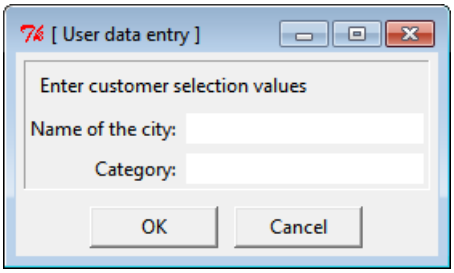



**Figure 12.1 -** Simple data entry box

This data entry box is created by statement **askData** (or **ask** for short). This statement comprises three kinds of information (Script 12.1):

1. the text to display in the message field ([`Enter customer selection values`])
2. the list of labels associated to entry fields (`Name of the city | Category`)
3. the list of names of the target variables that receive the data entered by the user (`city,cat`).

```
ask city,cat = [Enter customer selection values]
                Name of the city: | Category:;
write $city$,$cat$;
```

**Script 12.1 -** SQLfast statement to generate a simple data entry box [Chapter-12-01.sql]

1. **G**raphical **U**ser **I**nterface. It means just what it tells.

The message is enclosed between symbols **[** and **]**. It is optional. If missing (no **[]** bloc or empty message), no message field appears in the dialog box. The text may include end of line (**@n**), tab (**@t**) and space (**@s** and **@S**) control characters:

- **@n**: go to next line (or new line)
- **@t**: go to next tab stop
- **@s**: one space character.
- **@S**5: five space characters (**@S**1 = **@s**).

The list of labels and the list of target variable names must have the same (non null) length. Labels may include control characters as well, though they should be used with some caution!

At completion of the **ask** statement, all the variables mentioned in the variable name list exist. If they did not exist before this execution, they are created. If the user clicked on button **Cancel**, existing variables keep their initial values while newly created variables receive an empty value.

The message text may play several roles: (talkative) box title, instructions, warning, etc. In some cases, the way it is displayed may be felt a bit shy, particularly for a title or an important warning. To emphasize the message, we can ask it to appear in boldface characters (Figure 12.2). This is obtained through special directive **/b** at the beginning of the message bloc, as illustrated in Script 12.2.

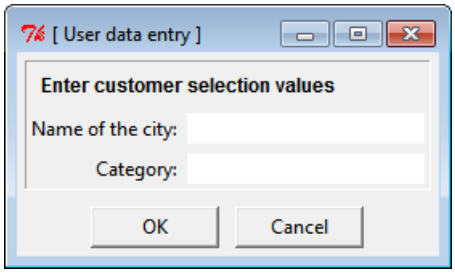



**Figure 12.2 -** Simple data entry box with a boldface message

```
ask city,cat = [/b Enter customer selection values]
               Name of the city: | Category:;
```

**Script 12.2 -** Emphasizing the message

Figure 12.3 shows a larger data entry box, comprising six data entry fields and a multi-lines message field (Script 12.3).

```
ask cid,nam,add,cit,cat,acc = [CUSTOMER data entry.
     @n@tFill the following fields,@n@tCat is optional]
     CustID|Name|Address|City|Cat|Account;
```

**Script 12.3 -** Asking the user to provide a set of data [Chapter-12-02.sql]

The labels are separated by symbols "|". This implies that this character cannot appear in labels. If such is the case, this symbol can be replaced by setting parameter `label-separator` to another default value in file **SQLfast.ini**:

```
label-separator = /
```

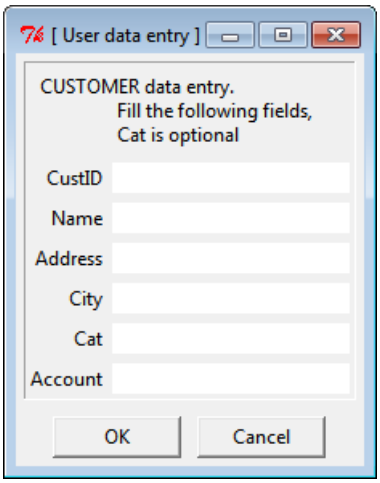



**Figure 12.3 -** Standard data entry box

The separator symbol can also be changed dynamically, through the **`parameter`** statement. It is illustrated in Script 12.4.

```
parameter label-separator = /;
ask cid,nam = [CUSTOMER data] CustID / Name;
parameter label-separator = default;
ask cid,nam = [CUSTOMER data] CustID | Name;
```

**Script 12.4 -** Dynamically changing the label separator symbol [Chapter-12-03.sql]

The length of the entry fields, that the number of characters that are visible, is a parameter defined by system variable **entryWidth**, set in initialization file SQLfast.ini:

```
entryWidth = 30
```

This value can also be consulted and set dynamically, from any script:

```
set currentEntryWidth = $entryWidth$;
set entryWidth = 65;
```

In some case, we would like to set the entry field width for a definite entry box, without modifying the global system variable. This can be done through parameter **/x**, as follows:

```
ask cid,nam = [/b/x65CUSTOMER data] CustID | Name;
```

### Note on keyboard equivalent of buttons

In most dialog boxes[2], hitting the **Enter** key has the same effect as clicking on button **OK** and key **Esc** is equivalent to button **Cancel**.

## 12.3 Controlling user action

When a dialog box is displayed, the user is expected to fill the fields, then to click on button **OK**. However, he can also click on button **Cancel** or, equivalently, close the dialog window. To capture this action, an SQLfast system variable is set when the box closes: **DIALOGbutton**. Its value is **OK** when button **OK** was clicked on, and **Cancel** otherwise. This value can be checked as illustrated in Script 12.5.

```
ask cit = Name of the city:;
if ('$DIALOGbutton$' = 'OK');
   showMessage City: $cit$;
else;
   showMessage No city;
endif;
```

**Script 12.5 -** Checking user action [Chapter-12-04.sql]

## 12.4 Update a data set

A standard data entry box displays empty entry fields (Figure 12.3). A variant of the **ask** statement, namely **ask-u** (**-u** for update), copies the values of the target variables in each entry field before submitting the entry box to the user. The latter can change some of the values instead of entering all of them from scratch.

2. **askText** is the main exception

Script 12.6 shows how an **ask-u** statement is used to modify columns Address and City of a selected row of table CUSTOMER (Figure 12.4).

```
set cid = C400;
extract adr,city = select Address,City from CUSTOMER
                   where  CustID = '$cid$';
ask-u adr,city = [/bUpdate address of customer '$cid$']
      New address:|New city:;
update CUSTOMER set Address = '$adr$', City = '$city$'
where  CustID = '$cid$';
```

**Script 12.6 -** Use of *ask-update* data entry box [Chapter-12-05.sql]

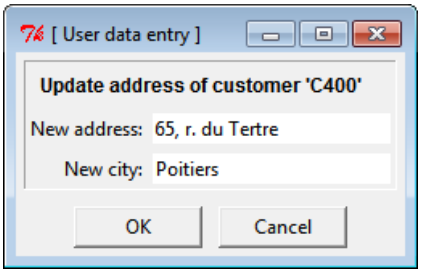



**Figure 12.4 -** Initial state of an entry box for data update

## 12.5 Text entry box

The standard data entry box is not suited to input a large amount of text. Even though a long character string can be entered, only a short part of it appears in the entry field.

The *text entry box* is the best way to let the user enter free text, possibly comprising many long lines. Of course, only a limited part of such long text appears in the entry box, but the latter, with its 20 lines of 64 characters, plus vertical and horizontal scrollbars, is more than comfortable, even to enter a full novel.

The text entry box is generated by statement **askText**, that comes in two variants: **askText** that acquires a new text and **askText-u**, that invites the user to update an existing text. The dialog box comprises a message field and a text entry field.

In the example of Figure 12.5, the user is asked to complete a document the paragraph names of which are already included. The user has to fill these paragraphs according to the directives of the message field then to click on button **OK** (or **Cancel** to drop the update).

**askText** specifies the target variable and the text to display in the message field. The statement syntax is simpler than that of the **ask** statement: only one target variable name and no field label.

We must notice something special in the initial text set by statement **set** `vitae = ...`. This text includes several sequences **\n**, that seem to force the display to go to next line in the text entry field. This is a common convention to say *new line*, when we don't mind the platform on which our script is executed. Indeed, the low level code sequence to go to next line is different for MSDos, MS Windows, Linux or Mac OS platforms. These **\n**'s are converted into the code sequence specific to the machine on which we are working.

We must note that SQLfast control symbols such as **@n** and **@t** are not preprocessed when initializing the text entry field. This text is a sort of raw *external material* on which SQLfast applies no specific treatment.

```
set vitae = IDENTITY\n\nEDUCATION\n\nWORK EXPERIENCE
            \n\nSKILLS\n\nREFERENCES;
askText-u vitae = [Enter the curriculum vitae of employee
            $empName$@nComplete the following paragraphs:];
```

**Script 12.7 -** Generating a text entry box [Chapter-12-06.sql]

When a text line is longer than the width of the text field, the part that goes beyond the right border is hidden, but can be seen by sliding rightward the handle of the bottom horizontal scrollbar.

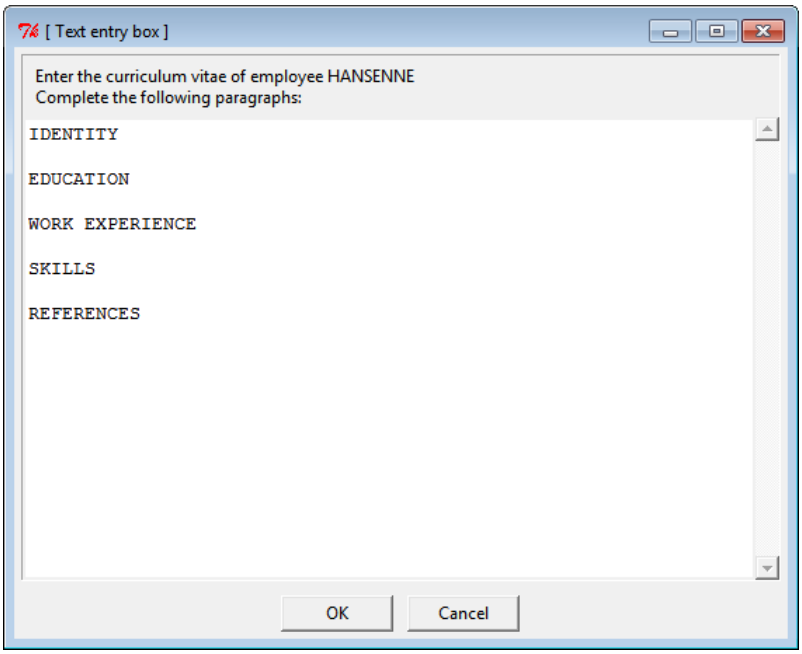



**Figure 12.5 -** A pre-filled form in a text entry box

The layout of the source text is respected but reading this text can be a bit tedious. Another way to cope with long lines is to ask that they continue at the next line of the field once they reach the right side. The aspect of the text changes, but it is completely contained in the field space (at least horizontally!) This technique is called *word* or *line wrapping*.

The way long lines are displayed depends on parameter **`textwrap`**. Its default value is **`Y`**, meaning that lines must be wrapped, but this can be changed in the **SQLfast.ini** initialization file through setting

```
textwrap = N
```

This default behavior can be dynamically changed for the current text field through display parameter **`/w`** specified in the message block **`[]`**, where **`w0`** means no wrapping and **`w1`** means line wrapping:

```
askText v = [/w0 Message: ...];
```

The standard size of text fields is 20 lines of 64 characters. If we find this space too small or too large, there are two ways to change these figures.

First, the standard size can be specified explicitly in the **SQLfast.ini** initialization file through parameters **`stdTextX`** and **`stdTextY`**:[3]

```
stdTextX = 72
stdTextY = 24
```

In addition, this default size can be replaced by a custom size for each individual dialog box, for example to a more comfortable 36 lines of 80 characters. To this aim, we state display parameters for the current text field as follows, where parameter **/y** specifies the field height (in lines) and **/x**, the line size (in characters)[4]:

```
askText v = [/y36/x80 Message];
```

Figure 12.6 shows the aspect of a small text box of 4 lines of 48 characters

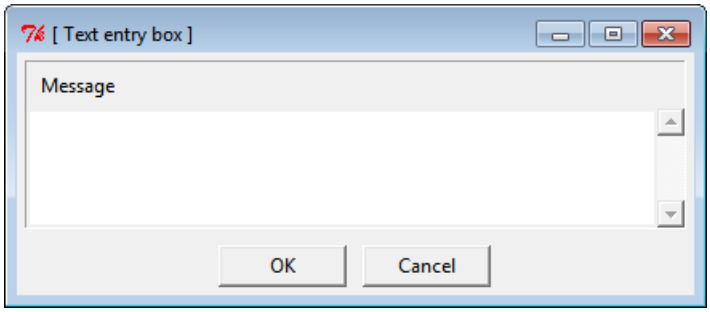



**Figure 12.6 -** A smaller text window

For clarity sake, it is recommended to end the list of parameters by a space character or by symbol `]` if the message is empty.

In fact, the rules for `askText` are more flexible. This space is optional, as well as the `[]` block and even the `=` sign, as long as no ambiguity arises.

So, the following form is also a valid statement.

```
askText v;
```

## Note on keyboard equivalent of buttons

The behavior of the **Enter** key in a text entry box is different from that in other dialog boxes. Instead of quitting the dialogue, it inserts a new line in the text. To close the box, one must click on button **OK**. The **Esc** key is still active however.

3. Actually, `stdTextX` and `stdTextY` are system variables that can also be set and used as any user variable.
4. **x** is adjusted to fall in range [8-132] and **y** in [1-60].

### About message delimitation and display parameters

We have learned that a message is specified in a **[]** bloc, possibly starting with a list of parameters. But what if the message itself also includes symbols **[** and/or **]**? Detecting the end of the message bloc may become tricky. For example, in statement

```
ask x,y = [Indiquer les coordonnées [en mètres] :] X | Y
```

we must ignore the first **]**, which belongs to the message, and only consider the second one[5]. We could observe that, in this example, these symbols are *paired*, and that it would suffice to search for the **]** symbol that matches the opening **[**, ignoring the internals pairs of **[]** symbols.

However this hypothesis is really too strict! So, we will adopt a simple convention if internal **]** are needed:

1. the message closing symbol **]** is followed by a space character, unless it is the last character of the statement
2. an internal **]** character is not followed by a space character; if needed, we will use **@s** control symbol to specify one space.

So, to avoid any ambiguity if we really want a space after the first **]**, we must rewrite the last statement as follows:

```
ask x,y = [Indiquer les coordonnées [en mètres]@s:] X | Y
```

If all this seems complicated, don't worry, the SQLfast engine will quickly detect something wrong in your statement or the dialog box will appear quite weird on the screen!

What about the list of display parameters? Two simple rules only: *no space before* and *no space inside*. The following expressions are **correct**:

```
[/x68/b/y12Enter your comment]
[/b/y12/x68 Enter your comment]
```

while the following are **invalid** (they will just be ignored):

```
[ /x68/b/y12Enter your comment]
[/x68 /b /y12 Enter your comment]
```

## 12.6 Selecting one item among several

A *selection box* comprises an (optional) message field and a list of **buttons** the user is invited to check (Figure 12.7). Only one button can be checked. When the user

5. In French, a colon symbol is preceded by a space character

clicks on a button, the previously checked button is reset. These are called *radio buttons*, by analogy with (*pre-iPod*) radio sets in which one channel button can be pushed at a time.

Script 12.8 includes a **selectOne** statement that creates this selection box. It comprises a message, a list of labels that will be associated to the check buttons and the name of the target variable (**table**) that will receive the number (here from **1** to **4**) of the button that was checked by the user. If the user clicks on button **Cancel**, the box returns value **0**.

```
selectOne table =
    [Select the table to update or Cancel if none]
    CUSTOMER|CUSTORDER|DETAIL|PRODUCT;
goto update-$table$;
```

**Script 12.8 -** Selecting one name among four table names [Chapter-12-07.sql]

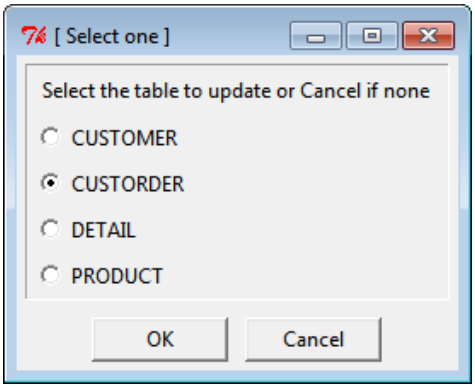



**Figure 12.7 -** Selecting a table to update

The target variable is created if it did not exist before select box building. SQLfast provides a variant of the **selectOne** statement (namely **selectOne-u**) that initializes the checked button before submitting the selection box to the user (Script 12.9).

```
set table = 2;
selectOne-u table =
    [Select the table to update or Cancel if none]
    CUSTOMER|CUSTORDER|DETAIL|PRODUCT;
```

**Script 12.9 -** Preselection of item 2

If, instead of the number of the button selected, we would consider receiving its name, we could use the procedure of Script 12.10.

```
set tabList = CUSTOMER|PRODUCT|CUSTORDER|DETAIL;
selectOne tabNum = [Select a table to update] $tabList$;
compute tabName = item('$tabList$',$tabNum$,'|');
write Table $tabName$ has been selected.;
```

**Script 12.10 -** Extracting the name of the button checked [Chapter-12-07bis.sql]

Function **item**(list,n,separator) returns the $n^{th}$ item of list list, where items are separated by symbols separator.

## 12.7 Selecting items among several

The *multi-choice selection box* is quite similar to the former one, except that it allows more than one button to be checked. It includes a message field and a list of check buttons, each being accompanied by a label (Figure 12.8).

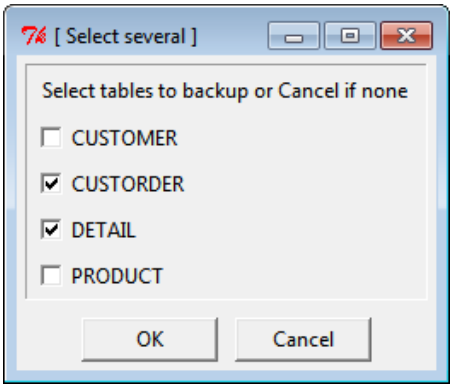



**Figure 12.8 -** Selecting tables to backup

A multi-choice box is created by a **selectMany** statement. It specifies the text of the message and a list of labels. It also specifies the list of names of the variables that will receive the state of the buttons when the user clicks on button **OK**. Each variable is associated to one button. It receives value **1** if the button has been checked and **0** if the button is unchecked.

At completion of the **selectMany** statement, all variables mentioned in the variable name list exist. If they did not exist before the execution, they are created. If the user clicked on button **Cancel**, existing variables keep their initial values while newly created variables receive value **0**.

```
selectMany C,O,D,P =
    [Select tables to backup or Cancel if none]
    CUSTOMER|CUSTORDER|DETAIL|PRODUCT;
write Tables to update: $C$,$O$,$D$,$P$;
```

**Script 12.11 -** Selecting tables [Chapter-12-08.sql]

SQLfast also provides a variant of the **selectMany** statements (namely **selectMany-u**) that initializes the check buttons before submitting the selection box to the user (Script 12.12).

```
set C = 0;
set O = 1;
set D = 1;
set P = 0;
selectMany-u C,O,D,P =
    [Select tables to backup or Cancel if none]
    CUSTOMER|CUSTORDER|DETAIL|PRODUCT;
```

**Script 12.12 -** Preselecting tables CUSTOMER and DETAIL [Chapter-12-09.sql]

To convert the target variable values (`0`,`1`) into table names, we can use the procedure of Script 12.13.

```
set tabList = CUSTOMER|PRODUCT|CUSTORDER|DETAIL;
selectMany t1,t2,t3,t4 = [Select tables to backup] $tabList$;
for tabNum = [1,4];
   if ($t$tabNum$$ = 0) next;
   compute tabName = item('$tabList$',$tabNum$,'|');
   write Table $tabName$ has been selected.;
endfor;
```

**Script 12.13 -** Extracting the names of the buttons checked [Chapter-12-10.sql

Expression `$t$tabNum$$` comprises two substitutions. Let us suppose that variable `tabNum` contains value `3`:

- `$t$tabNum$$` is first transformed into `$t3$`,
- `$t3$` is then transformed into `1` if the third button was checked and `0` otherwise.

## 12.8 List boxes

The **selectMany** command is quite fine to select one or several items in a short list of candidate elements. The **selectList** commands, which come in two variants, will be used instead when this list is large or when the order of the items selected is significant. Let us examine the first form, that asks the user to select a subset of customer id's:

```
openDB ORDERS.db;
extract list = select group_concat(CustID) from CUSTOMER;
selectList select = [Select customers] [($list$)];
write $select$;
```

**Script 12.14 -**

With the selection of Figure 12.9, this box will return this list:

```
B512,C123,C400
```

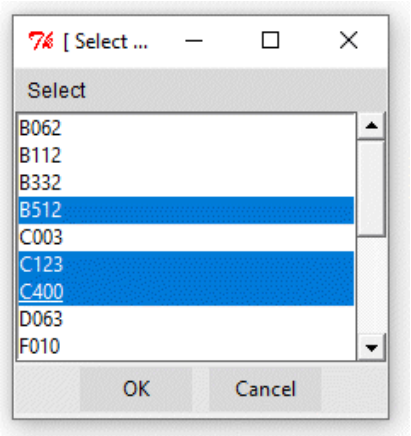



**Figure 12.9 -** Selecting items from a list box

The values of the list are separated by commas[6]. We observe that the order in which the result is produced is that of the list box. This may not fit the requirements of the user, who could want to receive the items of the list in his own order. In such a case, the **selectList2** command its more convenient.

This variant uses two list boxes. The left one shows the items that have not been selected yet while the right one shows those that have already been selected.

6. Actually, this separator is specified by system variable **constSeparator**, set in the SQLfast.ini file or by the script. The default separator is the comma.

This box comprises a series of buttons to move items from one list box to the other. The selection shown in Figure 12.10 will produce the following result:

```
C400,C123,B512
```

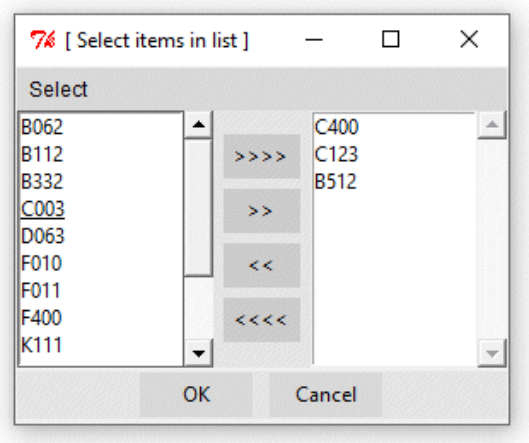



**Figure 12.10 -** Selecting items in a user-defined order

## 12.9 Summary

SQLfast offers six elementary data acquisition dialog boxes:

- **`askData`** (or **`ask`**): collection of entry fields. Returns one value per entry field.
- **`askText`**: one text field. Returns a text value.
- **`selectOne`**: collection of radio buttons. Returns a button number.
- **`selectMany`**: collection of check buttons. Returns a collection of 0/1 values.
- **`selectList`**: simple list box. Returns a list of selected values in the default order.
- **`selectList2`**: double list box. Returns a list of selected values in the user-defined order.

For all these statements, there exists a variant (suffix **`-u`**) with which pre-existing values are submitted to the user, who is invited to modify them if desired.

- Each statement comprises a list of target variables (only one variable for **`askText`** and **`selectOne`**), an optional text to display in the message field (enclosed between **`[`** and **`]`**) and a list of field labels (except for **`askText`**).

- When separated by symbol |, the elements of the box are placed vertically. This symbol can be changed through statement **parameter** label-separator). Separator | | places the next element on top of the next column.
- The message may include display parameters: **/b** (message in boldface), **/w** (line wrapping in **askText**), **/y** (number of line in **askText**) and **/x** (number of characters per line in **askText** and in **askData**. The line wrapping mode can also be specified by parameter **textwrap**. Default size of the text field is specified through system variables **stdTextY** and **stdTextX**. **textwrap**, **stdTextY** and **stdTextY** are specified in initialization file **SQLfast.ini**.
- Messages and labels may include format control characters: **@n** (newline), **@t** (tab), **@s** (space character), **@S5** (**5** space characters), **@@** (means single **@**).
- Missing target variables are created before dialogue execution.
- When the dialogue is cancelled (button **Cancel**), the variables keep their initial values.
- The label of the button through which the box was closed is specified in system variable **DIALOGbutton**.

# Chapter 13

# Displaying data

**Objective**: This chapter describes *read-only* dialog boxes through which a script merely displays data on the screen. They include the message box, simplified versions of **`ask`** and **`askText`** dialog boxes, and two graphical display boxes, one for bitmap images and the other one for still and animated vector graphics.

**Keywords**: read-only dialog box, message box, bitmap image, still vector graphics, animated vector graphics, eps, gif, SQLdraw language, SQLtuto language, help document, custom buttons

**Statements and functions**: showMessage, showData, showText, showPicture, showDrawing, BfileToVariable, @, displayHelp, close Help

**System variables and parameters**: DIALOGbutton, buttonYES, button NO, infoFont, infoWeight, infoWrap, maxPictureY, maxPictureY

## 13.1 Information message box

An SQLfast script can suspend its execution to display information in a simple message box (Figure 13.1). The execution resumes when the user clicks on one of the buttons **OK** or **Cancel**.

This box is opened by statement **`showMessage`** (or simply **`pause`**, which is a synonym). Its only argument is a character string to display. The latter may span several lines and include variables. Forced new line and alignment can be defined

through control characters **@n**, **@t** or **@s**, just like in the other dialog boxes (see Section 12.2).

The message box of Figure 13.1 was created by Script 13.1, that shows how variables and alignment are used to defined a neat and readable layout.

A message box will be used to provide the user with some important information about the execution of the script, such as errors or special conditions. It can also be used to let the user perform some required action, such as copying a file, renaming a file or plugging an external device.

```
set Tab1 = CUSTOMER;
set Tab2 = PRODUCT;
showMessage You have selected tables:@n- $Tab1$@n- $Tab2$.;
```

**Script 13.1 -** The **showMessage** (or **pause**) statement displays a simple message box

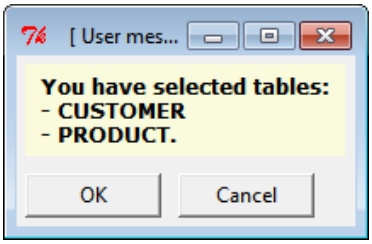



**Figure 13.1 -** Simple message box

The message of **showMessage** can span several lines in the script. However, it must be taken into account that, when it rebuilds the message, the SQLfast engine inserts one space character between two consecutive lines.

The font used to display the message is controlled by three system variables:

- `infoFont = verdana,9`
  font family and font size
- `infoWrap = 1`
  whether long lines are wrapped (1) or not (0)
- `infoWeight = 0`
  boldface (1) or normal (0)

## Note on the buttons of the display boxes

All the boxes created in this chapter comprise two buttons; **OK** and **Cancel**. Both have the same behavior: closing the box. However, the name of the closing button used is stored in system variable **DIALOGbutton**, so that the display box can also be used to ask users a binary question (Yes/No) to which they can answer through the closing button (OK = Yes, Cancel = No).

These buttons can be renamed to make them more expressive. See Section 14.9.

## 13.2 Displaying a list of values on the screen

Statement **showData** can be seen as a simplified version of **ask-u**. It comprises a list of variables, the contents of which is to be displayed, an optional message, that appears in the message field and a list of labels (Script 13.2 and Figure 13.2). The variable values are displayed but they cannot be modified by users. Parameter **/x** in the message field specified the width of the value fields.

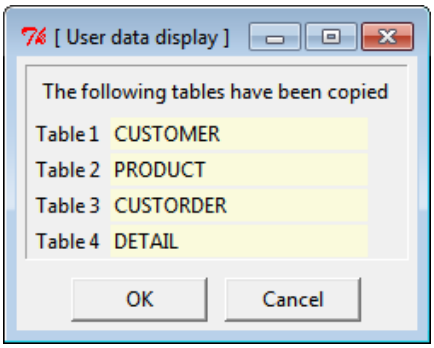



**Figure 13.2 -** Displaying a set of data

```
set T1 = CUSTOMER;
set T2 = PRODUCT;
set T3 = CUSTORDER;
set T4 = DETAIL;
showData T1,T2,T3,T4 = [The following tables have been copied]
                        Table 1|Table 2|Table 3|Table 4;
```

**Script 13.2 -** Script to display a set of data [Chapter-13-01.sql]

## 13.3 Displaying a text

Statement **showText**, a variant of **askText-u**, displays a text contained in a variable. The way long lines appear in the text display field depends on parameter **textwrap** (and its variant **/w**), as for **askText** statement (Figure 13.3, Script 13.3).

The message field is optional. It can include display parameters **/b**, **/w**, **/x** and **/y**. The last two parameters define the initial size of the text area.

```
showText TEXT;
```

```
showText TEXT = [/b/w1/x72/y24SQLfast tutorial];
```

```
set TEXT = In some cases, a script just wants to display
information without letting the user modify it. Statement
showMessage, that opens a simple message box and described in
Section 13.2, meets such requirement. For more complex data to
display, we will use statements showData and showText.;
showText TEXT = [Final version of the introduction];
```

**Script 13.3 -** Script to display a text [Chapter-13-02.sql]

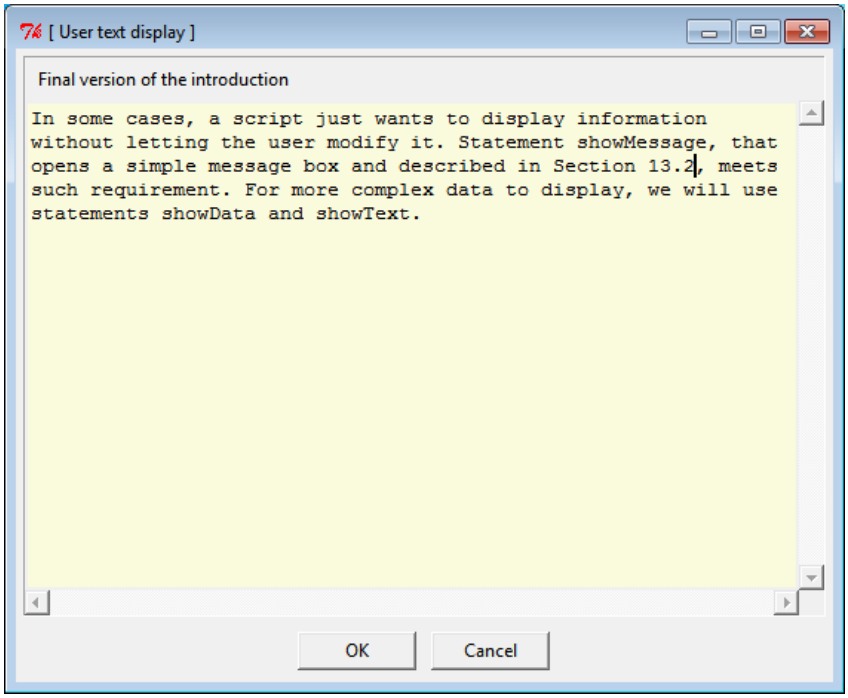



**Figure 13.3 -** Displaying a text

## 13.4 Displaying a picture

The contents of a **gif**, **jpeg** (or **jpg**), tiff, bmp file (and other standard formats) can be displayed through the **`showPicture`** statement. As for the **`showText`** statement, we specify the name of the source (file TTBS1.gif) and, optionally, a message (Script 13.4). The picture appears in a *picture box* (Figure 13.4).

```
showPicture Images/TTBS1.gif;
showPicture Images/TTBS1.gif = [/bThe Tiny Bang ...];
```

**Script 13.4 -** Displaying the image contained in a **gif** file [Chapter-13-03.sql]

The source picture can also be stored in a variable, as shown in Script 13.5. Statement **`BfileToVariable`** assigns an SQLfast variable the contents of a binary file, whatever its type (text, list of numbers, picture, video clip, etc.).

The converse operation, that is storing the value of a variable in a file, is carried out by statement **`variableToFile`**.

```
BfileToVariable pic = Images/TTBS1.gif;
showPicture pic = [/bThe Tiny Bang ...];
```

**Script 13.5 -** Displaying the image contained in a variable [Chapter-13-03.sql]

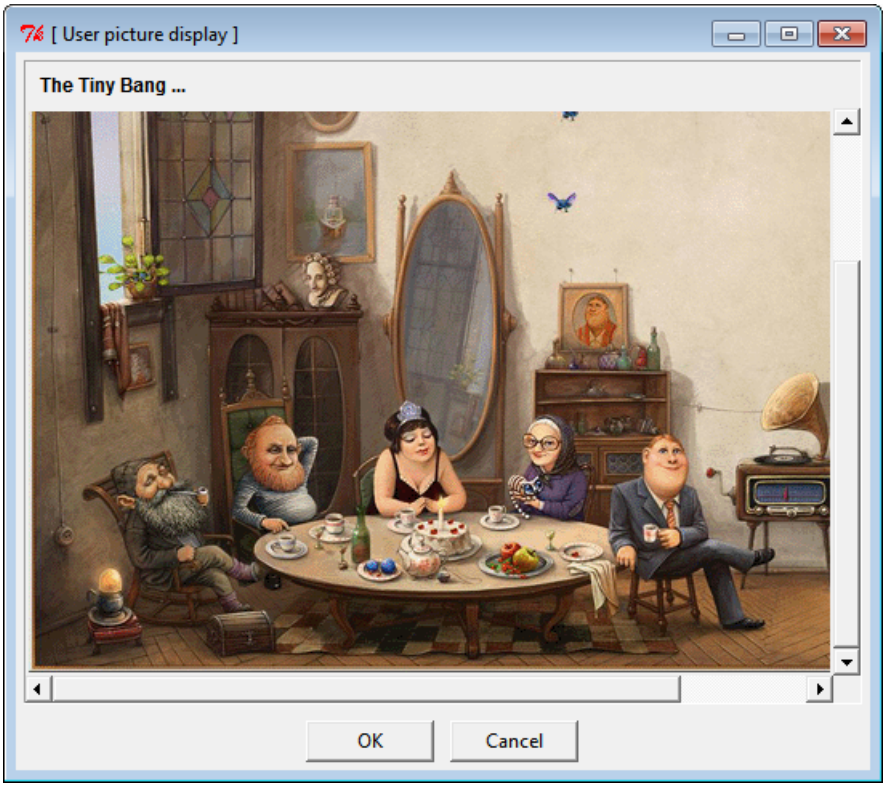



**Figure 13.4 -** An image in a picture box

The way the image appears in the picture space can be specified through display parameters. However, displaying a picture is not as straightforward as displaying a mere text since a picture can be quite large (possibly larger than the largest screen you can afford) and SQLfast cannot resize nor scale a picture dynamically.

The rules are the following.

1. The image has its own size, measured in horizontal and vertical pixels. For instance, the image shown in Figure 13.4 is 600x462 pixel large.
2. The image is displayed in a *picture box* of a (reasonable) definite size, e.g., 500x350 in Figure 13.4. If the picture box is smaller than the image, only a part of the latter appears in the box. In this case, one or two scroll-bars appear, that allow all the parts of the image to be examined.
3. However, a picture box cannot be larger than a *maximum size*. This size is defined by system variables **`maxPictureX`** and **`maxPictureY`**. Their default values are 640x480, but they can be changed in two ways, in the **SQLfast.ini** initialization file,

   ```
   maxPictureX = 400
   maxPictureY = 280
   ```

   and, dynamically, by assigning them new values in the script:

   ```
   set maxPictureX = 400;
   set maxPictureY = 280;
   ```
4. The message part of **`showPicture`** statement may specify display parameters **`/x`** and **`/y`** as in **`showText`**.

   However, their meaning is slightly different.
5. If display parameters **`/x`** and **`/y`** are ***not specified*** in the message part of the statement, then the size of the picture box is that of the source image, bounded by maximum values `maxPictureX` and `maxPictureY`.
6. If one or both display parameters **`/x`** and **`/y`** are ***specified*** in the message part of the statement, then the corresponding dimensions are forced to these values[1], independently of the actual dimensions of the picture and of the current maximum values. These parameters define a *fixed size* picture box in which the actual picture may appear too small or too large (in this case with scroll-bars).

The example of Figure 13.4 has been obtained by statement:

```
showPicture Images/TTBS1.gif = [/b/x500/y350The Tiny Bang ..];
```

All this may seem a bit complicated, so, let us summarize:

The picture is displayed in a picture box of a certain size. This size is that of the picture, unless it is defined by display parameters `/x` and `/y` of `showPicture`. If the picture box is too small, scroll bars appear. One can specify the maximum

1. **`x`** is adjusted to fall in [30-2000] interval and **`y`** in [30-1200].

size of the picture box through system variables `maxPictureY` and `maxPictureX`). However, parameters `/y` and `/x` can overwrite this maximum size.

### Notes

- While former versions of SQLfast could display **gif** images only, the recent versions (from v4.0) can process virtually all standard bitmap format. For more information, see the properties of the the PILLOW module.
- Animated **gif**'s are not rendered. To produce animated pictures, use the SQLdraw format instead.
- No, there is no **`askPicture`** statement! So far.

## 13.5 Displaying drawings

The SQLfast environment includes a graphical engine that draws vector graphics defined in a **SQLdraw** script, stored in a file or in a variable. An SQLdraw drawing is made up of shapes of different types: collections of points, polylines, polygons, rectangles, circles, ellipses, bitmaps and texts. An SQLdraw drawing can be still or animated, just like **gif** files.

Vector drawings can be used to show geographic maps (A Geographic Information System application), building maps, geometric shapes, electrical circuits, flowcharts, database schemas (of course!), genealogy trees (Kings of France application), data plot, statistical data. Through fast succession of shapes, elementary cartoons can also be built (nice example in Conway's Game of Life application), though this is not the primary goal of the SQLfast environment!

SQLdraw scripts can be written by hand or generated by SQLfast scripts. See for example case studies Conway's Game of Life, Kings of France and Agent-based modeling and simulation.

The SQLdraw processor and the SQLdraw language are described in Chapter 21.

The display box for vector graphics is similar to that of picture display shown in Script 13.4 and Figure 13.4).

Statement **`showDrawing`** specifies the source file or variable and an optional message (Script 13.6 and Figure 13.5).

The size of the drawing space is specified by commands **`area`** or **`picture`** of the SQLdraw script. The drawing is rendered in a graphical window the size of which is similar to that of picture box. This size can be explicitly set through display parameters **`/x`** and **`/y`**. If the graphical window is smaller than the drawing space, then one or two scroll bars are displayed.

```
showDrawing Files/WindEnergyUK2001.draw
           = [/b/x395/y245Wind energy consumption ...];

fileToVariable energy = Files/WindEnergyUK2001.draw;
showDrawing energy = [/b/x395/y245Wind energy consumption ...];
```

**Script 13.6 -** Displaying the vector graphics contained in an SQLdraw script file or in a variable

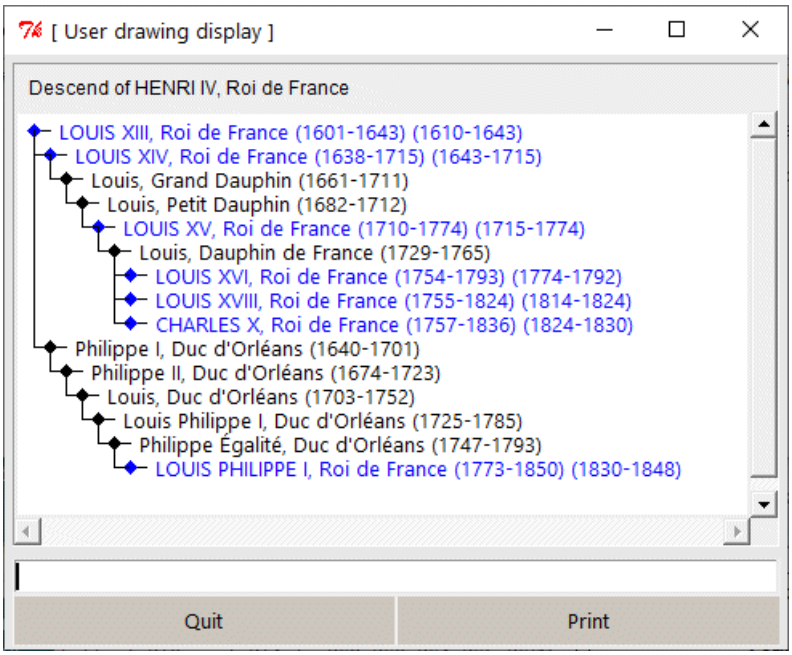



**Figure 13.5 -** Displaying vector graphics

The set of buttons of the box depends on the nature of the drawing. For still drawing, button **Quit** appears, that closes the box. For animated drawing, additional buttons **Play/Pause** and **Next** are provided, allowing the user to start, suspend and resume the animation (Script 13.7 and Figure 13.6). An animated drawing is recognized by the presence of commands **`step`** in the SQLdraw script.

Button **Print** generates a vector graphic file from the drawing in **`eps`** (*Embedded PostScript*) format. This format preserves the quality of the drawing whatever the zoom factor. It can be inserted in Word, Powerpoint, Framemaker documents. It can easily be transformed into other formats (bmp, tiff, jpeg, gif) by standard picture manipulation software, such as Adobe Acrobat, GIMP or Ghostview.

```
showDrawing Files/LIFEGAME-Glider.draw
            = [/x300/y300/bLIFE Game];
```

**Script 13.7 -** Displaying the evolution of a cellular automaton [Chapter-13-05.sql ]

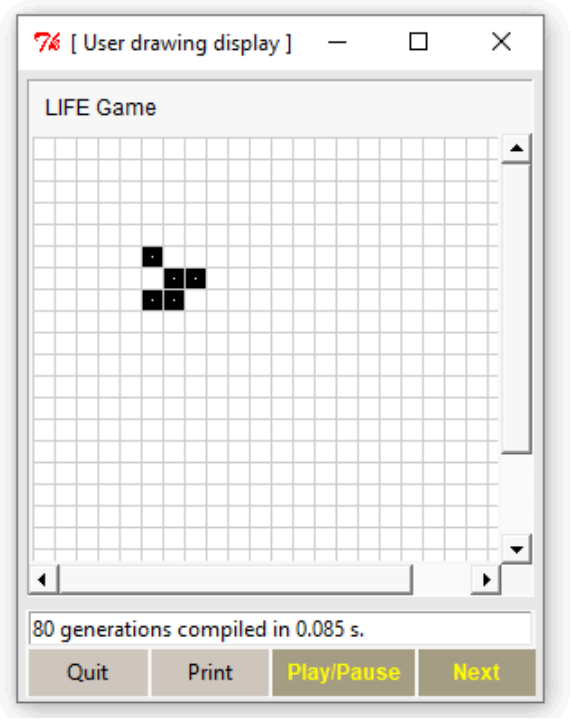



**Figure 13.6 -** One of the state of a Conway cellular automaton (*Glider*)

## 13.6 *Won't you please, please help me*

Some applications or even some individual dialog boxes can be so complex that a little help to fill their fields would be welcome. This help can be provided through the general message of dialog boxes, as we have shown in several examples examined so far. However, this technique is convenient for simple and concise recommendations only.

This help can also be provided by an independent document in, say, *Word*, *pure text* or *pdf* format displayed by the appropriate software through statement **`execProg`**.

In addition to these documentation means, SQLfast provides an integrated help quite similar to its *integrated tutorial subsystem*, but that can be controlled from within a script. Any help/tutorial document written in the **SQLtuto** language can be opened with statement **`displayHelp`**, then, when appropriate, closed through statement **`closeHelp`** (Script 13.8).

Such documents profit from the power of the tutorial language: text formatting, images, internal links, etc. The only limitation is that the directive **`link`**, which allows external links, is inactive.

```
displayHelp = How-to-fill-form08.tuto;
execSQL fillForm08.sql;
closeHelp;
```

**Script 13.8 -** Opening and closing a **Help** document from within a script]

## 13.7 Summary

SQLfast offers five elementary data display boxes:

- **`showMessage`** (or **`pause`**): displays a short text. Its unique argument is the text to display.
- **`showData`**: displays a collection of data items. It specifies the source variables, an optional message and a list of labels.
- **`showText`**: displays a text of arbitrary length. It specifies the source variable and an optional message. The display parameters are those of **`askText`** statement.
- **`showPicture`**: displays a **gif** bitmap picture, such as a photo. It specifies the source file or variable and an optional message. The picture is displayed in a picture box the size if which is that of the picture, unless it is defined by display parameters `/x` and `/y` in the message block. If the picture box is too small, scroll bars appear. The maximum size of the picture box can be specified by system variables `maxPictureY` and `maxPictureX`). However, parameters `/y` and `/x` can overwrite this maximum size.
- **`showDrawing`**: displays a vector drawing. It specifies the source file or variable and an optional message. The display parameters specify the size of the graphical window: **`/x`** and **`/y`**.

It also allows help documents to be displayed and closed from within a script.

- **`displayHelp`**: opens a Help window and displays an SQLtuto help document.
- **`closeHelp`**: closes the current Help window.

# Chapter 14

# Advanced dialogues

**Objective**: When simple dialog boxes described in Chapters 12 and 13 no longer meet the needs of an application, more sophisticated constructs can be used. In this chapter, we will learn some of them: multi-column boxes, field grouping through titles, empty labels, predefined value lists and composite dialog boxes.

**Keywords**: multicolumn dialog box, title, logical grouping, empty label, matrix data entry, menu, predefined value list, displayed values, returned values, composite dialog box, heterogeneous composite dialog box, dialog box style, dialog box color

**Statements and functions**: ask, selectOne, selectMany, ||, |||, {}, [()], [select ], !, askCombo

**System variables and parameters**: stylexxx, colorxxx, entryLength, constSeparator

## 14.1 Multicolumn data entry and selection

The dialog box layout defined by **`askData`** and **`selectOne/Many`** statements are sober and unsophisticated but quite satisfying when the box comprises a small number of homogeneous fields and buttons.

However, when the box includes a large number of components, and/or when these components form different logical groups, this simplistic layout can prove

inconvenient. SQLfast offers several layout refinements that can be used to design much more elegant (and therefore, more ergonomic) box structures. This section copes with the first of them, the *multi-column* organization.

Figures 14.1, 14.2 and 14.3 include the same information content as their original versions (Figures 12.3, 12.7 and 12.8) but are structured according to a multi-column layout of the entry fields and buttons.

Script 14.1, that creates these three dialog boxes, shows how to break the initial unique column into two shorter columns: while symbol | places the next field below the previous one, therefore building a column of fields, symbol || places it on top of a new column.

```
ask cid,nam,add,cit,cat,acc =
    [CUSTOMER data entry. @n@tFill the following fields,
                         @n@tCat is optional]
    CustID|Name|Address||City|Cat|Account;

selectOne table =
    [Select the table to update or Cancel if none]
    CUSTOMER|CUSTORDER||DETAIL|PRODUCT;

selectMany C,O,D,P =
    [Select tables to backup or Cancel if none]
    CUSTOMER|CUSTORDER||DETAIL|PRODUCT;
```

**Script 14.1 -** Scripts for multicolumn data entry and selection boxes [Chapter-14-01.sql]

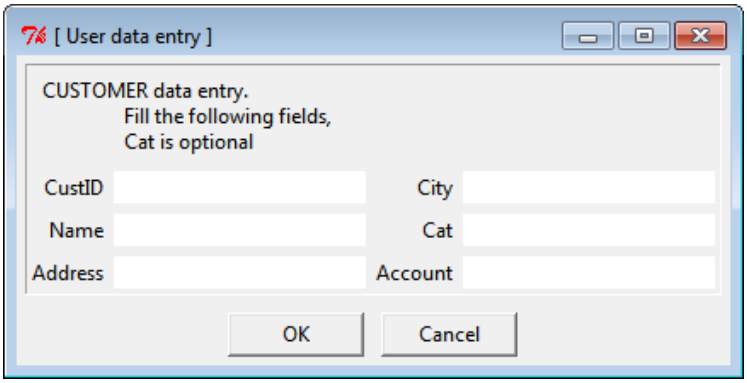



**Figure 14.1 -** Multicolumn data entry

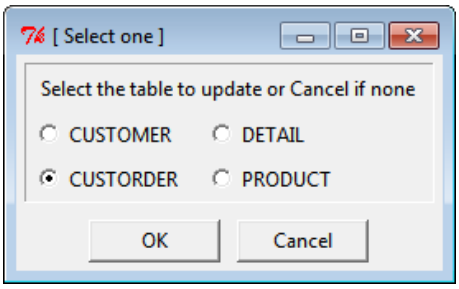



**Figure 14.2 -** Multicolumn single selection

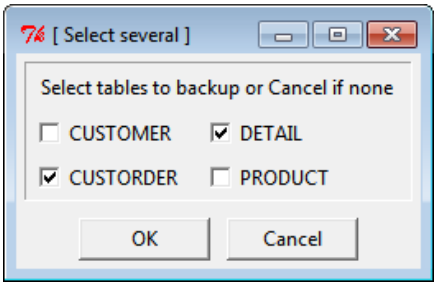



**Figure 14.3 -** Multicolumn multiple selection

## 14.2 Logical field grouping

The second improvement to component layout is the possibility to include **titles**. When combined with multicolumn organization, it allows complex dialog boxes to be defined. In particular, it can be used to define groups of components that have similar meaning.

The first example is shown in Figure 14.4 and is built by Script 14.2. The seven fields associated to the columns of (slightly modified) table **CUSTOMER** are grouped into *Identification data*, *Contact data* and *Buying data*, each category being collected into a specific column, headed with a title. A title is created in the **`ask`** statement by a label enclosed in symbols **{** and **}**. Label `{Contact data}` in Script 14.2 will not generate a labelled field, but will merely appear as an independent boldfaced label. This does not change the behavior of the **`ask`** and **`ask-u`** statements.

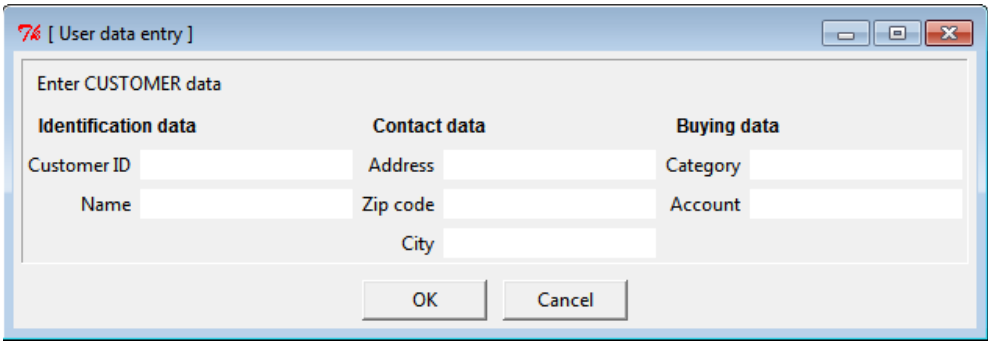



**Figure 14.4 -** Thematic field grouping of a data entry box

```
ask t1,t2,t3,t4,t5,t6,t7 = [Enter CUSTOMER data]
                {Identification data}|Customer ID|Name||
                {Contact data}|Address|Zip code|City||
                {Buying data}|Category|Account;
```

**Script 14.2 -** Introducing titles [Chapter-14-02.sql]

## 14.3 Logical button grouping

The insertion of titles also applies to selection dialog boxes. Figure 14.5 shows three improved arrangements of the boxes of Figures 12.8 and 14.3.

They are created by Script 14.3, that combines single/multi-column presentation with title insertion. Layout 14.5/top-right is obtained by a series of empty labels. Not sure that this is a model to follow! Using a *composite box* (Section 14.10) may be more appropriate.

Titles can also be placed at the right or at the left of the fields and buttons among which they appear. The box of Figure 14.6 uses a title as a substitute for the message field, to specify the role of the buttons. It is created by the following statement:

```
selectOne answer = {Apply modification?}||Yes||No;
```

```
selectMany t1,t2,t3,t4 = [Select one or several tables]
           {The customers}|CUSTOMER|
           {The orders}|CUSTORDER|DETAIL|
           {The products}|PRODUCT;

selectMany t1,t2,t3,t4 = [Select one or several tables]
           {The customers}|CUSTOMER|
           {The orders}|CUSTORDER|
           {The products}|PRODUCT||
           {}|{}|{}|
          DETAIL;

selectMany t1,t2,t3,t4 = [Select one or several tables]
           {The customers}|CUSTOMER||
           {The orders}|CUSTORDER|DETAIL||
           {The products}|PRODUCT;
```

**Script 14.3 -** Three scripts for button grouping [Chapter-14-03.sql]

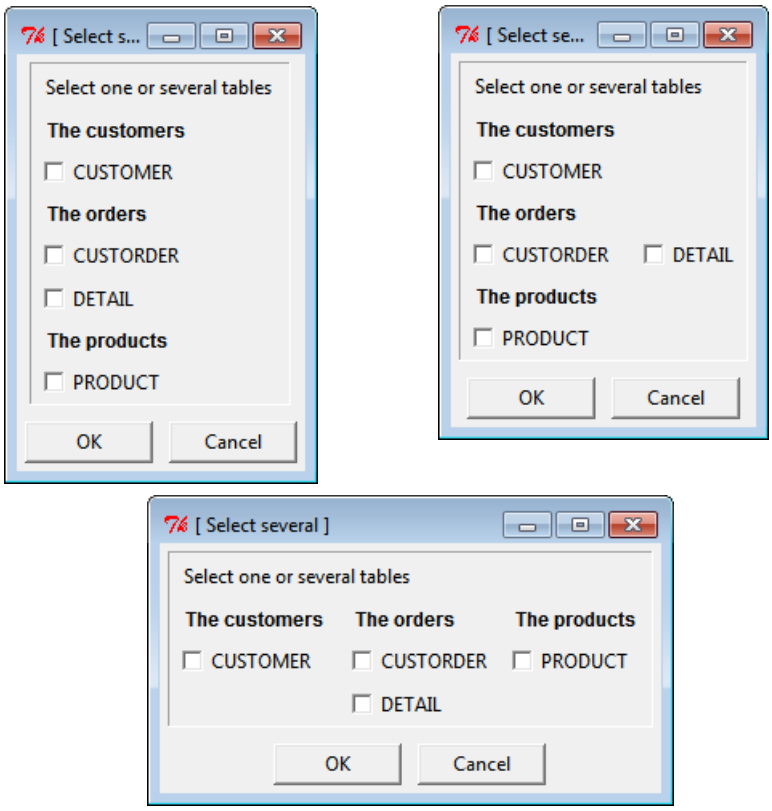



**Figure 14.5 -** Three ways of grouping items

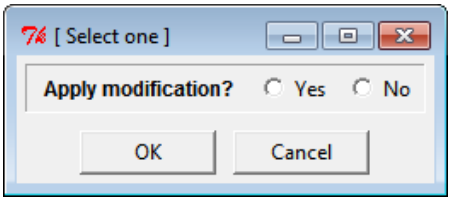



**Figure 14.6 -** Horizontal alignment of items

## 14.4 *No label please*!

The **`ask`**, **`selectOne`** and **`selectMany`** dialog statements associate a *label* to each field and button to specify its meaning. In some cases, we do not want all or some of the fields/buttons to be labelled. Figure 14.7 shows a box intended to capture data for a series of order details. The fields on the first column are labelled `line 1:` to `line 3:`, while the fields of the second column have no associated labels.

The most obvious way would be to specify *empty labels* for the last three fields. Unfortunately, this concept does not exist in SQLfast, since two successive `|` separators with *nothing* between them cannot be distinguished from the *next column* `||` operator.

On the other hand, a *one-space* string is not a valid label, because leading and trailing spaces are removed from labels. However, these spaces are removed **before** the substitution of control characters. So, specifying a *one-space* label as `@s`, actually generates a true *one-space* label. Finally, the dialogue manager is smart enough to empty labels comprising one space only.

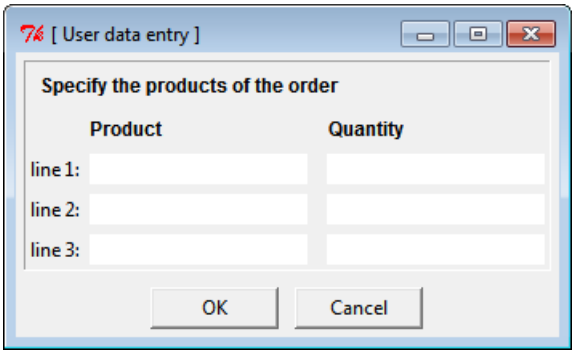



**Figure 14.7 -** Data entry box with some labels missing

Now, we are able to define the data entry box of Figure 14.7 through Script 14.4. Note control symbol **@S10** (*insert 10 spaces*) in the first column title, that aligns string **Product** on the fields (instead of on labels) by introducing a 10 space offset.

```
ask p1,p2,p3,q1,q2,q3 = [/bSpecify the products of the order]
                          {@S10Product}|line 1:|line 2:|line 3:
                        ||{Quantity}|@s|@s|@s;
```

**Script 14.4 -** Specifying empty labels as single characters [Chapter-14-04.sql]

Figure 14.8 shows the use of empty labels to design a two dimensional data entry box. By selecting one of the nine radio buttons, the user specifies two independent values, the *model* and the *color* of a bicycle, here (C, RED). The SQLfast statement that generates this dialog box is given in Script 14.5.

```
selectOne a = [Select a bicycle.
               Specify@n - the model (A,B,C)@n - the color]
               {A}|@s|@s|@s||{B}|@s|@s|@s||{C}|WHITE|RED|BLUE;
```

**Script 14.5 -** Matrix-based data entry

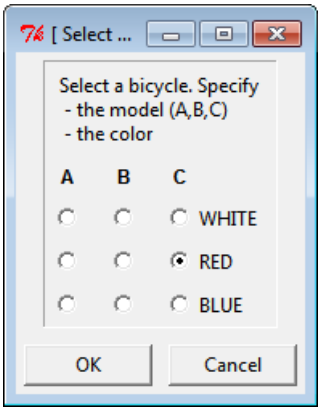



**Figure 14.8 -** Matrix-based data entry

This box returns a number from 1 to 9, which can be converted into *(model,color)* information as shown in Script 14.6.[1]

1. Function **items** is a simplified version of **item**: `items(S,n) = item(S,n,';')`

```
selectOne a = [Select a bicycle.
               Specify@n - the model (A,B,C)@n - the color]
               {A}|@s|@s|@s||{B}|@s|@s|@s||{C}|WHITE|RED|BLUE;
compute model = items('A;B;C',1 + ($a$ - 1) / 3);
compute color = items('WHITE;RED;BLUE',1 + ($a$ - 1) % 3);
write Bicycle ($model$, $color$) has been selected;
```

**Script 14.6 -** Extracting information (model, color) from button number [Chapter-14-05.sql]

## 14.5 Application: building a 2D menu

This example is a nice application of the two dimensional layout of a large number of buttons. The box of Figure 14.9, resulting from Script 14.7 presents a menu for a (fictitious) complex application that can perform 35 functions, grouped into 7 classes.

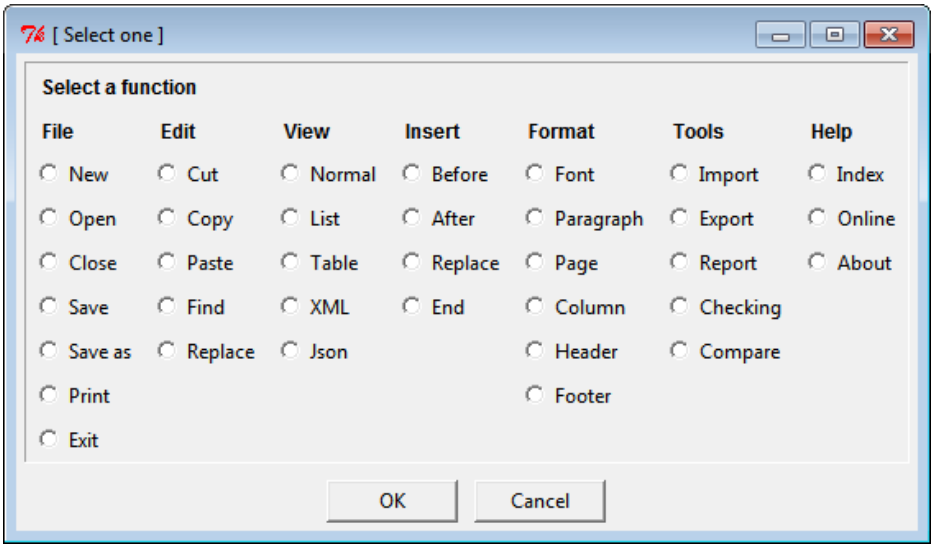



**Figure 14.9 -** Building a complex menu through a multicolumn selectOne statement

```
selectOne t = [/bSelect a function]
          {File}|New|Open|Close|Save|Save as|Print|Exit||
          {Edit}|Cut|Copy|Paste|Find|Replace||
          {View}|Normal|List|Table|XML|Json||
          {Insert}|Before|After|Replace|End||
          {Format}|Font|Paragraph|Page|Column|Header|Footer||
          {Tools}|Import|Export|Report|Checking|Compare||
          {Help}|Index|Online|About;
```

**Script 14.7 -** How to build a menu with the selectOne statement [Chapter-14-06.sql]

## 14.6 Data entry through selection in a predefined value list

In the standard dialog box generated by an **`ask`** statement, the user is invited to type full values into empty entry fields. In many situations, however, the set of possible values is predefined, so that the user just has to select the appropriate value instead of painfully typing it in the entry field.

This is the objective of a variant of the standard entry box, to which a list of possible values is attached[2]. Figure 14.10 shows, at the left side, the standard data entry box, where the user may type a value in field Cat. At the right side, a drop-down list presents suggested values `(B1,B2,C1,C2)` among which the user can make his choice. However, he can also choose to enter another value (e.g., **`c3`**) that is not included in the suggested list.

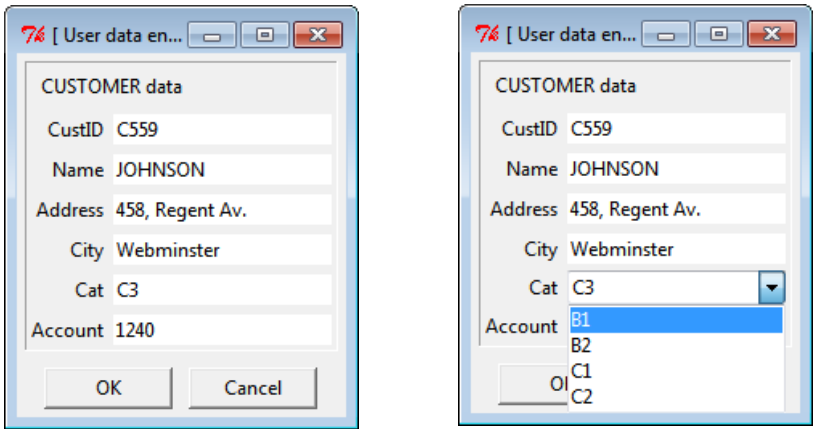



**Figure 14.10 -** In the right side box, the value of CAT **can** (but need not) **be chosen** in a predefined list

2. The usual technical name of this widget, that combines an entry field and a list-box, is **combo-box**. Not to be confused with the `askCombo` box we will study below.

This list is specified in the **ask** statement as a complement to the label, such as **[(B1,B2,C1,C2)]** in Script 14.8 (bottom). By default, the expression of the list follows the SQL syntax for a row of constants (as in SQL `insert-values` query): character strings are single-quoted, numeric values are unquoted and internal single-quotes (or apostrophes) are doubled:

```
('John''s home', 125.45, 'London', 12, '10, Downing Street')
```

```
ask cid,nam,add,cit,cat,acc =
    [CUSTOMER data] CustID|Name|Address|City
    |Cat|Account;

ask cid,nam,add,cit,cat,acc =
    [CUSTOMER data] CustID |Name|Address|City
    |Cat[(B1,B2,C1,C2)]|Account;
```

**Script 14.8 -** Modification of the standard ask statement (top) so that the value of CAT can now be chosen in a predefined list (bottom) [Chapter-14-07.sql]

Actually, the syntax of the list of values is defined by the following rules, that are more flexible:

- two successive values are separated by a comma[3]
- values **may be** single-quoted, be they numeric or character strings
- if a value includes commas, it **must be** single-quoted
- if a value includes a single-quote, the latter must be doubled
- leading and trailing spaces and quotes are ignored
- if the list of values does not comply with these rules, the engine does its best to extract a valid list.

With this feature, the user can **select** a predefined value or **type** its own value. It is possible to forbid the latter possibility and to force the user to select a predefined value (Figure 14.11). To do so, symbol **!** is inserted before the expression of the list, as in Script 14.9.

3. Actually, this separator is specified by system variable **constSeparator**, set in the SQLfast.ini file or by the script. The default separator is the comma.

```
ask cid,nam,add,cit,cat,acc =
    [CUSTOMER data] CustID |Name|Address|City
    |Cat[!(B1,B2,C1,C2)]|Account;
```

**Script 14.9 -** Symbol **!** indicates that the value of CAT must be chosen from the predefined list [Chapter-14-07.sql]

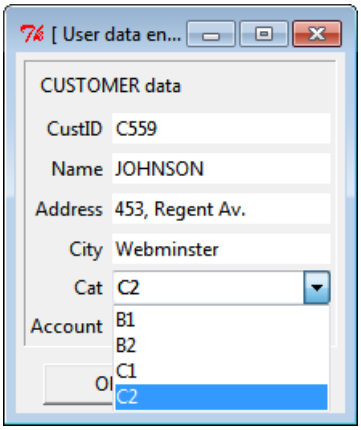



**Figure 14.11 -** The value of CAT **must be selected** from the predefined list

## 14.7 Predefined values from the database

The list of values can also be **extracted from the database**. A nice example is related to foreign keys. Entering the value of a foreign key is much easier if the user can choose it in the list of the values of the target primary key.

In Figure 14.12, the user must enter the three values required to insert a DETAIL row in the database:

- a value of OrdID that must exist in table CUSTORDER
- a value of ProdID that must exist in table PRODUCT
- a value of Qord

It is natural to associate with the first two fields the authorized values extracted from tables CUSTORDER and PRODUCT. Each value list is built by an SQL query, respectively "`select OrdID from CUSTORDER order by OrdID`" for field OrdID and "`select ProdID from PRODUCT order by ProdID`" for field ProdID.

This is what Script 14.10 shows. The three field names are "Order", "Product" and "Quantity". The appropriate SQL query is associated with each of the first two fields while a list of suggested values is provided for field Quantity.

To ensure the correctness of the two foreign keys, we force the user to **select** a value from the predefined list through the `!` symbol. For field Quantity, the user is free to select or to type a value.

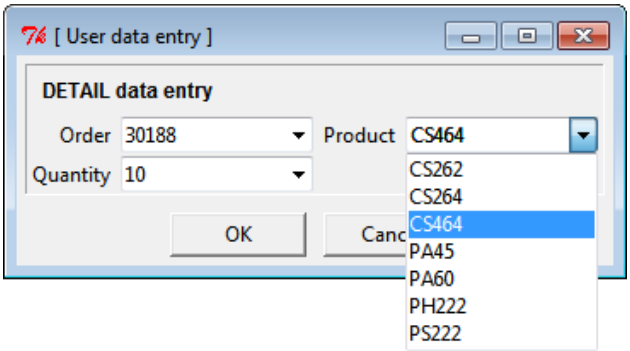



**Figure 14.12 -** The data entry box to insert a DETAIL row in the database

**Note**

Script 14.10 is illustrative but incomplete. If referential integrity is guaranteed[4], primary key uniqueness is unfortunately ignored. In case of violation, the script stops with an error message. It is easy to add the verification statements that control the uniqueness constraint.

```
openDB ORDERS.db;
ask oid,pid,q = [/bDETAIL data entry]
      Order[!select OrdID from CUSTORDER order by OrdID]
     |Quantity[(1,2,5,10,20,50,100,200,500,1000)]
    ||Product[!select ProdID from PRODUCT order by ProdID];
insert into DETAIL values ('$oid$','$pid$',$q$);
```

**Script 14.10 -** Elementary script to insert a row in table DETAIL [Chapter-14-08.sql]

4. At least if the script is the only process that works on the database. In some situations, referential integrity could be threaten (but not violated), for example if a concurrent process deletes the selected CUSTORDER row between the execution of the `ask` and `insert` statements. It is possible to prevent such conflict with an SQLite *exclusive transaction*. This statement does not exist in SQLfast but can be executed through a `generic` statement.

The **!** constraint entails an interesting issue. Let's suppose that:

- a predefined value list is associated with field **Order**, as in Script 14.10,
- the **ask** statement is in *update* mode (**ask-u**),
- an initial value is stored in variable **oid**,
- this value is not (or no longer) in the predefined value list, and therefore is invalid (say **X999**),
- the user doesn't want to change it before clicking on button **OK**.

First idea: the initial situation should be forbidden and cause an execution error. This is not the case, because SQLfast thinks that this could be a legitimate state of affair: this value could have been valid, though it isn't currently. It also thinks that what must be forbidden is to confirm this invalid value without changing it. Instead of triggering an error, SQLfast returns an empty value for this field. It is up to the script to manage this result.

## 14.8 Displayed *vs* returned values

Let us now consider a short script that allows the user to change the address and the city of a customer. The data entry box comprises three fields:

- identification of the customer (Customer)
- new address (Address)
- new city (City)

It would be nice to let the user select the customer from a predefined list extracted from the database, but we must admit that selecting a customer on the basis of its **CustID** value alone is not particularly easy. Some additional information, such as the customer name, could make the user more comfortable when selecting a customer. This is what Figure 14.13 shows.

The drop-down list presents values built from the *Id* and the *name* of customers. In addition, the list is sorted on customer names, which make customer selection much faster. Script 14.11 shows how this data entry box is generated. The values of the list result from SQL expression

```
Name || ' [' || CustID || ']'
```

in which symbol || denotes the operator that concatenates two character strings into a single one.

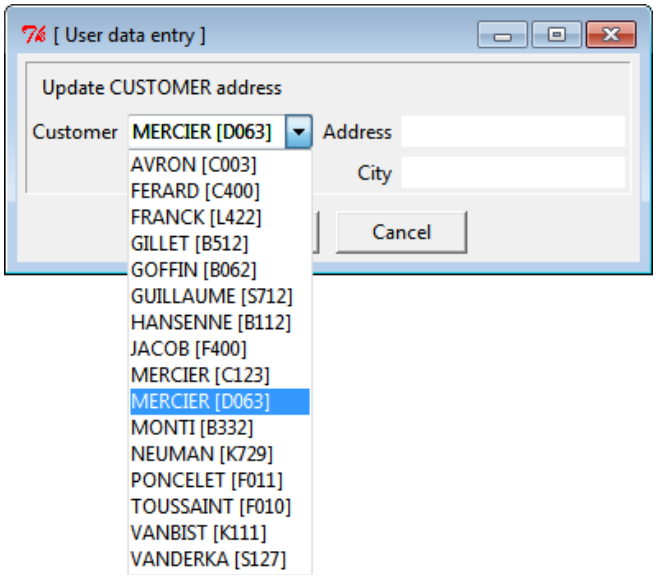



**Figure 14.13 -** Selecting a CustID value from a more expressive list

```
ask cid,adr,cit = [Update CUSTOMER address]
     Customer[!select Name||' ['||CustID||']'
                from   CUSTOMER order by Name]
   ||Address|City;
```

**Script 14.11 -** Generation of the value list of Figure 14.13 [Chapter-14-09.sql]

So far, so good. However, extracting CustID value **`D063`** from the returned value **`MERCIER [D063]`** is not that straightforward as witnesses Script 14.12.

Function **`substr2(s,i1,i2)`** returns the part of string **`s`** from position **`i1`** to position **`i2`**, those included. Function **`instr(s,r)`** returns the position of the first instance of string **`r`** in string **`s`**.

```
compute CustID = substr2('§cid§',instr('§cid§','[')+1,
                                 instr('§cid§',']')-1);
write Customer $CustID$ selected;
```

**Script 14.12 -** (Painfully) extracting the value of CustID from the value returned in **cid** by Script 14.11 [Chapter-14-09.sql]

SQLfast provides a more elegant and more general way to solve this issue by distinguishing the values that are *displayed* in the predefined list from the values that are *returned* in the target variables. Let us examine Script 14.13, in which the *select list* of the SQL query now comprises *two elements*:

```
select Name||' ['||CustID||']', CustID
from   CUSTOMER
order by Name
```

The first element (`Name||' ['||CustID||']'`) generates the list that will be displayed in the dialog box (as in Figure 14.13), and the second one (`CustID`) provides the values to be returned in variable **`cid`**. When a value is selected in the first list, the dialog box returns the value of the second list which is at the same position.

If the SQL query specifies only one *select element*, then the displayed and returned lists are the same, as in Figure 14.12.

```
ask cid,adr,cit = [Update CUSTOMER address]
     Customer[!select Name||' ['||CustID||']', CustID
                from CUSTOMER order by Name]
   ||Address|City;
```

**Script 14.13 -** Distinguishing displayed and returned values [Chapter-14-10.sql]

### Note

The use of delimiters **`[`** and **`]`** to specify predefined value lists implies that including these symbols in labels and in titles may confuse the SQLfast analyzer. If they are really needed, they must appear *paired* in labels and titles. Examples: "Address[1]" and "Comment [optional]" are valid. They can also appear in the value list expression, as shown in Script 14.13.

## 14.9 Customizing standard buttons

Most dialog boxes comprise two standard buttons, a *positive* one, generally labelled **OK** and a *negative* one, called **Cancel**. These names can be modified, either in the initialization file **SQLfast.ini**

```
buttonYES = Continue
buttonNO = Stop
```

or, dynamically by the script (Figure 14.14):

```
set buttonYES = I accept;
set buttonNO = I refuse;
```

```
showMessage Do you want to buy this book?;
if ('$DIALOGbutton$' = '$buttonNO$') stop;
execSQL ProcessBookBuying.sql;
```

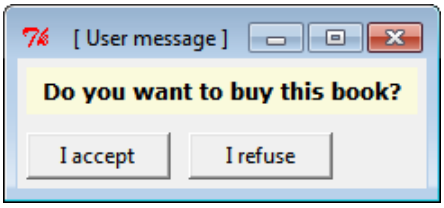



**Figure 14.14 -** ShowMessage box with custom buttons

If the name of one of these buttons is empty, this button will not appear in the next boxes (Figure 14.15):

```
set buttonYES = Continue;
set buttonNO = ;
showMessage This operation is cancelled;
```

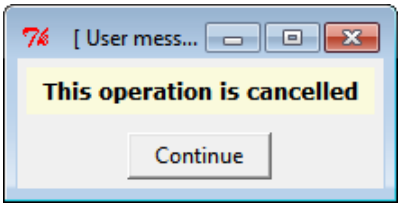



**Figure 14.15 -** Only one button is active

The default labels can be assigned either manually, or through menu Session > Close session.

## 14.10 Composite dialog boxes

Let us suppose that we want to register a new employee. To this aim, we ask the user to provide the basic employee data: *ID*, *name*, *address*, *city*, *date hired* and *function*, through an **`ask`** statement such as that of Script 14.14 (1st statement).

Then, we want to know whether the creation must be *immediate* or if it can be *delayed*. This is done via a **`selectOne`** statement (second statement of Script 14.14).

We must also know to which *services* the new employee could be sent: *Finance*, *Production* or *Personnel* (third statement of Script 14.14).

Finally, we would like to know, through a short free text, the *skills* of the employee (fourth statement of Script 14.14).

```
ask eid,nam,add,cit,dat,fun = [/bEnter employee data]
    Employee ID|Name|Address||City|Date hired|Function;
selectOne delay = [/bImmediate or deferred creation?]
    immediate || deferred;
selectMany fin,pro,per = [/bHost service]
    Finance|Production|Personnel;
askText ski = [/bSkills of the employee];
```

**Script 14.14 -** Four independent dialogues to acquire information on a new employee

The process of data acquisition is split into a sequence of four different dialogue steps, which certainly will be perceived inconsistent and tedious by users. It would be nice to aggregate these dialogues into a unique data entry box, that lets the user fill the fields in the order that suits her best. Figure 14.16 could be such a dialog box.

However, SQLfast cannot offer a model for each of the dialog boxes that would exactly fit the needs of any development. On the other hand, flexibility in building custom graphical interfaces has a high price, that is, mastering and programming, in a standard language, complex objects and algorithms, an effort that only seasoned programmers can afford.[5] For instance, a simple **`askText`** statement such as "**`askText`** `ski = [Skills of the employee]`" is equivalent to more than 60 Python instructions, most of them devoted to building the dialog box.

Fortunately, the SQLfast engine allows complex dialog boxes to be built from elementary building blocks generated by **`ask`**, **`selectOne`**, **`selectMany`**, **`askText`**, **`showData`**, **`showText`** and **`showPicture`** statements.

The statement that combines several dialog boxes is extremely simple. Before examining how the complex dialog box of Figure 14.16 can be built, let us first work out a simpler layout. We want to combine a two field data entry box with a box to select one item among two, as in Figure 14.17.

5. See document Code_Comparison_EN.pdf, available on the SQLlite website for a convincing example.

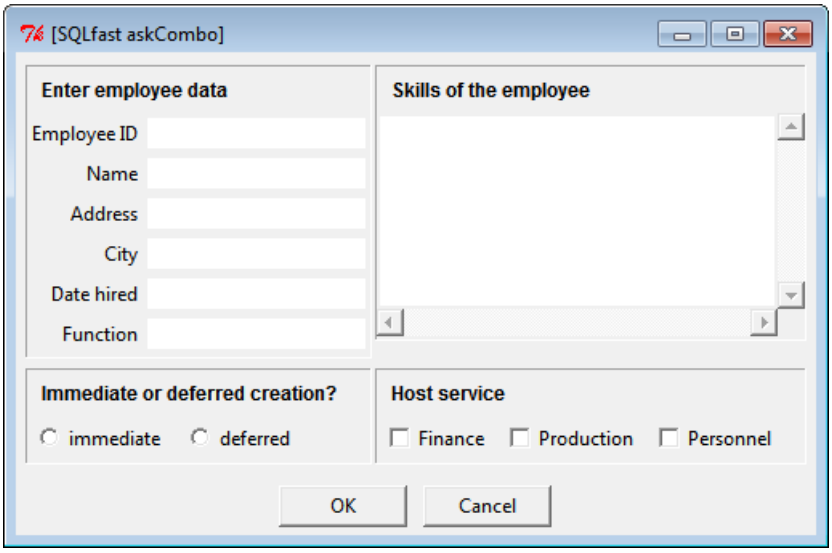



**Figure 14.16 -** A complex dialog box that, at first glance, couldn't be built in SQLfast

The first part of the dialog can be generated by statement:

```
ask a,b = [Me1] Lab1|Lab2
```

The second part requires a statement such as the following:

```
selectOne c = [Me2] Lab3|Lab4
```

The combined box is obtained by an **askCombo** statement that just includes these two elementary statements, each enclodes in square brackets (script in file Chapter-14-11.sql):

```
askCombo [ask a,b = [Me1] Lab1|Lab2]||[selectOne c = [Me2] Lab3|Lab4]
```

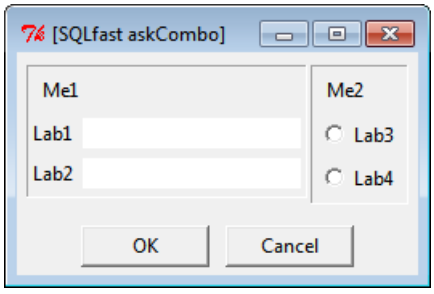



**Figure 14.17 -** A combined dialog box made up of a data entry box and a selection box

Each elementary statement is enclosed into `[` and `]`, and two consecutive statements are separated by symbol `|`, `||` or `|||`. Vertical alignment (symbol `|`) and column break (symbol `||`) work just like in label sequences in elementary dialog statements. Symbol `|||` is new. It will place the next box under the last one, but as far as possible at the left.

When this statement is executed, each variable `a`, `b`, `c` receives a value, unless the dialogue has been cancelled by the user.

To generate the large box of Figure 14.16, we proceed in the same way: we assemble the four elementary statements of Script 14.14 into a single **`askCombo`** statement. To make the script more readable, we first store the elementary statements in four variables **D**, **O**, **T** and **M**, from which we build the combined statement (Script 14.15).

This complex dialog box returns eleven values, stored in variables `eid`, `nam`, `add`, `cit`, `dat`, `fun`, `delay`, `fin`, `pro`, `per` and `ski`.

To give the dialog box a nice layout, we may have to reorder the elementary boxes and to tweak the different components (labels, messages, text width and height, spaces) until we get a satisfying result.

Each elementary box is displayed in a frame (a rectangular space) and the frames are aligned in rows and columns (Figure 14.19). All the frames of a row have the same height and all the frames of a column have the same width.

So far, we used to insert a message field in all the dialog boxes, in order to inform the user on the goal of the box or on the way its fields must be filled. Such a message can also be added to a composite box (Figure 14.18). It is specified in a `[]` block before the list of elementary boxes, as shown in Script 14.16.

```
set D = ask eid,nam,add,cit,dat,fun = [/bEnter employee data]
    Employee ID|Name|Address|City|Date hired|Function;
set O = selectOne delay = [/bImmediate or deferred creation?]
    immediate || deferred;
set T = askText ski = [/b/w0/y7/x26 Skills of the employee];
set M = selectMany fin,pro,per = [/bHost service]
    Finance||Production||Personnel;
askCombo [$D$]|[$O$]||[$T$]|[$M$];
```

**Script 14.15 -** Building the complex dialog box of Figure 14.16 [Chapter-14-12.sql]

```
askCombo [/b Recording an employee] [$D$]|[$O$]||[$T$]|[$M$];
```

**Script 14.16 -** Adding a message field to a composite dialog box [Chapter-14-12.sql]

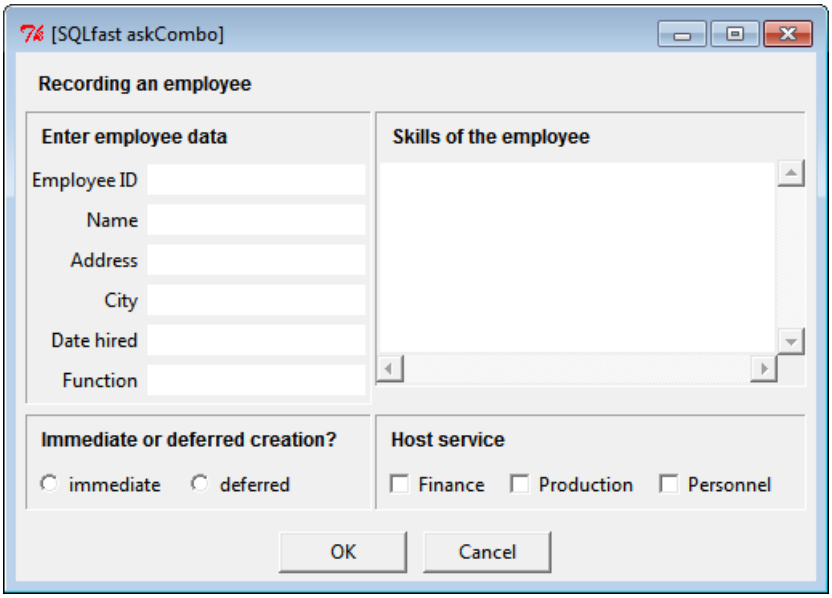



**Figure 14.18 -** A composite dialog box with a message field

## Notes

- Contrary to elementary dialog boxes, the definition of a composite dialog box may not include *titles*.
- A composite dialog box may comprise several components of the same kind.
- A composite box can be made up of elementary boxes of any kind, except `showMessage` (easily replaced by `showData` or `showText`) and `showDrawing`.

## 14.11 Heterogeneous composite boxes

In the composite layout, the box space is organized as a rectangular grid of **frames**. Each box is placed in a frame (Figure 14.19).

The fact that the boxes of a column all have the same width and the boxes of a row have the same height may prove too strong a constraint.

For instance, placing a small **`selectOne`** box just below a large **`askText`** box will produce an awkward composite box with wide wasted space. It would be better to place two or three small boxes below this large text box.

This can be done by using *heterogeneous composite boxes*, in which an elementary box can span more than one frame.

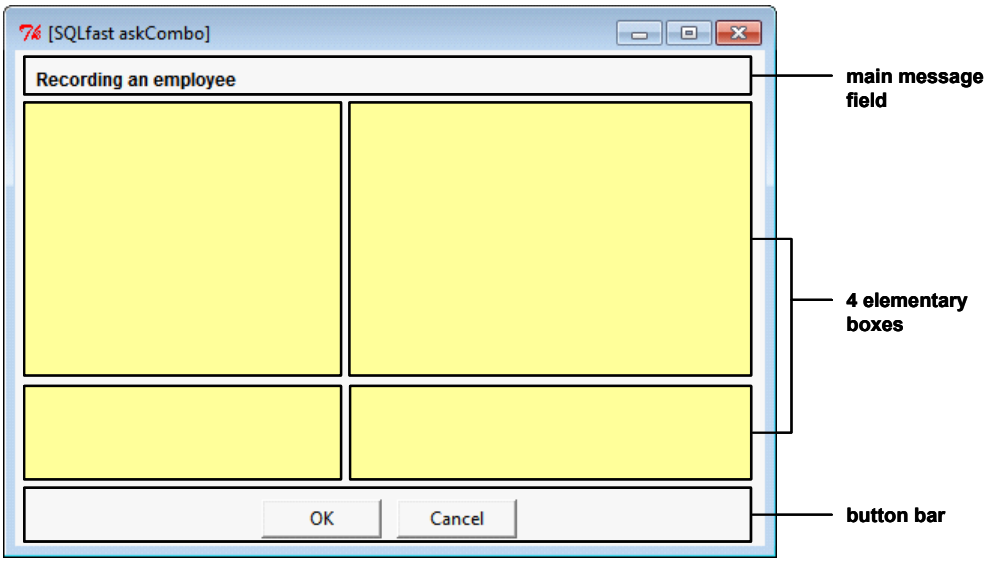



**Figure 14.19 -** The boxes are arranged in a rectangular grid

In the composite box of Figure 14.20, the space comprises 9 frames, organized in a 3 x 3 grid.

Elementary **`askData` Box A** uses three horizontal frames while **`askText` Box B** is two frame high. Generally, an elementary box is

– **`x`** frame wide, and

– **`y`** frame high.

This (**`x`**,**`y`**) specification of a variable size box should remind us another case, in which we were allowed to specify, through the **`/x`** and **`/y`** parameters the size of the text area or a picture area. This is just the same here: for each elementary box, we can specify, with parameter **`/x`** the number of horizontal frames and with **`/y`** the number of vertical frames it will span. Values **`/x1`** and **`/y1`** are default and need not be specified.

Figure 14.21 shows how the space is distributed among the six elementary boxes while Script 14.17 defines the composite box.

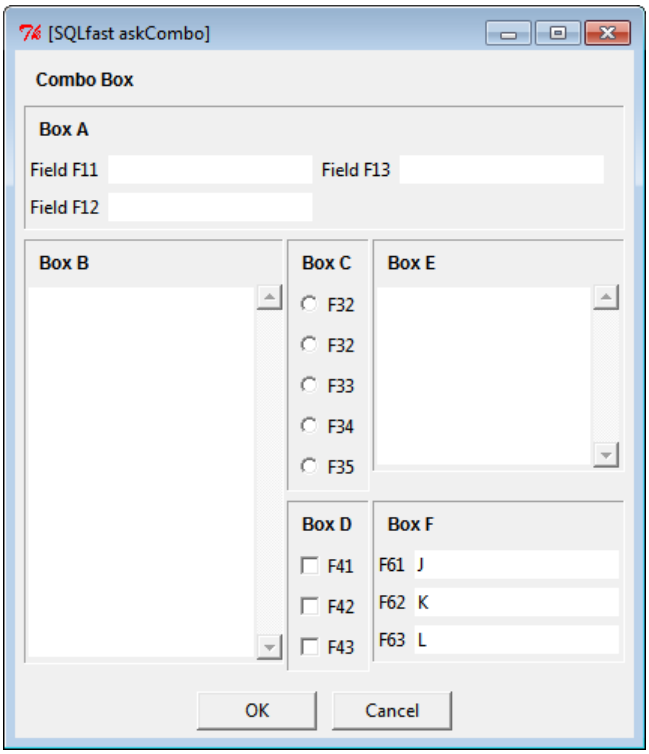



**Figure 14.20 -** A composite box in which elementary boxes span several frames

```
set bA = ask A,B,C = [/bBox A] Field F11|Field F12||Field F13;
set bB = askText D = [/b/x18/y15Box B];
set bC = selectOne E = [/bBox C] F32|F32|F33|F34|F35;
set bD = selectMany F,G,H = [/bBox D] F41|F42|F43;
set bE = askText I = [/b/x17/y7Box E];
set bF = askData J,K,L = [/bBox F] F61|F62|F63;
askCombo [/bCombo Box] [/x3$bA$]|[/y2$bB$]
                       ||[$bC$]|[$bD$]||[$bE$]|[$bF$];
```

**Script 14.17 -** Building the composite box of Figure 14.20 [Chapter-14-13.sql]

Rules of box placement among the frames (Figure 14.21):

– the space is structured as a grid of elementary frames, horizontally and vertically aligned,

– a frame contains only one elementary box or a part thereof,

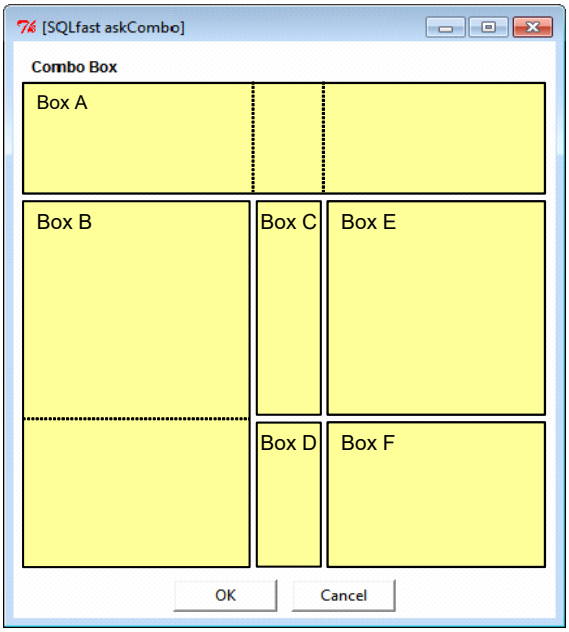



**Figure 14.21 -** Geometry of the composite box of Figure 14.20

– the height of the frames of a row is the size of the tallest elementary box in this row; the width of the frames of a column is the size of the widest elementary box in this column,
– when a simple separator | is encountered, the next box is placed in the next frame of the same (left most) column,
– when a double separator | | is encountered, the next box is placed in the highest available frame of the next column,
– when a triple separator | | | is encountered, the next box is placed on top of the left-most column. The code of Script 14.18 and Figure 14.22 illustrate the user of this new separator.

Defining a good layout of heterogeneous composite boxes may prove a bit tricky when the components have very different sizes. Surely, solid skill in Tetris may be an asset!

```
set bA = ask a,b,c = [/bBox bA] A||B||C;
set bB = askText text1 = [/b/y6/x32Box bB];
set bC = askText text2 = [/b/b/y6/x32Box bC];
set bD = selectOne s = [/bBox bD] V||W||X||Y||Z;
set bE = askText text3 = [/b/y6/x32Box bE];
set bF = askText text4 = [/b/b/y6/x32Box bF];

askCombo [/bComposite box]
         [/x2$bA$]|[$bB$]||[$bC$]|||[/x2$bD$]|[$bE$]||[$bF$];
```

**Script 14.18 -** Using the ||| separator to place the bD box

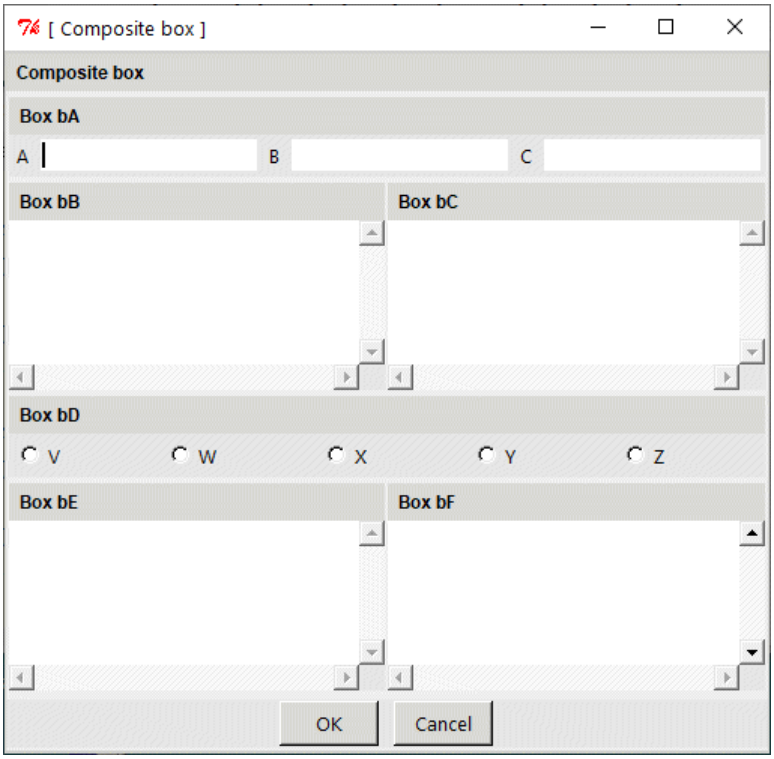



**Figure 14.22 -** Another complex composite box using the ||| separator

## 14.12 Of styles and colors

Some of the visual aspects of dialog boxes can be customized: *style*, *color* and *field length*.

– The **style** property specifies whether the component appears flat in its environment (value `flat`) or outlined, to give it some relief (value `relief`).
– The **color** property specifies its color as a string of six hexadecimal digits (two for each basic color).
– The **entryLength** property specifies the length, in characters, of data entry fields.

Style and color properties can be set for the following components:

– the main frame of a composite box,
– the general message of a composite box,
– the frame of an elementary boxes,
– the message of an elementary box,
– a text field,
– an entry field,
– a check button (`selectMany`),
– a radio button (`selectOne`),
– control buttons (OK, Cancel, etc.).

These properties are parameters that receive default values at start time through the SQLfast.ini initialization file:

```
styleBoxMessage = relief
```

In addition, their values can be dynamically modified through the `parameter` statement:

```
parameter styleBoxMessage = relief;
```

Figure 14.23 shows the same composite box displayed in three different styles.

## 14.13 Example of complex composite box

Figure 14.24 shows the main control panel of database ORDERS.db as it has been generated by the prototype generator developed for Chapter 19 (*Conceptual modelling*) of the reference book (in French). This generator is available in folder Scripts/BD-2022/Chapter-19 of the standard distribution.

## 14.14 Summary

SQLfast offers various means to improve the layout of dialog boxes.

• The components of a box (entry fields and buttons) can be arranged in **several columns**, thanks to the column break symbol | | and left align symbol | | |.

- **Titles** can be inserted among the list of components (entry fields and buttons) of a box. They are enclosed between curly brackets: **`{title}`**.

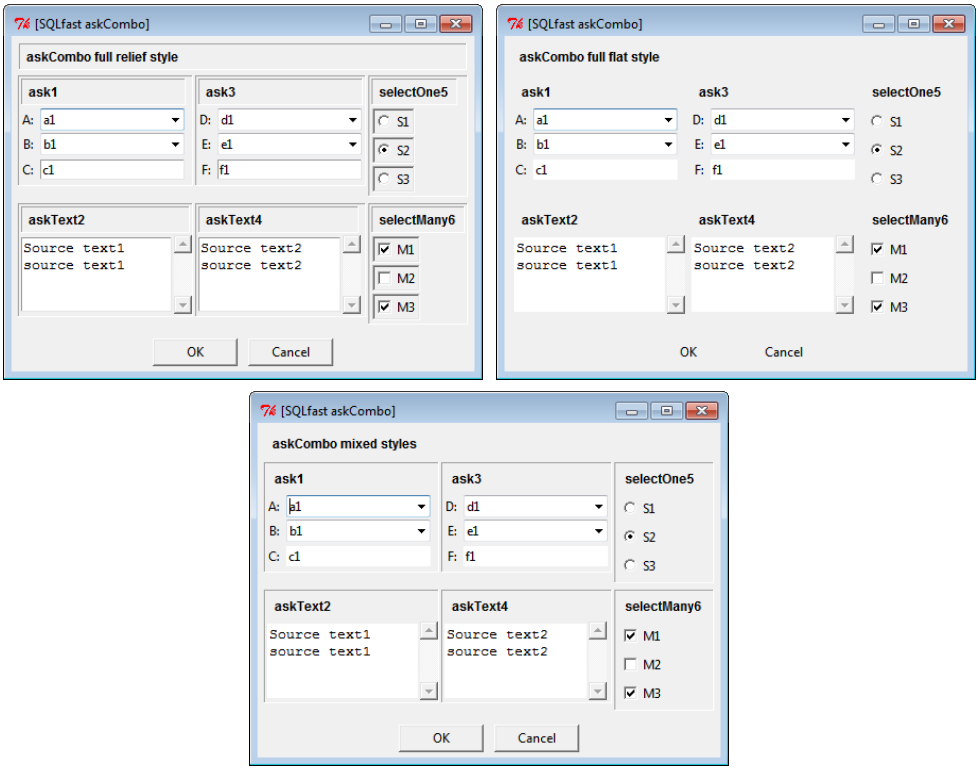



**Figure 14.23 -** Three typical dialogue styles [Chapter-14-14.sql]

- Labels of some components can be **hidden**, notably to place them in arrays.
- A list of **predefined values** can be associated with an entry field (in an **`ask`** box). The user is invited to select a value from a drop down list or to type another value in the entry field. The specification of the value list appears between square brackets ( **`[]`** ) after the field label
- A list of predefined values can be defined in two ways, as a **list of constants** or as the result of an **SQL select query**.
- The value returned generally is the value that appears in the entry field, be it selected from the predefined value list or typed by the user. When the list is defined by an SQL query, then the box can return an associated, but different, value in the target variable. In such a case, the select list of the query specifies *two expressions*. The first one is the value that will appear in the drop down list (the

**displayed value**) and the second one is the value actually sent in the target variable (the **returned value**).

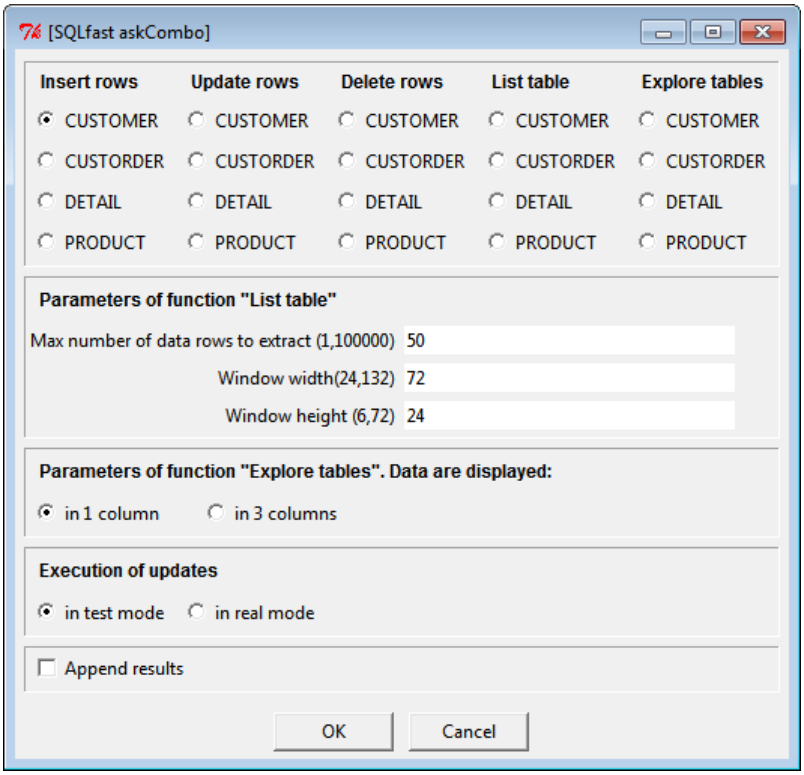



**Figure 14.24 -** The control panel of the database prototyping application

- When a list of predefined values is associated to an entry field, the user can be prevented to type his own value by making the field **read-only**. This is specified by control character `!` in the value list specification.
- The names of the standard buttons of dialog boxes are stored in system variables `buttonYES` and `buttonNO`. They can be modified. If the name of a button is an empty string, this button will not appear.
- A **composite dialog box** is made up of one or several standard dialog boxes (all kinds except **`showMessage`**). It is created through **`askCombo`** statement that specifies the sequence of the statements that create the elementary boxes, each enclosed between square brackets `[]`. A message field and column breaks can be specified but titles are not allowed though.

- The **style** and **color** of the components of dialog boxes can be customized. In addition, the standard **length** of data entry fields can be specified.
- In a **heterogeneous composite dialog box**, the elementary boxes are placed within a grid of *frames*. They can span one or several frames, so that both small and large boxes can be combined. The size of a box is defined by its width and height as a number of frames. The default value is *one frame* in both dimensions.

# Chapter 15

# Large data objects

**Objective**: We study the specific characteristics of large data objects, such as long character strings, document files and binary files. We examine how to display, store, transform, transmit and extract such objects.

**Keyword**: binary data, large data, character coding, ASCII, Unicode, UTF-8, hexadecimal coding, base64 coding, secure hashing, encryption, cracking, inference, bulk loading

**Statements and functions**: *file for loop*, openTextFile, nextTextLine, closeTextFile, copyFile, onError, insertLOB, variableToFile, fileToVariable, BfileToVariable, HEXencodeVarToVar, HEXdecodeVarToVar, b64encode VarToVar, b64decodeVarToVar, hex2char(), char2hex(), b64encode(), b64decode(), hash(), hashFile, encrypt, decrypt, execProg, execSQL, fastExec

**System variables and parameters**: FILEdiag, incoding_file, outcoding_ file

## 15.1 Beyond elementary data items

In all the data manipulations we have performed so far, we have collected, read, written, computed and compared *small data items*: numbers, names, dates and sometimes short sentences. The world of information is also populated with larger data pieces, like documents, sounds, images, videos and computer programs. By contrast,

we generally call them *large objects*. We will study how to manipulate large objects in SQLfast scripts.

## 15.2 Displaying large character strings

Column values that appear in the result sets written on the output channel or in *Show Data* windows may be so large that the resulting display is practically unreadable. On a normal monitor screen, columns of more than 50 cm would hardly be tolerated!

Let us suppose, for the sake of the demonstration, that table PRODUCT comprise additional text column HISTORY, declared `varchar(2048)`. Clearly some of its value are likely to be too long to appear on the screen.

There are several ways to limit the size of the values of these columns, to, say, 60 characters.

– The more straightforward technique consists in *explicitly truncating* the values of HISTORY in the `select` query:

```
select PridID,substr(HISTORY,1,60) from PRODUCT;
```

– Column values can be automatically limited through system variables maxSelectWidth for the results sets on the *output channel* and maxShowDataWidth in *Show data* windows. However, these settings are global to all columns of all tables. They can be specified in file SQLfast.ini:

```
maxSelectWidth = 60
maxShowDataWidth = 60
```

or dynamically, in scripts:

```
set maxSelectWidth = 60;
set maxShowDataWidth = 60;
```

– In some cases, the values are too long because they include long strings of identical characters that are considered meaningless. For example, short texts from external sources of poor quality includes sequences of dashes to simulate underlining and spaces to indent lines. By condensing these meaningless sequences, such column values may get a more modest length. SQLfast provides two reducing functions:

- **`reduce(s,c)`**: returns **`s`** in which all the sequences of each character in **`c`** have been reduced to one instance. For example, **`reduce`**`(HISTORY,' -')` condenses all sequences of spaces into one space and all sequences of `'-'` into a single dash.
- **`reduceAll(s)`**: same as **`reduce`** in which **`c`** includes all characters.

## 15.3 Reading and writing (large) text files

In this first experiment, we store a multiline text in external file SampleFile.txt, then we read it back in an SQLfast variable (Script 15.1).

Statement [1] stores the sample text in variable **txt1**. For the sake of the demonstration, and with some imagination, we will consider it a *large text*.

Statement **variableToFile** [2] has already been used in Section 13.4, to store a picture in a file. Here, its function is to store the value of variable **txt1** in file **SampleFile.txt** in system folder **Output**. If this file does not exist, it is created. If it already exists, its former contents is lost.

```
set txt1 = This is a simple text\n comprising\n three lines; [1]
variableToFile txt1 = Output/SampleFile.txt;                 [2]
fileToVariable txt2 = Output/SampleFile.txt;                 [3]
write $txt2$;                                                [4]
```

**Script 15.1 -** Writing and reading an (allegedly) *large* text file [extended version in Chapter-15-01.sql]

Statement **fileToVariable** [3] has the inverse effect: it reads the contents of file **SampleFile.txt** and stores it in variable **txt2**. Statement [4] just checks that we have lost nothing in this process.

When the text file is made up of a *sequence of lines*, that is, character strings terminated, or separated, by a *new line* symbol, a variant of statement **fileToVariable**, called *indexed* **fileTovariable**, can select the line at a specific position in the sequence:

```
fileToVariable lin = SampleFile.txt #2;
```

As we shown it in Section 9.4, the file can be read line by line, by a ***file*** *for loop*. For instance, the following statement writes non empty lines on the output channel:

```
for lin = [file SampleFile.txt]
          if ('$lin$' <> '') write $lin$;
```

This loop can be simulated with an indexed **fileTovariable**, though this form is more complex and much slower than the former expression:

In some cases, it could be appropriate to develop the *file for loop* by using file access primitives (Script 15.3):

```
for iL = [1,999999];
   fileToVariable lin = SampleFile.txt #$iL$;
   if ('$FILEdiag$' = 'NONE') exit;
   if ('$lin$' <> '') write $lin$;
endfor;
```

**Script 15.2 -** Reading a text file, one line at a time

- `openTextFile` *filename*: open a text file
- `nextTextLine` var = *filename*: get next line in an opened text file
- `closeTextFile` *filename*: close an opened text file.

Though the script appears to be more complex than the pure *file for loop*,[1] it allows more flexible algorithms to be written, such as comparing two files line by line, for example to detect their difference.

```
openTextFile SampleFile.txt;
while (True);
  nextTextLine lin = SampleFile.txt;
  if ('$FILEdiag$' <> 'OK') exit;
  write $lin$;
endwhile;
closeTextFile SampleFile.txt;
```

**Script 15.3 -** Reading a text file by file access primitives

Copying a file can be carried out by loading the file contents in a variable (**`fileToVariable`**) then by storing the contents of the variable to a new file (**`variableToFile`**). Nevertheless, the simplest way to copy a file, whatever its type, is through specific statement **copyFile**:

```
copyFile ORDERS.db, ORDERS.db.bak;
```

These scripts and some others comprising text file reading are available in file Chapter-15-02.sql.

## 15.4 Reading and writing large binary files

Reading and writing large objects through a variable is not limited to *text* files.

1. Anyway, these two forms have exactly the same execution time, which is proportional to the size of the file.

Script 15.4 shows that exchanging a picture in jpeg format with an external file can be done via the same statements as for text files.

In this script, we decide to better control the existence of the source file. First, we tell SQLfast that we will take in charge the management of script errors. For this, we disable the standard error management of SQfast, which stops script execution with an error message:

```
onError continue
```

This setting tells SQLfast that, from now on, when an error occurs, the execution must continue.

Then, when the file has (tentatively) been loaded in variable **var**, we check whether this operation succeded:

```
if ('$FILEdiag$' = 'OK')
```

Finally, we restore the default error management:

```
onError stop;
```

```
onError continue;
BfileToVariable var = Screen.jpg;
if ('$FILEdiag$' = 'OK') variableToFile var = copy_Screen.jpg;
onError stop;
```

**Script 15.4 -** Reading and storing an (image) binary file [Chapter-15-03.sql]

## 15.5 Storing large objects in a database

Large objects, be they text or binary, can also be stored in a database and retrieved later on. Let us first create a database in which we can store arbitrary documents of any type (Script 15.5). Column **Contents**, declared of type **blob**, for ***B**inary **L**arge **Ob**ject*, can accommodate text or binary data of any type and any size.

```
createOrReplaceDB DOC.db;
create table DOCUMENT(
       ID       integer not null primary key,
       Name     varchar(128),
       Content  blob);
```

**Script 15.5 -** Creating a document database

Storing a large piece of data cannot be done through a mere **insert** statement. Indeed, storing and reading large objects require special procedures in relational DBMS. We can understand that storing a 500 MB video clip and retrieving it could be a bit more tricky than merely manipulating numbers and names.

To store a large object in a database, SQLfast offers a special statement, **insertLOB**, that hides the complexity of the underlying DBMS. It goes as follows:

```
insertLOB lob = Contents from DOCUMENT where ID = 1;
```

Variable **lob** initially contains the large object we intend to store in the database. "Content **from** DOCUMENT **where** ID = 1" is an SQL fragment that speaks for itself: it specifies where to store this large object, that is, in column Content of the row of table DOCUMENT with value of ID = 1.

Script 15.6 shows how the pdf document of file SQLfast-Tutorial.pdf is stored in the database. It first stores the content of the file in variable **lob** [1]. Since the file may contain binary data (a **pdf** file *definitely* is binary), we use the variant **BfileToVariable**.[2] Then, it creates the host row that specifies the *id* and the *name* of the object [2]. Finally, through statement **insertLOB**, it stores the document in this row [3].

```
BfileToVariable lob = Files/Tutorial.pdf;                    [1]
insert into DOCUMENT(ID,Name) values (1,'Tutorial.pdf');     [2]
insertLOB lob = Content from DOCUMENT where ID = 1;          [3]
```

**Script 15.6 -** Storing a pdf document in a database

Extracting this document and transforming it into a pdf file is quite straightforward (Script 15.7). The name and content of the document are loaded in variables **name** and **lob**. The value of **lob** is then saved as a file with name $name$ in directory ExtractDir.

```
extract name,lob = select Name,Content
                   from   DOCUMENT where ID = 1;
variableToFile lob = ExtractDir/Tutorial.pdf;
```

**Script 15.7 -** Extracting a pdf document from the database

An extended version of these scripts is given in Script 15.8.

2. Standard **fileToVariable** statement tries to decode the contents of external file according to the coding scheme of the platform (Windows, Mac OS, Linux) as specified by parameter **incoding_file** in initialization file **SQLfast.ini**. This decoding is likely to fail for non text contents, leaving the value of target variable inconsistent. **BfileToVariable** just loads the file contents in the variable, without additional processing.

### Note

Unless the large object is known for sure to be a pure text, it is not recommended to **write** it on the output channel! Indeed, SQLfast has so far no drivers to display and print pdf or jpg documents.

## 15.6 About binary values

We have used the terms *binary* value and *binary file* all along this chapter. It is time to examine in more detail what they mean and what they imply.

```
openDB DOC.db;
set source = Tutorial.pdf;
set target = Copy_of_$source$;
onError continue;
fileToVariable lob = $source$;
if ('$FILEdiag$' = 'OK');
  insert into DOCUMENT(ID,Name) values (1,'$source$');
  insertLOB lob = Content from DOCUMENT where ID = 1;
  extract name,lob = select Name,Content1
                     from   DOCUMENT where ID = 1;
  variableToFile lob = $target$;
else:
  showMessage Source file $source$ not found;
endif;
onError stop;
commitDB;
closeDB;
```

**Script 15.8 -** Storing and extracting a pdf document from the database [Chapter-15-04.sql]

Let us first leave aside elementary values such as those known by relational DBMS: *integer*, *real* (or *float*), *character*, *varying character*, *boolean*, *date* and the like. When they come from a database, they can be extracted, processed and displayed without problem by the database interface, notably by SQLfast.

Things may become more tricky when the data come from files. A file, whatever its type, may be considered as an *uninterpreted* sequence of bytes, each one representing a number from 0 to 255. Most file can also be interpreted as something significant: a text, a list of comma-separated values, a program, an image, a piece of music.

### 15.6.1 Text files: from cp850 to UTF-8

As we have seen in the previous sections, unformatted text files seem to be particularly simple to read and to interpret: the file contents is a sequence of lines, each line being made up of a sequence of characters. If we examine such a file with a simple text editor (*Notepad* or *TextEdit* for instance), this is what we observe.

#### End of line coding

However, if we open a text file with a more technical editor, such *Notepad++*, we observe that lines are terminated by special characters, generally called *end of line*, or *eol* for short.

If we push the examination a bit further, we notice that this *eol* characters comprise one or two bytes, and that these bytes vary depending on the platform in which the file was created: byte 13 (called *carriage return*) for an old Mac machine, byte 10 (called *line feed*) on Linux and OS X machines and two bytes 13 and 10 for a Windows machine or standard HTML code. Though these *characters* have a direct effect on the rendering of the text, they are qualified *non-printable* since they have no proper typographic representation.

How does SQLfast cope with this variety? Let us say that it does its best to identify and normalize eol symbols in order to process, to display and to store text with the appropriate eol coding. If the result is not satisfying, then the text value can be modified through statement **`replace`**.

#### The desolation of ASCII

Everything works fine as long as the characters of the text are drawn from the upper and lower case US character set (A-Z + a-z), the decimal digits (0-9) and some common symbols such as *dot*, *comma*, *colon*, *arithmetic* operators and some others. As a whole, these characters form the so-called ASCII character set (the first one of which is a space):

```
  !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRS
TUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

They are coded as numbers from 32 (space) to 126 (tilde). Bytes below 32 represent operations, such as *carriage return* and *line feed*, but also several others which we are not interested in. It is interesting to notice that this coding requires 7 bits, leaving the higher order bit unused. The ASCII character set is sufficient to represent simple English texts but falls short for *foreign languages* (from the US point of view!). For instance, European alphabets include many additional characters, such as accented letters.

Coding these additional characters seems quite easy: by using the 8th, unused, bit of the byte, we are provided with 128 more codes, from 128 to 255. For example, letter ç is coded as 135, ¿ as 168 and α as 224.

Unfortunately, while the 128 first codes are common to all coding schemes, dozens of different mappings for the 128 next codes have been designed and imple-

mented. Not only they represent different character sets, but the same character may be represented by different codes. Among these schemes, ISO-8859-1 (also called Latin-1), cp850 (DOS), cp1252 (Windows, close to Latin-1 but not quite equivalent) and MacRoman (Mac OS until version 9) are among the most popular.

This has a strong implication for SQLfast scripts: one cannot read a text file if we do not know with which coding scheme it was written. Specifying this scheme is the role of system parameter `incoding_file`, which is set in initialization file SQLfast.ini:

```
incoding_file = cp1252
```

The second most important drawback of these codings, based on the ratio *one byte per character*, is that they are unable to represent non western character sets such as Arabic, Chinese or Indian, which comprise many more symbols.

Finally, since the coding schema is a property of the file, one cannot mix several characters sets in the same text.

Clearly, the 8-bit character codings are obsolete and must be replaced by better schemes. Unfortunately, they still are very popular, so that they cannot be ignored.

### Here comes Unicode

The *Unicode* coding schema is the universal solution to address all these problems. The idea is simple: we assign a non-negative integral number (called *code point*) to each character of all the alphabets, be they past, current, future or even fictitious.[3] This way, all the symbols are represented in a unique numbering system, that is, the same alphabet. An interesting property of Unicode is that the code points 0-127 are the ASCII codes, so that any text using the ASCII character code only is automatically coded in Unicode.

Unicode, which is based on a (theoretically) infinite set of numbers[4], is an *abstract coding*. To translate a text into a byte string, we must define the way the code points are represented. Since some of these numbers can be very large, their representation must used several bytes. The way Unicode code points are coded as sequences of bytes is called *Unicode encoding*. As we can expect, there are several of such encoding.

The simplest of them is UTF-32. It assigns a 32-bit integer (4 bytes are more than enough to represent all of them) to each code points, so that a text of N characters requires **4xN** bytes.

UTF-16 assigns two bytes to most code points (four bytes for the others), so that a text of N characters requires from **2xN** to **4xN** bytes.

The most popular encoding is UTF-8, which produce a variable length encoding, more efficient than UTF-16. Each code point is encoded into one, two, three or four

3. For example, the **Klingon** alphabet has been assigned Unicode code points.
4. and practically 1,114,112.

bytes. The 127 ASCII characters are encoded in one byte, so that an ASCII text is also a UTF-8 Unicode encoded text. The other characters require from 2 to 4 bytes according to somewhat tricky rules that are not worth being detailed here.[5]

If a script intends to read external text files encoded in UTF-8, the following setting must be specified:

```
incoding_file = utf8
```

Similarly, to create a UTF-8 Unicode encoded text file, one must include, in SQLfast.ini parameter file:

```
outcoding_file = utf8
```

Though Unicode encodings definitely solve the multi-alphabet problem, they have a major drawback: processing strings of variable-length characters is more complex than coping with strings of fixed-length character. For instance, computing the length of a string (that is, the number of its characters) is no longer related to the number of its bytes (one exception: `utf32`). The interpretation of a byte requires the interpretation of some of the bytes that precede.

### 15.6.2 Binary files

We will call *binary* any file the content of which cannot be described as a sequence of character representations, as we have studied in the previous section. This definition encompasses many file types such as formatted texts (e.g., *.doc, *.odt, *.docx document files) or picture files (such as *.bmp, *.jpg, *.gif files).

When examined with a text editor, their contents appears *gibberish*, full of strange symbols that suggest nothing coherent. For this, this contents is qualified *not printable*. Trying to load it in a variable, then writing it on the output channel will produce nonsense stuff, at best, and will perhaps crash the script.

Clearly, such files must be coped with in specific ways. For instance, their contents must be loaded through statement **`BfileToVariable`** and not with **`fileToVariable`**.

When a variable has been loaded with the content of a binary file, it gets the *binary* status. From now on, its use is limited, to avoid risky operations. For instance, it will be ignored when applying *variable/value* substitution.

### 15.6.3 Displaying extracted binary data

Standard operations provided by the graphical interface are unable to display non binary data extracted from a database. For example, let us consider the **DOC.db** data-

5. They are described in function `utf8Compliant()` of UDF library SQLfastUDFlib.py available in the standard distribution.

base, in which table DOCUMENT contains copies of binary documents. Trying to display its contents on the output channel with query

```
select * from DOCUMENT;
```

will display the following result.

```
+----+--------------+-----------------+
| ID | Name         | Content         |
+----+--------------+-----------------+
| 1  | Tutorial.pdf | <not printable> |
+----+--------------+-----------------+
```

Trying to examine the contents of this table through the Show Data function (menu Databases > Show DB data) fails with an explicit message (Figure 15.1).

The best way to avoid this is to define a view in which binary values are treated in an appropriate way, as shown in the example of script 15.9. In the Show DB data function, choosing view DOCUMENT_V instead of DOCUMENT will avoid such failure.

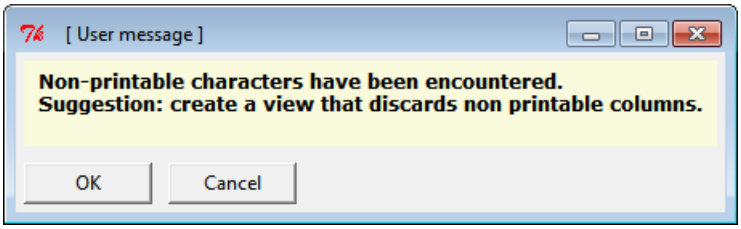



**Figure 15.1 -** Trying to examine binary data through the *Show data* function

```
create view DOCUMENT_V as
select ID,Name,
       case when Content is not null
            then '<not printable>'
            else null
       end as Content
from DOCUMENT;
select * from DOCUMENT_V;
```

**Script 15.9 -** Coping with non printable values [Chapter-15-05.sql]

This does not mean that SQLfast cannot display a binary file. If this file contains a **gif** or **jpeg** image, it can be shown through a **`showPicture`** dialogue box. For all kinds of binary files, it is best to make them processed by the appropriate software through statement **`execProg`**:

```
execProg $PDFprocessor$ Files/Tutorial.pdf;
```

In this example, system variable **PDFprocessor** contains the name of our preferred **pdf** viewer. This variable is initialized in file SQLfast.ini:

```
PDFprocessor = "C:/Program Files/Foxit Software/Foxit Reader.exe"
```

## 15.7 Binary data encoding/decoding

The non-ASCII characters, and particularly the *non-printable* bytes, in text and binary files may lead to storage, extraction and transmission problems. A common way to get rid of these problems is to encode the file contents in a reduce character set in such a way that all bytes are printable. We examine two common encoding techniques, *hexadecimal* and *base64*.

### 15.7.1 Hexadecimal encoding/decoding

*Hexadecimal* encoding transforms an arbitrary byte string (containing for example accented characters, UTF-8, MacRoman or cp1252 codes, images, movies or binary code of programs) into a string of characters from the reduced 16-symbol alphabet `0123456789abcdef`. This source byte string can be stored in a variable or in a file. The resulting character string can be stored in a variable or in a text file as well.

The *hexadecimal* encoding converts each half-byte (4 bits) of the source byte string into one of the sixteen characters of the hexadecimal alphabet. A source of **N** arbitrary bytes is transformed into a string of **2xN** printable characters. There is a size increase of 100%, which may be significant for large values.

Script 15.10 applies *variable to variable* conversions to a short sentence.

```
set address = Rue de l'été;
HEXencodeVarToVar hexAddress = address;
HEXdecodeVarToVar newAddress = hexAddress;
write $address$ --> $hexAddress$ --> $newAddress$;
```

**Script 15.10 -** Hexadecimal coding/decoding

It writes this line in the output window:

```
Rue de l'été --> 527565206465206c27e974e9 --> Rue de l'été
```

Large binary byte strings, such as images, generally are stored in files, so that *file to file*, *file to variable* or *variable to file* conversions may prove more convenient:

```
HEXencodeFileToVar hexImage = cat005.jpg;
variableToFile hexImage = cat005.txt;
```

or, more concisely,

```
HEXencodeFileToFile cat005.txt = cat005.jpg;
```

then, later,

```
HEXdecodeFileToFile pet1032.jpg = cat005.txt;
```

Light versions of HEX conversion statements are available as UDF functions: **char2hex** and **hex2char** (Script 15.11). Their usage is limited to character string conversion.

```
set address = Rue de l'été;
compute hexAddress = char2hex('§address§');
compute newAddress = hex2char('$hexAddress$');
write $address$ --> $hexAddress$ --> $newAddress$;
```

**Script 15.11 -** Hexadecimal coding/decoding through UDF functions

An extended version of hexadecimal scripts is given in file Chapter-15-06.sql.

### 15.7.2 base64 encoding/decoding

***base64*** encoding follows the same principles as *hexadecimal* encoding, but relies on a richer reduced alphabet made up of uppercase letters `A-Z`, lowercase letters `a-z`, digits `0-9` + additional characters '+' and '/', totalling 64 distinct characters (hence the name).

Each sequence of **three bytes** (8-bit entities) of the source is interpreted as a sequence of **four sextets** (6-bit entities), each denoting one of the 64 characters of the reduced alphabet. So, a source of **N** arbitrary bytes is transformed into a string of about **4/3 x N** printable characters, which results into a size increase of about 33%.[6]

In the *url-safe* variant of this reduced alphabet, the two additional characters are replaced by '-' and '_'. This alternative alphabet generally is used to avoid character strings that could erroneously be confused with file paths and url's, in which characters '+' and '/' have a special meaning.

Script 15.12 applies *variable to variable* conversions.

6. Why *about*? Because the resulting base64 code generally is right padded with `'='` characters to make its length a multiple of 4.

```
set address = Rue de l'été;
b64encodeVarToVar b64Address = address;
b64decodeVarToVar newAddress = b64Address;
write $address$ --> $b64Address$ --> $newAddress$;
```

**Script 15.12 -** base64 coding/decoding through UDF statements

It writes this line in the output window:

```
Rue de l'été --> UnVlIGRlIGwn6XTp --> Rue de l'été
```

Other expressions:

```
b64encodeFileToVar b64image = cat005.jpg;
variableToFile b64Image = cat005.txt;
```

or, more concisely,

```
b64encodeFileToFile cat005.txt = cat005.jpg;
```

then, later,

```
b64decodeFileToFile pet1032.jpg = cat005.txt;
```

To use *url-safe* reduced alphabet, just add suffix '**-u**' to encoding forms:

```
b64encodeFileToFile-u cat005.txt = cat005.jpg;
```

UDF functions also exist for base64 conversion statements: **b64encode** and **b64decode** (Script 15.13). Their usage is also limited to character string conversion.

```
set address = Rue de l'été;
compute b64Address = b64encode('§address§','std');
compute newAddress = b64decode('$b64Address$');
write $address$ --> $b64Address$ --> $newAddress$;
```

**Script 15.13 -** base64 coding/decoding through UDF functions

An extended version of base64 scripts is given in file Chapter-15-07.sql.

### 15.7.3 Secure hashing

The *hexadecimal* and *base64* conversions are reversible: the source data can be recovered from the encoded version through the decode function.

The SQLfast **hash** function is of a different nature. It generates a string of 64 hexadecimal characters (called *hash value*) computed from any byte string, character or binary, whatever its length. This transformation is not reversible: there is no (realistic) way to recover the source byte string from its hash value.

An important property of hash functions is that two different source strings don't generate the same hash value, such an unfortunate event being called a *collision*.

Of course this *collision-free* property, mathematically speaking, cannot be guaranteed, since general-purpose hash functions are not *injective*. In practice, good hash functions minimize the risk of collision to such a way that they can be considered injective, practically speaking (see *Technical note*, below).

This function can be used in many ways to protect data against accidental or intentional corruption, tampering or privacy breach. Let us describe two of the most common usages.

#### Securely storing passwords

Storing user passwords in a table for further lookup is a very bad idea, for obvious reason! It is best to store their hash values. To check whether an input password is valid, it suffices to check whether its hash value is in the table (Script 15.14).

```
ask pw = Password;
extract authorization = select count(1) from PASSWORD
                        where  Value = hash('$pw$');
if ($authorization$);
   write Access permitted;
else;
   write Access denied;
endif;
```

**Script 15.14 -** Checking passwords

#### Software certification

When we download a software or a software update as executable binary file, it is important to be sure that we get the original version uncorrupted (or not tampered by hackers). To this aim, the software producer also provides the hash value of the binary file, so that the recipients can check that the downloaded file generates the same hash value.

In Script 15.15, the hash value of a selected SQLfast distribution **zip** file is computed through statement **hashFile** and stored in table DISTRIBUTION.

```
selectFile file = D:/SQLfast/Distr, Zip files, zip;
hashFile hash = $file$;
insert into DISTRIBUTION values ('$file$','$hash$');
```

**Script 15.15 -** Storing in a database the hash value of an SQLfast zip file

### Technical notes

The SQLfast **hash** function and statement are based on SHA-256, one of the best *secure hashing algorithm* (SHA) The term *secure* refers to the robustness of the algorithm against *collisions*. A collision occurs when two different source strings generate the same hash value.
A *secure hash algorithm* makes finding a string `S1` that produces the same hash value as a given source string `S0` too expensive to be realistic. For example, finding a password that hashes into the same value as an unknown password is practically impossible, at least with the current state of the technology.

Examples of hashing scripts are given in file Chapter-15-08.sql.

## 15.8 A touch of cryptography

*Encryption* is a data protection transformation technique that converts source data into a form that is unreadable, except for authorized persons. Among the numerous encryption techniques, SQLfast has chosen a variant of the *Vigenère algorithm*, which is both simple and considered reasonably difficult to crack. This technique is symmetrical: the source data are *encoded* through a secret key and encrypted data are *decoded* with the same key. Two encryption functions are available:

- `encrypt(S,K)`: encodes source data **`S`** with key **`K`**, then, transforms the result into base64 code. null and empty strings are encrypted as empty strings.
- `decrypt(C,K)`: decrypt base64 encoded encrypted data **`C`** with key **`K`**.

For not null character string source `S`, these functions are inverse of each other:

```
decrypt(encrypt(S,K),K) = S
```

If `S` is not null and numeric, since `encrypt` and `decrypt` produce character strings, the equivalence becomes:

```
cast(decrypt(encrypt(S,K),K) as numeric) = S
```

In Script 15.16, the new value of column Name is *encrypted* with the key specified by the user.

```
ask cid,nam,key = CustID:|Name:|Encryption key:;
update CUSTOMER
  set  Name = encrypt('$nam$','$key$')
  where CustID = '$cid$';
```

**Script 15.16 -** Update data with encrypted values

To recover its original value, the data stored in the database must be *decrypted*:

```
select CustID,decrypt(Name,'$key$')
from   CUSTOMER where CustID = '$cid$';
```

The following list shows excerpts of the content of table CUSTOMER in which most columns have been encrypted.

```
+--------+--------------+--------------+------+----------+
| CustID | Name         | City         | Cat  | Account  |
+--------+--------------+--------------+------+----------+
| B112   | sq22vbG2uLE= | utvR3tXN3N8= | rZ0= | m56dmg== |
| C123   | t7G6rbWtvA== | uM3V394=     | rZ0= | l56bmpw= |
| B332   | t7u2vrU=     | sdHWUuLN     | rJ4= | mg==     |
| F010   | vru9vb-ps7q8 | utvR3tXN3N8= | rZ0= | mg==     |
| ....   | ...          | ...          | ...  | ...      |
| C400   | sLG6q76s     | utvR3tXN3N8= | rJ4= | naGY     |
| ....   | ...          | ...          | ...  | ...      |
| S712   | scGxtripv7mt | us3a098=     | rJ0= | mg==     |
| D063   | t7G6rbWtvA== | vtvd1tvd3dE= |      | l56an5w= |
| F400   | tK2rua4=     | rN7d4tHU1tHb | rZ4= | mg==     |
+--------+--------------+--------------+------+----------+
```

A careful examination of the encrypted data shows quite interesting patterns. For example, customers B112, F010 and C400 live in the same city, and this city has a shorter name than that of customer C123. Similarly, customers B112, C123 and F010 belong to the same category, while customer D063 has none (null value). Though guessing the actual values would be a bit more difficult (it would use data mining and statistical inference techniques), encrypting data in this way does not appear very secure.

Hence the idea to encrypt *whole rows* instead of *individual columns*. Script 15.17 shows how this can be done for table CUSTOMER. The values of each row are concatenated according to these rules:

- null values are transformed into empty strings
- numeric values are transformed into numeric character strings
- the values are separated by a symbol that cannot appear in values, here `'§'`, but `'|'` would be fine too In this way, elements can be further extracted through function `item`.

Then, the resulting string is encrypted and stored in column Name. The values of the other columns are wiped.

```
update CUSTOMER
set Name = encrypt(Name
                   ||'§'||Address
                   ||'§'||City
                   ||'§'||coalesce(Cat,'')
                   ||'§'||cast(Account as char),'$key$'),
           Address = '',
           City = '',
           Cat = '',
           Account = 0;
```

**Script 15.17 -** Encrypting whole rows of table CUSTOMER

Table CUSTOMER now looks like this:

```
+--------+-------------------------------------------------------+--.
| CustID | Name                                                  |...
+--------+-------------------------------------------------------+--.
| B112   | k62Rkuangb7a3-Pn1SGts8LMpsiztxvJ0OPZ29nr1CGooxuqk6-V |...
| C123   | k6-RkuangcbK39Vn1ezKGcG-s72ut8Ygr9vS5-YgpKsMn6askao= |...
| B332   | kquXnpTrj5qz1-nvxiGywcLNqiGs1-Jh198MtKYgkQ==          |...
| F010   | lqaF5KKZqOnJ19rr0OPJGcjIts24s73HtSG14d3tyt_X5Ru8kiGV |...
| K111   | krKVnpTrj5qr3uPryufU4Oggt7uztL3MtSGx2-DlxiGnoxuwk6o= |...
| ....   | ....                                                  |...
| F400   | mLKRktfhj5rJ55TG0O_R2-Igq7uowbYgo-za6tnlzd_YGberCKo= |...
+--------+-------------------------------------------------------+--.
```

It is clear that trying to crack encrypting data would now be more difficult.

Decrypting these data can be done statically (Script 15.18), on the fly (in SQL queries) or through an SQL view. Null values and numeric values must be recovered explicitely.

```
update CUSTOMER
set Name    = item(decrypt(Name,'$key$'),1,'§'),
    Address = item(decrypt(Name,'$key$'),2,'§'),
    City    = item(decrypt(Name,'$key$'),3,'§'),
    Cat     = case when item(decrypt(Name,'$key$'),4,'§') = ''
                   then null
                   else item(decrypt(Name,'$key$'),4,'§')
              end,
    Account = cast(item(decrypt(Name,'$key$'),5,'§') as decimal);
```

**Script 15.18 -** Decrypting the rows of table CUSTOMER

*Cracking methods* generally are based on guessing the key length and on assuming that this key may comprise real words, drawn from a dictionary. The Vigenère encryption method is reasonably secure for *long random keys* and *short source data*.

Most DBMS include built-in encryption functions that are more powerful than those we have examined above. They generally work at the *page* level instead of at the column or row levels, and use stronger encryption algorithms, such as the AES (Advanced Encryption Standard) currently recommended by the National Institute of Standards and Technology.

The scripts of file **Chapter-15-09.sql** illustrate various usage patterns of encryption functions.

## 15.9 Where to store large objects

Large objects generally come from individual files and take a lot of space in the database. Extracting and storing large objects in a database may be time consuming, as compared to elementary file manipulations. In addition, coping with a huge monolithic database may prove less flexible than managing a large number of independent files.

So, the question is, is it worth to store them in a database, or is it better to let them in their original form and to just store in the database (meta-)data describing them (name, location, owner, creation date, tags, and so on)?

DBMS ensure a quality of service that cannot be guaranteed by a simple file system, particularly when the file portfolio comprises thousands or even millions of objects. Data integrity, crash recovery, concurrent access management, indexing, and distributed data management are DBMS staple functions that cannot be provided by generic file systems.

So, except in extreme situations (e.g., a small 1,000 picture database or a multi-terabytes database), there is no clear and definitive answer. SQLfast can cope with both scenarios but the final decision about the database structure is up to the user.

The script of file **Chapter-15-10.sql** shows an example of creation, loading and exploitation of an image database in which the **gif** files are stored in an SQL table.

## 15.10 Loading large volumes of data

The last example of large objects management concerns the loading of a large number of rows in a table, what is called *bulk loading*. If the source data are available as a sequence of **`insert`** statements stored in an **.sql** file, the operation can be performed as shown in Script 15.19.

```
openDB ORDERS.db;
  execSQL CUSTOMER-load.sql;
  commitDB;
closeDB;
```

**Script 15.19 -** Usual way to load new rows in table CUSTOMER

However, SQLfast provides a simplified variant of **execSQL** for fast data modification, namely **fastExec**. It too executes the SQL queries stored in an **.sql** file , but the statements of the source file are not validated nor prepared (no variable/value substitution). Only SQL queries are allowed. If an error of any kind occurs, the operation stops. The script is not preloaded before execution as is the case for normal SQLfast scripts, so that the size of scripts is only limited by disk space.

The script file typically comprises DDL queries that create tables then a large number of **insert**, **update** and **delete** queries, plus blank lines and comments.

**fastSQL** has four parameters: the full name of the source file, the line number of the first query to execute, the line number of the last query to execute and the step, i.e., the number of lines after which a checkpoint is executed. At each checkpoint, a commitDB is executed if suffix **-c** is specified and a message comprising the current line number and the current execution time is written to the console if suffix **-t** is specified. Suffixes **-c** and **-t** can be combined in **-ct** or **-tc**. If an error occurs **before** a committing checkpoint has been successfully executed, all the operations executed **after** the last committing checkpoint are discarded.

Example:

```
fastExec-ct E:/CUSTOMER-load.sql,1,10000000,100000;
```

Script 15.20 shows a small procedure that selects a source file then executes the statements of this file.

```
selectFile source = Files,Loading files,sql;
if ('$source$' = '') stop;
ask from,to,step = [/bLoading data from file $source$]
                   from row|to row|checkpoint step;
if ('$DIALOGbutton$' = 'Cancel') stop;
openDB ORDERS.db;
  fastExec-ct $source$,$from$,$to$,$step$;
closeDB;
```

**Script 15.20 -** Fast loading new rows in table CUSTOMER [script file Chapter-15-10.sql]

To evaluate the speed of this operation, we load 100,000 new rows in table CUSTOMER, with a checkpoint every 2,000 rows. On a typical machine, we get the following results:

- **fastExec** (no trace, only final commit): 3.3 s., i.e., **30,300** inserts/s.
- **fastExec-t**:(tracing) 3.4 s.
- **fastExec-c** (commit at checkpoint): 3.6 s.
- **fastExec-ct** (tracing and commit at checkpoint): 3.8 s.
- **execSQL** (no trace, only final commit): 9.8 s., i.e., **10,200** inserts/s.

In this case, **fastExec** is about 3 times faster than **execSQL**.

## 15.11 Summary

SQLfast is able to read, store, retrieve and write large data objects such as texts, **jpg** images or **pdf** documents.

- The contents of a text file, whatever its size, is stored in a variable through statement **fileToVariable**. If the file contains binary byte string, variant **BfileToVariable** must be used. In this case, the target variable gets the status *binary*.
- If the large object is in a text file, either the whole text or a selected line of this text (denoted by index expression `#2`), can be stored in a variable.
- A variable containing a large object cannot be written on the output channel (except if this object is a text).
- The contents of a variable is stored in a file through statement **variableToFile**.
- Statement **onERROR** `continue` disables the standard error management of SQLfast. The standard behavior is restored through statement **onError** `stop`.
- System variable **FILEdiag** is used to check the status of a file operation: **OK** if it worked fine and **NONE** if the file has not been processed properly (it is missing for example).
- A large object can be stored in a database in a column of type **blob** (binary large object). To store such an object, a row must first be inserted (if it does not exist already) through a standard **insert** statement. Then, the object is inserted in this row through statement **insertLOB**.
- Binary large objects can be made printable by *hexadecimal* and *base64* conversion.
- Hexadecimal conversion can be performed by statements `HEXencode` and `HEXdecode` statements. These statements allow source and target containers to be variables or files: `HEXencodeFileToVar`, `HEXencodeFileToFile`, `HEXencodeVarToVar`, `HEXencodeVarToFile`. Same for `HEXdecode` statements.

- Hexadecimal conversion of text strings can be done through SQLfast and SQL functions `char2hex` and `hex2char` UDF functions.
- base64 conversion can be performed by statements `b64encode` and `b64decode` statements. These statements allow source and target containers to be variables or files: `b64encodeFileToVar`, `b64encodeFileToFile`, `b64encodeVarToVar`, `b64encodeVarToFile`. Same for `b64decode` statements.
- base64 conversion of text strings can be done through SQLfast and SQL functions `b64encode` and `b64decode` UDF functions.
- The secure **`hash`** function of a byte string can be generated either through statement `hashFile` or (for text byte strings) by SQLfast and SQL functions hash.
- Sensitive data can be encrypted (function **`encrypt`**) then decrypted (function **`decrypt`**) through a secret key.
- **`fastExec`** is a simplified variant of **`execSQL`**, intended to fast load large volume of data into tables.

# Chapter 16

# Writing data tables - Part 1

**Objective**: The way the `select` statement writes data tables on the output channel makes the result simple and readable We will learn how to change this layout and also to produce professional formats, such as CSV and Key-value (and much more in next chapter).

**Keyword**: display format, csv, key-value, data loading

**Statements and functions**: select, execProg, parameter, import

**System variables and parameters**: select-align, bar11, bar12, bar13, bar14, bar21, bar22, bar23, bar24, bar31, bar32, bar33, bar34, head1, head2, head3, head4, data1, data2, data3, data4,

## 16.1 An in-depth look at output tables

The SQL query of Script 2.1, that we recall below,

```
select CustID,Name from CUSTOMER where City = 'Poitiers';
```

writes the following tabular result on the output channel:

```
+--------+-----------+
| CustID | Name      |
+--------+-----------+
| B112   | HANSENNE  |
| F010   | TOUSSAINT |
| C400   | FERARD    |
+--------+-----------+
```

This seemingly simple structure deserves some further analysis.

- First, let us examine each column. We observe that its width is just what is needed, no more, no less, to write the longest character string that appears in it, be it a column value (such as TOUSSAINT) or a column name (such as CustID) + one leading space + one trailing space. Indeed, SQLfast computes the longest value of each column before effectively writing the table according to the most appropriate layout.
- Each column is made up of a head row, comprising column names, and zero, one or several data rows. We will call `head` the format of the head row and `data` the format of each of the data rows.
- The head row and data rows are displayed in a rectangular frame made up of symbols that form horizontal and vertical bars.
- There are three horizontal bars: one at the top of the table, the second one that separates the head row from the first data row, and the bottom bar. We call them `bar1`, `bar2` and `bar3` respectively. Each of them is formed with four special (strings of) characters: beginning of the bar (e.g, `'+-'`), horizontal part (a series of `'-'`), column separator (`'-+-'`) and the end of the bar (`'-+'`). We will call these symbols `bar11`, `bar12`, `bar13` and `bar14` for the top bar, and so on for the other two bars. The symbols with which the three bars are built are defined by 12 variables, from `bar11` to `bar34`.
- The `head` and `data` rows each include three symbols that form the vertical lines: beginning (e.g., `'| '`), separator (`' | '`) and end of row (`' |'`). We call them `head1`, `head3`, `head4` for the `head` row and `data1`, `data3` and `data4` for the data row. Finally, these rows include the value fields, between two consecutive separators. These fields are called, respectively, `head2` in head row and `data2` in data rows.

The following schema summarizes this standard layout:

```
1----2---3-----2-----4  bar1 = (bar11,bar12,bar13,bar14)
1 222222 3 2222      4  head = (head1,head2,head3,head4)
1----2---3-----2-----4  bar2 = (bar21,bar22,bar23,bar24)
1 2222   3 22222222  4  data = (data1,data2,data3,data4)
| F010   | TOUSSAINT |
| C400   | FERARD    |
1----2---3-----2-----4  bar3 = (bar31,bar32,bar33,bar34)
```

So, the layout of the tables on the output channel is defined by 20 variables (actually 22, as we will see later). These variables are built-in in the SQLfast engine and have been assigned, as initial values, the symbols used in the tables above, i.e.,

```
set bar11 = +-;        set bar21 = +-;        set bar31 = +-;
set bar12 = -;         set bar22 = -;         set bar32 = -;
set bar13 = -+-;       set bar23 = -+-;       set bar33 = -+-;
set bar14 = -+@n;      set bar24 = -+@n;      set bar34 = -+@n;
```

```
set head1 = | ;          set data1 = | ;
set head2 = @h;          set data2 = @v;
set head3 = @s| ;        set data3 = @s| ;
set head4 = @s|@n;       set data4 = @s|@n;
```

The role of some variables is easy to understand. For instance,

```
bar1 = bar11 + n1 x bar12 + bar13 + n2 x bar12 + bar13 + ··· + bar14
```

where `n1`, `n2`, etc. are the lengths of the successive columns. For the standard format, `bar1` translates into (for `n1=4` and `n2=8`):

```
bar1 = +------+----------+
```

However, special symbols also appear in some variables: **`@n`**, **`@s`**, **`@h`** and **`@v`**.These are called *control symbols*, since they ask SQLfast to *do something* when it writes a line on the output channel. For instance, control symbol @**n** (= ***n**ew line*) instructs SQLfast to write on the next line. The value of `bar14 = '-+@n'` tells SQLfast to write `'-'`, then `'+'`, then to *go to next line*. There are eight control symbols, that we will use in the following sections:

- `'@n'` represents a *new line* command; when encountered, writing continues on the next line (also called *next line*, *next paragraph*, *carriage return*, *line feed*, etc.)
- `'@S5'` represents 5 space characters; when encountered, is replaced by five space character.[1]
- `'@s'` represents a space character; when encountered, is replaced by one space character. Equivalent to `'@S1'`.
- `'@t'` represents a tabulation; when encountered, writing continues at the next tab stop.
- `'@h'` denotes the column name; when encountered, is replaced by the name of the current column (for instance `CustID` for the first column).
- `'@v'` denotes the current value; when encountered, is replaced by the current value of the current column (for instance `B112` for the first column of the first data row).
- `'@A'` designates system variable `A` which is incremented each time it is referenced.[2] The initial value and the increment are defined by system variables selectA0 and selectAi.
- `'@0'`, ... , `'@4'` designates five user variables; when encountered, `@i` is replaced by the value of user variable `selecti`. They will be discussed later on.

1. Also works for 4, and even for 9 and 17!
2. For instance, statement `write @A,@A,@A;` writes `0,1,2` on the output channel.

To understand the role of these control symbols, we need some information on how SQLfast decodes a statement. The components of an SQLfast statement are separated by one or several space characters. Statements `'set X = 123;'` and `'set  X  =  123;'` are equivalent, while `'set X=123;'` is invalid[3]. While leading spaces are ignored (as in `'  set X = 123;'`), trailing spaces are included in the statement. Statements `'set X = Bye;'` and `'set X = Bye  ;'` store different values in variable `X`.

If we want to store spaces in a variable, we have a problem, since all the leading spaces of the values will be discarded. For instance, in statement `'set head2 =  | ;'`, the (intentional) space character that immediately precedes `'|'` will be ignored, though the last one, between `'|'` and `';'` will be kept. In such problematic situations, an intentional space character can be replaced by special symbols `'@s'` or `'@Sn'`, that will be replaced just before the character string in which it appears is used. The role of this control symbol that appears in the statements above is clear: value `'@s| '`, for instance, just means `' | '`, as would also `'@s|@s'`.

Now, an assignment such as `'data4 = @s|@n;'` makes sense: first write one space character (`@s`), then vertical bar `'|'` then go to the next line (`@n`).

Two control symbols `@h` and `@v` are particularly important. They appear in variables `head2` and `data2`, where they tell SQLfast that column names (`@h`) and values (`@v`) must appear between separators. These symbols could appear in other variables and in other lines, as we will observe in Key-Value and XML formats for instance.

The standard settings are built in SQLfast engine but they can also be found in script UTIL-SELECT-parameters-standard-1.sql. It will be used for instance to reset the output layout after writing with special formats.

## 16.2 Changing the layout of an output table

Though they are built-in and are assigned initial values, the 20 table formatting variables are just plain variables, so that we can modify their values to change the aspect of the output tables. For instance, many text books devoted to SQL show tabular results in a simpler layout such as the following:

```
 CustID   Name
======== ===========
 B112     HANSENNE
 F010     TOUSSAINT
 C400     FERARD
```

There are no top bar, bottom bar nor vertical lines. This format can be obtained by the following variable assignments:

3. This rule does not apply to pure SQL statements, where the SQL syntax prevails. Another exception: the element separator in list is a *comma* (`','`) as in `A,B,C`.

```
set bar11 = ;          set bar21 = ;          set bar31 = ;
set bar12 = ;          set bar22 = =;         set bar32 = ;
set bar13 = ;          set bar23 = @s;        set bar33 = ;
set bar14 = ;          set bar24 = @n;        set bar34 = ;

set head1 = ;          set data1 = ;
set head2 = @h;        set data2 = @v;
set head3 = @s;        set data3 = @s;
set head4 = @n;        set data4 = @n;
```

These settings can be found in script UTIL-SELECT-parameters-standard-2.sql.

If all the variables of a row are empty, this row will not be written on the output channel. To suppress the top bar and the title row, we just declare their eight variables empty. With the standard layout, this will produce the following output:

```
+--------+-----------+
| B112   | HANSENNE  |
| F010   | TOUSSAINT |
| C400   | FERARD    |
+--------+-----------+
```

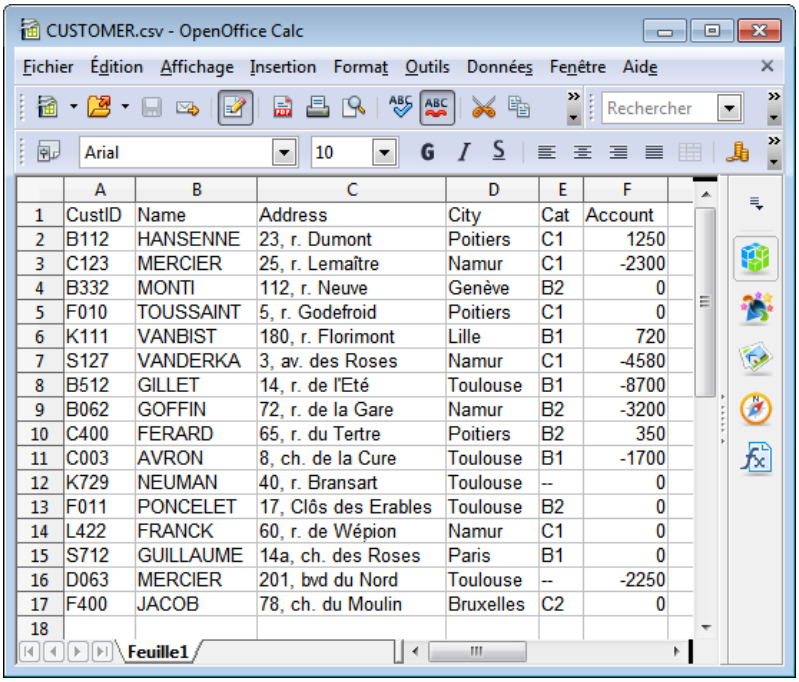
## 16.3 CSV output format

CSV, or *Comma separated values*, is a very simple format to exchange data in a tabular form. It is one of the Microsoft Excel external format, but is has been adopted by many other data management softwares, which are able to read and generate CSV data. Most DBMS offer import/export functions based on this format. However, we will see that CSV generation of data extracted from the database is fairly easy. Let us first analyze the CSV format, through the following example.

```
CustID;Name;Address;City;Cat;Account
B112;HANSENNE;23, r. Dumont;Poitiers;C1;1250.0
C123;MERCIER;25, r. Lemaître;Namur;C1;-2300.0
. . .
D063;MERCIER;201, bvd du Nord;Toulouse;null;-2250.0
F400;JACOB;78, ch. du Moulin;Bruxelles;C2;0.0
```

Each record of table CUSTOMER is represented by a list of values, separated by `';'` characters[4]. The first row provides the column names. Figure 16.1 shows how Excel could interpret this text.

Unfortunately, the CSV format is not unique and complies with no agreed upon standard. Everybody can define his own variant depending on the nature of source data, on the environment and on the format expected by the software that will import these data.




| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | CustID | Name | Address | City | Cat | Account |
| 2 | B112 | HANSENNE | 23, r. Dumont | Poitiers | C1 | 1250 |
| 3 | C123 | MERCIER | 25, r. Lemaître | Namur | C1 | -2300 |
| 4 | B332 | MONTI | 112, r. Neuve | Genève | B2 | 0 |
| 5 | F010 | TOUSSAINT | 5, r. Godefroid | Poitiers | C1 | 0 |
| 6 | K111 | VANBIST | 180, r. Florimont | Lille | B1 | 720 |
| 7 | S127 | VANDERKA | 3, av. des Roses | Namur | C1 | -4580 |
| 8 | B512 | GILLET | 14, r. de l'Eté | Toulouse | B1 | -8700 |
| 9 | B062 | GOFFIN | 72, r. de la Gare | Namur | B2 | -3200 |
| 10 | C400 | FERARD | 65, r. du Tertre | Poitiers | B2 | 350 |
| 11 | C003 | AVRON | 8, ch. de la Cure | Toulouse | B1 | -1700 |
| 12 | K729 | NEUMAN | 40, r. Bransart | Toulouse | -- | 0 |
| 13 | F011 | PONCELET | 17, Clôs des Erables | Toulouse | B2 | 0 |
| 14 | L422 | FRANCK | 60, r. de Wépion | Namur | C1 | 0 |
| 15 | S712 | GUILLAUME | 14a, ch. des Roses | Paris | B1 | 0 |
| 16 | D063 | MERCIER | 201, bvd du Nord | Toulouse | -- | -2250 |
| 17 | F400 | JACOB | 78, ch. du Moulin | Bruxelles | C2 | 0 |
| 18 | | | | | | |

**Figure 16.1 -** How Open Office Calc renders the CSV format

Some examples:

- should character strings be quoted? with which characters (e.g., `"`, `'`)?
- can we quote numbers? In this case, will they still be interpreted as numbers?
- what if this quote character appears inside a character string? Must we double it, protect it with an escape character (e.g., `\"`) or use another delimiter?
- what separator will be used (e.g., `,`, `;`, |, tab)?
- what if the separator appears inside a value? in particular, how do we represent numbers with a decimal part (3.1416 or 3,1416)?
- can a character string include control characters (e.g., *new line*, *tabulation*)?
- if the target table is known, is the first row specifying column names necessary?

4. Right, a semi-colon is not a comma, so, this should be an example of SCSV, or *Semi-colon separated format*, a format that doesn't exist (yet). This is a problem of the CSV format: its syntax depends of the *localization* of the computer. This text has been generated on a machine in which π is written 3,1416 and not 3.1316. Since comma is already used in number representation, the item separator in lists is `';'` instead of `','`. The problem is worsen by the fact that the DBMS uses the dot format to represent numbers. Coping with this issue is beyond the scope of this introduction.

With these interesting questions in mind, let us go back to the problem of generating CSV data.

Of course, if the CSV format may seem quite different from the tabular layout, it includes the same information, the differences lying in the presentation:

- there is no bars
- the first and last symbols of `head` and `data` are missing
- the column separators `head3` and `data3` comprise the CSV value separator.

We also decide to quote all the values and to use the semi-colon character (`';'`) as a value separator.[5] Should values include this character, then we would change the separator character.

We translate these facts into the following variable values:

```
set bar11 = ;          set bar21 = ;          set bar31 = ;
set bar12 = ;          set bar22 = ;          set bar32 = ;
set bar13 = ;          set bar23 = ;          set bar33 = ;
set bar14 = ;          set bar24 = ;          set bar34 = ;

set head1 = @n;        set data1 = ;
set head2 = @h;        set data2 = "@v"
set head3 = ;;         set data3 = ;;
set head4 = @n;        set data4 = @n;
```

The result is as follows:

```
CustID;Name     ;Address              ;City     ;Cat;Account
"B112  ";"HANSENNE ";"23, r. Dumont       ";"Poitiers ";"C1 ";"1250 "
"C123  ";"MERCIER  ";"25, r. Lemaître     ";"Namur    ";"C1 ";"-2300"
. . .
"D063  ";"MERCIER  ";"201, bvd du Nord    ";"Toulouse ";"-- ";"-2250"
"F400  ";"JACOB    ";"78, ch. du Moulin   ";"Bruxelles";"C2 ";"0    "
```

Not bad but a bit awkward: the spaces characters that define the vertical alignment are no longer needed and should be discarded as well.

The fundamental difference between the tabular and CSV formats is that the first one is intended for human readers while the second one, though readable, mainly is a machine format. Humans like tidy, well aligned, information while machines do not care. So, it would be nice to *deactivate* the alignment formatting when writing CSV data in the output file. This is the role of a new statement:

```
parameter select-align = N;
```

This statement sets a global parameter that will be taken into account by all the **`select`** statements from now on. The **`select-align`** parameter indicates whether

5. Interestingly, MS Excel and Open Office are able to interpret quoted numerics as pure numerics.

data values must be aligned (**Y**) in further `select` output or not (**N**). Here we choose not to align data (**select-align** = N). As a consequence, the internal surrounding spaces are suppressed. Now the output is exactly what we wanted it to be.

If generating CSV format is frequent, we can improve this script. First, we create a small *style* script that includes the settings for CSV generation: parameter `select-align`, `bar1`, `bar2`, `Bar3`, `head`, `data` (Script 16.1) This script is available in file UTIL-SELECT-parameters-for-CVS.sql.[6] The generation for table CUSTOMER now becomes quite simple (Script 16.2).

```
parameter select-align = N;
set bar11 = ;
. . .
set bar21 = ;
. . .
set bar31 = ;
. . .
set head1 = @n;
set head2 = @h;
set head3 = ;;
set head4 = @n;
set data1 = ;
set data2 = "@v"
set data3 = ;;
set data4 = @n;
```

**Script 16.1 -** Settings for the generation of CSV data [script UTIL-SELECT-parameters-for-CVS.sql]

```
execSQL UTIL-SELECT-parameters-for-CVS.sql;
select * from CUSTOMER;
```

**Script 16.2 -** A simple CSV generator

This script writes the result of the **select** statement (and of other statements, such as `write`), on the output channel. If we want to use the resulting text as a spreadsheet, it would be better to write it in a *user output file*, as discussed in Section 11.3:

```
outputOpen CUSTOMER.csv
```

6. Actually, the standard distribution provides two variants: `UTIL-SELECT-parameters-for-CVS1.sql` and `UTIL-SELECT-parameters-for-CVS2.sql`. The first one generates *semi-colon separated values* while the second one is pure CVS.

The new file can be opened and viewed with any spreadsheet processors (Script 16.3). Statement `execProg` runs an external program, here the default spreadsheet processor, against a source file (Output/CUSTOMER.csv). The default processor has been specified in the SQLfast.ini initialization file through parameter `WSprocessor`.

```
outputOpen CUSTOMER.csv;
execSQL UTIL-SELECT-parameters-for-CVS.sql;
select * from CUSTOMER;
outputAppend window;
execProg $WSprocessor$ Output/CUSTOMER.csv;
```

**Script 16.3 -** Generating and viewing a CSV file [Generator-for-CSV.sql]

## Complements

- If generating *csv* data from a database is an important function, the converse also is true. SQLfast has a statement for it: **`import`** `CUSTOMER = CUSTOMER.csv`, through which the data of a *csv* file can be loaded into existing table `CUSTOMER` or into a new one, created from the field names of the first record of the *csv* file.

  This powerful statement will be studied in Chapter 18.

- The `select` statement is not the only way to generate *csv* files. Besides the usual *export* functions of the DBMS, SQLfast scripts can be developed to take into account specific requirements that are not addressed by the `select` generator. Most languages also can access libraries to quickly build *csv* readers and writers. *CSVReader/CSVWriter* for Java and the *csv* module of Python are some examples.

## 16.4 Key-Value output format

The *Key-Value* format is a popular way to present the values of a list of items denoted by their names[7]. It is often used as a lightweight alternative to the verbose XML. The example below shows how we could write the contents of a row from table `CUSTOMER`:

```
CustID = B112
Name = HANSENNE
Address = 23, r. Dumont
City = Poitiers
```

7. It is used, among many others, in the POST method of HTTP, in OpenID, in log files, in parameter passing methods in many languages, in some NoSQL DBMS (key-value stores, such as Redis, Berkeley DB and Cassandra) and ... in the SQLfast initialization file (SQLfast.ini).

```
Cat = C1
Account = 1250.0
```

Each row of this table will be written in the same way. Different variants exist, notably on the way the value is connected to the column name: equal sign, space, colon, vertical bar, tabulation, etc.

At first glance, deriving this format from the `select` statement seems a bit more tricky than for other formats, with which there are two main differences: the column values are written on distinct lines and the column names are duplicated for each data row, and appear as a prefix of data values.

We will address them as follows.

- First of all, the vertical alignment is not required, hence the statement '**parameter** select-align = N;'.
- The three horizontal bars and the title row are omitted.
- Presenting values vertically in distinct rows, instead of horizontally, in a single row, is obtained by introducing a *new line* control character (`@n`) in the value separator `data3`.
- Finally, each data value is preceded by the column name and character `'='`. The column name is introduced through control symbol `'@h'` in variable `data2`, before control symbol `'@v'`.

Script 16.4 includes these settings.

```
parameter select-align = N;
set bar11 = ;
. . .
set bar21 = ;
. . .
set bar31 = ;
. . .
set head1 = ;
. . .
set data1 = ;
set data2 = @h = @v;
set data3 = @n;
set data4 = @n@n;
```

**Script 16.4 -** Settings for Key-Value formatted data [script UTIL-SELECT-parameters-for-KEY-VALUE.sql]

The generation of `CUSTOMER` rows can be written as Script 16.5. It first executes the external script UTIL-SELECT-parameters-for-KEY-VALUE.sql, in which the parameters and the layout variables are set as shown above. When the text file has been

generated, it is visualized with the standard text processor (here notepad.exe, as specified in file SQLfast.ini).

```
output CUSTOMER.txt;
execSQL UTIL-SELECT-parameters-for-KEY-VALUE.sql;
select * from CUSTOMER;
outputOpen window;
execProg $TXTprocessor$ Output/CUSTOMER.txt;
```

**Script 16.5 -** [Key-Value generation of the data of table CUSTOMER [Generator-for-KEY-VALUE.sql]

## 16.5 List output format

This format is intended to produce simple lists of values, one for each row. Values are separated by a comma, a semi-colon or any appropriate symbol:

```
B112; HANSENNE; 23, r. Dumont; Poitiers; C1; 1250
C123; MERCIER; 25, r. Lemaître; Namur; C1; -2300
B332; MONTI; 112, r. Neuve; Genève; B2; 0
...
D063, MERCIER, 201, bvd du Nord, Toulouse, --, -2250
F400, JACOB, 78, ch. du Moulin, Bruxelles, C2, 0
```

This format is quite similar to csv. All layout variables are set to empty, but three:

```
set data2 = @v;
set data3 = ; ;
set data4 = @n;
```

The settings for the list format are available in script UTIL-SELECT-parameters-for-lists.sql.

## 16.6 Text output format

This simple format derives from the List format. With such query,

```
select 'The balance account of customer ',Name,
       ', living in ',Address,
       ' in ',City,
       ', is ',Account
from CUSTOMER;
```

It allows the production of natural-like sentences from column data:

```
The balance account of customer HANSENNE, living in 23, r. Dumont
in Poitiers, is 1250.
...
The balance account of customer JACOB, living in 78, ch. du Moulin
in Bruxelles, is 0.
```

All layout variables are set to empty, but two:

```
set data2 = @v;
set data4 = .@n;
```

The settings for the list format are available in script UTIL-SELECT-parameters-for-text.sql.

## 16.7 Summary

SQLfast **`select`** statement writes the result of an SQL **`select`** query on the output channel in a tabular form. This form can be customized to a large extent.

- The tabular format is defined by 20 display system variables, that specify how the column names, the values and the frame must appear on the output channel.
- Modifying the values of these variables provides different output formats.
- *CSV* (*comma separated values*) format and *Key-Value* format can be specified by appropriate values of the display variables. Two scripts comprising these specifications are available:
  - UTIL-SELECT-parameters-for-CVS.sql for CSV generation
  - UTIL-SELECT-parameters-for-KEY-VALUE.sql for Key-Value generation.
- The standard layout is built in SQLfast but is also available in script UTIL-SELECT-parameters-standard-1.sql. A slightly different layout is provided by script UTIL-SELECT-parameters-standard-2.sql.
- More formats will be developed in the next chapter.

# Chapter 17

# Writing data tables - Part 2

**Objective**: In this second part on formating data extracted from the database, we will learn how to produce data in more sophisticated formats: XML, HTML, tuples, RTF, SQL, LateX and Json.

**Keyword**: display format, XML, HTML, tuples, RTF, SQL, LateX, Json

**Statements and functions**: select, execProg, parameter

**System variables and parameters**: selectIntro, selectClose, select0, selectA0, selectAi, select-align

## 17.1 XML output format

The XML data exchange standard offers at least two important advantages: its popularity and the many tools that can process it. In particularity, the XSLT language allows XML documents to be transformed into any other data presentation format[1].

A standard XML document for the contents of the CUSTOMER table could look like this text:

1. In some sense, we could have limited the development of these two chapters to XML + a collection of XSL scripts!

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<table Name = "CUSTOMER">
   <CUSTOMER>
      <CustID>B112</CustID>
      <Name>HANSENNE</Name>
      <Address>23, r. Dumont</Address>
      <City>Poitiers</City>
      <Cat>C1</Cat>
      <Account>1250.0</Account>
   </CUSTOMER>
   . . .
   <CUSTOMER>
      <CustID>F400</CustID>
      <Name>JACOB</Name>
      <Address>78, ch. du Moulin</Address>
      <City>Bruxelles</City>
      <Cat>C2</Cat>
      <Account>0.0</Account>
   </CUSTOMER>
</table>
```

At first glance, this format is fairly close to that of *Key-Value*. Each value line comprises a value ("`@v`") surrounded by twice the column name ("`@h`") into angle brackets (we just add two tab symbols ("`@t@t`") for pretty printing):

```
set data1 = ;
set data2 = @t@t<@h>@v</@h>;
set data3 = @n;
set data3 = @n;
```

This gives the expected row contents:

```
      <CustID>B112</CustID>
      <Name>HANSENNE</Name>
      <Address>23, r. Dumont</Address>
      <City>Poitiers</City>
      <Cat>C1</Cat>
      <Account>1250.0</Account>
```

However, this fragment must form a CUSTOMER XML element, and must appear between expressions `<CUSTOMER>` and `</CUSTOMER>` that define this element:

```
<CUSTOMER>
   <CustID>B112</CustID>
   <Name>HANSENNE</Name>
   <Address>23, r. Dumont</Address>
   <City>Poitiers</City>
   <Cat>C1</Cat>
   <Account>1250.0</Account>
</CUSTOMER>
```

So far, we can use column names ("@h"), column values ("@v") but we have no means to include the name of the table into the lines generated. This name could be extracted from the SQL statement (the from clause) but this is only valid for *single-table* queries. For more complex queries, the result has no name, so that we must solve this problem in a more general way. This is the role of control symbols "@0" to "@4" that denote the value of the five *built-in user variables* named select0 to select4. This role is simple: all the instances of "@*i*" are replaced by the value of variable select*i*.

Therefore, if we assign to variable select0 the name of the current table, or any name that we find pertinent, this name can be used anywhere in a formatted line through control symbol "@0".

```
set select0 = CUSTOMER

set data1 = @t<@0>@n;
set data2 = @t@t<@h>@v</@h>;
set data3 = @n>;
set data4 = @n@t</@0>@n
```

We are not completely done. Indeed, in the format we have chosen above, all the XML document should start with some standard information, such as,

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<table Name = "CUSTOMER">
```

and should include the closing clause

```
</table>.
```

We could write these lines in the output file from the main script but this is not particularly elegant. We could also use formatted lines bar1 and bar3 to write these lines but, once again, this would be rather tricky. SQLfast includes two special built-in formatting variable, named selectIntro and selectClose that, when not empty, are written before and after the contents of the table. Their value can include any number of control symbols of any kind.

```
set select0 = CUSTOMER
set selectIntro = <?xml version="1.0" encoding="iso-8859-1"?>
                  @n@n<table Name = "@0">@n@n;
set data1 = @t<@0>@n;
set data2 = @t@t<@h>@v</@h>;
set data3 = @n>;
set data4 = @n@t</@0>@n
set selectClose = </table>@n
```

**Script 17.1 -** Parameter and variable settings for XML generation [UTIL-SELECT-parameters-for-XML.sql]

The generation of the contents of table CUSTOMER in XML could be written as Script 17.2. It first executes external script UTIL-SELECT-parameters-for-XML.sql, in which the parameters and the layout variables are set as above. Note that variable `select0` is set in the main script, through an **`ask`** statement, and not in the parameter script. Double-clicking on the output file automatically opens it in the default browser.

```
execSQL UTIL-SELECT-parameters-for-XML.sql;
ask select0 = Enter a table name;
outputOpen $select0$.xml;
select * from $select0$;
```

**Script 17.2 -** XML generation of the data of the table specified by the user [Generator-for-XML.sql]

## 17.2 HTML output format

HTML code from the data of table CUSTOMER can be produced from the XML format or can be generated directly through an SQLfast script. This code could be the following. Its rendering in any browser would be something like Figure 17.1.

```
<html>
<table border="1">
  <tr bgcolor="#FEFC1E">
    <th>CustID</th>
    <th>Name</th>
    <th>Address</th>
    <th>City</th>
    <th>Cat</th>
    <th>Account</th>
  </tr>
  <tr>
    <td>B112</td>
    <td>HANSENNE</td>
```

```
    <td>23, r. Dumont</td>
    <td>Poitiers</td>
    <td>C1</td>
    <td>1250</td>
  </tr>
  . . .
  <tr>
    <td>B332</td>
    <td>MONTI</td>
    <td>112, r. Neuve</td>
    <td>Genève</td>
    <td>B2</td>
    <td>0</td>
  </tr>
</table>
</html>
```

| CustID | Name | Address | City | Cat | Account |
|---|---|---|---|---|---|
| B112 | HANSENNE | 23, r. Dumont | Poitiers | C1 | 1250 |
| C123 | MERCIER | 25, r. Lemaître | Namur | C1 | -2300 |
| B332 | MONTI | 112, r. Neuve | Genève | B2 | 0 |

**Figure 17.1 -** HTML code rendered by a browser

The layout is similar to that of XML, except for the header line, which must be defined explicitly. We therefore have to describe a header section (Script 17.3). The main generation script is shown in Script 17.4.

```
parameter select-align = N;

set selectIntro = <html>@n<table border="1">@n;

set head1 = @t<tr bgcolor="#FEFC1E">@n;
set head2 = @t@t<th>@h</th>;
set head3 = @n;
set head4 = @t</tr>@n;

set data1 = @t<tr>@n;
set data2 = @t@t<td>@v</td>;
set data3 = @n;
set data4 = @n@t</tr>@n;

set selectClose = </table>@n</html>@n;
```

**Script 17.3 -** Parameter and variable settings for HTML generation [UTIL-SELECT-parameters-for-HTML.sql]

```
outputOpen CUSTOMER.sql;
execSQL UTIL-SELECT-parameters-for-HTML.sql;
select * from CUSTOMER;
```

**Script 17.4 -** HTML generation of the data of table CUSTOMER [Generator-for-HTML.sql]

## 17.3 Tuple output format

The *tuple* format is not standard but is the preferred way to declare a relation and its tuples in the scientific literature on the relational model. Considering a relation[2] `R` with attributes `A`, `B` and `C`, this relation will be denoted by

```
R(A,B,C)
```

A tuple of this relation comprising values `a`, `b` and `c` will be noted

```
R(a,b,c)
```

or, to avoid any ambiguity,

```
R(A:a,B:b,C:c)
```

If attribute values may include commas (this is the case of Address), we can use semi-colons as separators, or we can quote values. Applying this concise format to table `CUSTOMER` would yield the following text:

```
CUSTOMER(CustID; Name; Address; City; Cat; Account)

CUSTOMER(CustID:B112; Name:HANSENNE; Address:23, r. Dumont;
         City: Poitiers; Cat:C1; Account: 1250.0);
CUSTOMER(CustID:C123; Name:MERCIER; Address:25, r. Lemaître;
         City:Namur; Cat:C1; Account:-2300.0);
. . .
CUSTOMER(CustID:D063; Name:MERCIER; Address:201, bvd du Nord;
         City:Toulouse; Account:-2250);
CUSTOMER(CustID:F400; Name:JACOB; Address:78, ch. du Moulin;
         City:Bruxelles; Cat:C2; Account:0.0);
```

This is a hybrid format, between CSV and XML. As in XML, we need the name of the table for the header and each data row. It is transmitted through variable `select0`, denoted by control symbol `@0`. The three bars are empty.

2. The relational theory uses a vocabulary that is slightly different from that of commercial databases systems. *Relation* is the noble term for *table*, *attribute* for *column* and *tuple* for *row.*

```
parameter select-align = N;
set bar11 = ;
. . .
set bar21 = ;
. . .
set bar31 = ;
. . .
set head1 = @0(;
set head2 = @h;
set head3 = ; ;
set head4 = )@n
set data1 = @0(;
set data2 = @h:@v;
set data3 = ; ;
set data4 = );@n;
```

**Script 17.5 -** Parameter and variable settings for TUPLE generation. The schema of the table is displayed as well. [UTIL-SELECT-parameters-for-TUPLE.sql]

```
execSQL UTIL-SELECT-parameters-for-TUPLE.sql;
ask select0 = Enter a table name;
outputOpen $select0$.txt;
select * from $select0$;
```

**Script 17.6 -** TUPLE generation of the data of the table specified by the user [Generator-for-TUPLE.sql]

## 17.4 RTF output format

The *Rich Text Format* (RTF) is a Microsoft format for storing and exchanging complex, formatted documents. A Word document can be saved in `*.doc`, `*docx` or `*.rtf` format. The RTF language is fairly complex but, here, we are concerned only by the way tables are described.[3] The table of Figure 17.2 results from the MS Word interpretation of the data provided by the execution of statement:

```
select CustID, Name, Address from CUSTOMER;
```

3. Reference: http://msdn.microsoft.com/en-us/library/aa140283%28v=office.10%29.aspx. See also, Sean M. Burke, *RTF Pocket guide*, O'Reilly, 2003

**Table CUSTOMER**

| CustID | Name | Address |
|---|---|---|
| B112 | HANSENNE | 23, r. Dumont |
| C123 | MERCIER | 25, r. Lemaître |
| B332 | MONTI | 112, r. Neuve |
| F010 | TOUSSAINT | 5, r. Godefroid |
| K111 | VANBIST | 180, r. Florimont |
| S127 | VANDERKA | 3, av. des Roses |
| B512 | GILLET | 14, r. de l'Eté |
| B062 | GOFFIN | 72, r. de la Gare |
| C400 | FERARD | 65, r. du Tertre |
| C003 | AVRON | 8, ch. de la Cure |
| K729 | NEUMAN | 40, r. Bransart |
| F011 | PONCELET | 17, Clos des Erables |
| L422 | FRANCK | 60, r. de Wépion |
| S712 | GUILLAUME | 14a, ch. des Roses |
| D063 | MERCIER | 201, bvd du Nord |
| F400 | JACOB | 78, ch. du Moulin |

**Figure 17.2 -** MS Word rendering of the RTF file

The RTF text that produces this table is shown in Figure 17.3. We cannot develop a complete discussion on how RTF specifies tables but we will comment the different sections of the code we intend to generate.

- section [1] is the prologue. It identifies the (unique!) rtf version (`\rtf1`), the character set (`\ansi`), the default font, which is the #0 in the following font table (`\deff0`), a table of fonts, with only one font, Helvetica (`{\fonttbl{\f0 Helvetica;}}`), the size of the paper sheet (A4, portrait) and the four margins (2 cm). The unit is the 1/1440th of inch or 1/57th of millimeter.
- section [2] defines the title "**Table CUSTOMER**" as two paragraph (`\pard ...\par`), one for the title and the second one, comprising one space, for a nice thin blank line. The font size is 10 points for the title (`fs20` means 20 half points) and 6 for the blank line. The title is in boldface (`\b ...\b0`).
- section [3] specifies the font size (9 points) for the remaining of the text.
- section [4] defines the header of the table, displaying the column names. This header is an RTF row (`\trowd ...\row`). It comprises three subsections. First, the inter-column space setting (`trgaph`). Second, the graphical aspects of the three cells: a shading of 15% (`clshdng1500`), that is, light grey, and the absolute position of the right margin (respectively 2000, 4000, 6000, in 1/57th of millimeter). Third, the specification of the cell contents: boldface printing (from `\b` to `\b0`), and the contents of each cell as a paragraph (`\pard...\cell`). This paragraph is a part of the table (`\intbl`), the text is centered (`\qc`) and finally we provide the text to display (`CustID`, `Name` and `Address`)
- sections [5] define, similarly, the RTF rows that display the data extracted from the CUSTOMER table. There is no shading and the values are left justified (`\ql`).

```
{\rtf1\ansi\deff0{\fonttbl{\f0 Helvetica;}}                    [1]
\paperw11909\paperh16834
\margl1134\margr1134\margt1134\margb1134
{\pard\fs20\b Table CUSTOMER\b0\par}{\pard\fs12 \par}          [2]
\fs18                                                          [3]
\trowd\trgaph100                                               [4]
  \clshdng1500\cellx2000
  \clshdng1500\cellx4000
  \clshdng1500\cellx6000
  \b
  \pard\intbl\qc CustID\cell
  \pard\intbl\qc Name\cell
  \pard\intbl\qc Address\cell
  \b0
\row
\trowd\trgaph144                                               [5]
  \cellx2000\pard\intbl\ql B112\cell
  \cellx4000\pard\intbl\ql HANSENNE\cell
  \cellx6000\pard\intbl\ql 23, r. Dumont\cell
\row
\trowd\trgaph144                                               [5]
  \cellx2000\pard\intbl\ql C123\cell
  \cellx4000\pard\intbl\ql MERCIER\cell
  \cellx6000\pard\intbl\ql 25, r. Lemaître\cell
\row
. . .
\trowd\trgaph144                                               [5]
  \cellx2000\pard\intbl\ql F400\cell
  \cellx4000\pard\intbl\ql JACOB\cell
  \cellx6000\pard\intbl\ql 78, ch. du Moulin\cell
\row
}
```

**Figure 17.3 -** RTF code of the table of Figure 17.2

We first notice two characteristics that will simplify the generation:

- the columns have the same size, independently of their contents,
- there are no borders, as is standard in tables inserted in documents.

The prologue section [1] is constant, and will be specified through the SQLfast variable **`selectIntro`**.

The title section [2] includes a varying component, the SQL table name. As usual, we will inject it through control character `@0` which denotes SQLfast variable `select0`, in which we have stored this table name. We suggest to add this section, as well as section [3], to variable `selectIntro`, since they are independent of the structure of the RTF table.

The header section comprises two main subsections: the graphical aspects and the contents. We will specify the first one in `bar1` and the second one in `head` row.

While the formation of the `head` row is fairly standard, we are faced with a quite new problem to define `bar1`.

In `bar1`, indeed, each cell has a varying attribute, its *right margin*, which is not derived from SQL column names (`@h`) and values (`@v`). Indeed, they depend mainly on the page size and on the number of columns (all the column have the same size). In addition, we would like to enforce a maximum column width to avoid tables with one or two columns that span the whole page width, which would be undesirable in most situations.

SQLfast offers a specific control symbol, called **`@A`**, that could help us to solve this problem. `@A` denotes the couple of variables (**`selectA0`**, **`selectAi`**) that can be used to define an *arithmetic sequence*, the starting element being the value of `selectA0` and the increment the value of `selectAi`. The successive instances of `@A` that appear in a line of a table (`bar1`, `bar2`, `bar3`, `head` or `data`), are replaced by the successive numbers of the suite. For instance, for `selectA0 = 10` and `selectAi = 5`, the string "`X=`**`@A;`**`Y=`**`@A;`**`Z=`**`@A;`**" will be transformed into "`X=`**`10;`** `Y=`**`15;`**`Z=`**`20;`**". This can also be used to define a numbering, such as in:

```
set head2 = name[@A] = "@h"; ;
```

which will generate, with appropriate values of `selectA0` and `selectAi` the line,

```
name[1] = "Cnum"; name[2] = "Name", name[3] = "Address"
```

Our problem is clearly solved: by setting `selectA0 = 2000` and `selectAi = 2000`, we can generate the graphical subsection of the table header section (through row `bar1`).

The data sections [5] exhibit patterns quite similar to those of the header section. The graphical and contents subsections have been interlaced, in order to define a single pattern.

The generation parameters and variables are defined in Script 17.7.

```
parameter select-align = N;

set nCol = 3;
set maxWidth = 2000;
set pageWidth = 9641;
compute colWidth = $pageWidth$ / $nCol$;
if ($colWidth$ > $maxWidth$) set colWidth = $maxWidth$;
set selectA0 = $colWidth$;
set selectAi = $colWidth$;

set selectIntro = {\rtf1\ansi\deff0{\fonttbl{\f0 Helvetica;}}@n
                   \paperw11909\paperh16834
                   \margl1134\margr1134\margt1134\margb1134@n
                   {\pard\fs20\b Table @0\b0\par}@n
                   {\pard\fs12 \par}@n\fs18@n;

set bar11 = \trowd\trgaph100\clshdng1500;
set bar12 = \cellx@A@n;
set bar13 = \clshdng1500;
set bar14 = @n;

set head1 = \b;
set head2 = \pard\intbl\qc @h\cell;
set head3 = @n;
set head4 = \b0\row@n;

set bar21 = ;
set bar22 = ;
set bar23 = ;
set bar24 = ;

set data1 = \trowd\trgaph144@n;
set data2 = \cellx@A\pard\intbl\ql @v\cell;
set data3 = @n;
set data4 = \row@n;

set bar31 = ;
set bar32 = ;
set bar33 = ;
set bar34 = ;

set selectClose = @n};
```

**Script 17.7 -** Parameter and variable settings for RTF table generation [UTIL-SELECT-parameters-for-RTF.sql]

```
exec UTIL-SELECT-parameters-for-RTF.sql;
ask select0 = Enter a table name;
outputOpen $select0$.rtf;
select CustID, Name, Address from $select0$;
```

**Script 17.8 -** TUPLE generation of the data of the table specified by the user [Generator-for-RTF.sql]

## 17.5 SQL output format

Generating SQL `insert` statement for each row of a table is fairly straightforward. The following scripts (Script 17.9 and 17.10) produce an SQL script that creates a backup database with the same contents as that of table `CUSTOMER`:

```
createOrReplaceDB ORDERS-BAK.db;
   create table CUSTOMER(...);
   insert into CUSTOMER values('B112','HANSENNE',
        '23, r. Dumont','Poitiers','C1','1250.0')
   insert into CUSTOMER values('C123','MERCIER',
        '25, r. Lemaître','Namur','C1','-2300.0')
   . . .
   insert into CUSTOMER values('F400','JACOB',
        '78, ch. du Moulin','Bruxelles','C2','0.0')
commitDB;
closeDB;
```

This simple approach considers all values as character strings.[4] Smarter scripts can be developed, but they need information of data types that we have not discussed yet. More detailed information will be found in Chapter 20 and in Code Generation application, where the use of the SQLfast Data dictionary is explained and illustrated by *schema-driven* applications.

4. This is no problem for SQLlite, thanks to its *affinity* typing, according to which a value is stored as far as possible in the type declared for its column. If a string is stored in a numeric column, it is converted into an number if possible, otherwise it is stored as is.

```
parameter select-align = N;
set selectIntro = createOrReplaceDB @0;@n
    create table(...);;
set data1 = @tinsert into @1 values(;
set data2 = '@v';
set data3 = ,;
set data4 = )@n;
set selectClose = closeDB;@n;
```

**Script 17.9 -** Parameter and variable settings for SQL generation [UTIL-SELECT-parameters-for-SQL.sql]

```
outputOpen CUSTOMER.sql;
execSQL UTIL-SELECT-parameters-for-SQL.sql;
set select0 = ORDERS-BAK.db;
set select1 = CUSTOMER;
select * from $select1$;
```

**Script 17.10 -** Generation of a backup script for table CUSTOMER [Generator-for-SQL.sql]

## 17.6 LateX output format

The following LateX fragment will produce the table of Figure 17.4.

```
\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{CustID} & \textbf{Name} & \textbf{Address}
 & \textbf{City} & \textbf{Cat} & \textbf{Account} \\
\hline
\hline
B112 & HANSENNE & 23, r. Dumont & Poitiers & C1 & 1250 \\
C123 & MERCIER & 25, r. Lemaître & Namur & C1 & -2300 \\
B332 & MONTY & 112, r. Neuve & Genève & B2 & 0 \\
\hline
\end{tabular}
```

| CustID | Name | Address | City | Cat | Account |
|---|---|---|---|---|---|
| B112 | HANSENNE | 23, r. Dumont | Poitiers | C1 | 1250 |
| C123 | MERCIER | 25, r. Lemaître | Namur | C1 | -2300 |
| B332 | MONTY | 112, r. Neuve | Genève | B2 | 0 |

**Figure 17.4 -** LateX layout for table CUSTOMER

This fragment is interesting in that the column structure of the table requires two sections. The first section defines the number of columns and the vertical lines ("`begin{tabular}{|c|c|c|c|c|c|}`") and the second section provides the names of the columns (`\textbf{CustID} & \textbf{Name} &` etc.). In short, this fragment includes two header sections. We have used `bar1` to define the structure and `head` to define the column names.

```
parameter select-align = N;
set selectIntro = <header LateX>;
set bar11 = \begin(tabular){;
set bar12 = |c;
set bar13 = ;
set bar14 = |}@n\hline@n;
set head1 = ;
set head2 = \textbf{@h} ;
set head3 = & ;
set head4 = \\@n\hline@n\hline@n;
-- set bar2 and bar3 empty
set data1 = ;
set data2 = @v ;
set data3 = & ;
set data4 = \\@n;
set selectClose = \hline@n\end{tabular}@n;
```

**Script 17.11 -** Settings for the generation of LateX files [script UTIL-SELECT-parameters-for-LATEX.sql]

```
outputOpen CUSTOMER.tex;
execSQL UTIL-SELECT-parameters-for-LATEX.sql;
select * from CUSTOMER;
```

**Script 17.12 -** Generation of a LateX script for table CUSTOMER [Generator-for-LATEX.sql]

## 17.7 JSON output format

**Json** (*JavaScript Object Notation*) is a format allowing complex structured data to be described in a simple and intuitive way. Originated in the *JavaScript* language, it has, just like CSV, been adopted as a data exchange standard in other applications. It is close to both XML and Key-Value formats. However, it is simpler (and less powerful) than XML and more powerful than Key-Value. Here follows an example

of a Json object describing a person. This example is intuitive enough to go without explanation (see description at http://json.org for example).

```
{"PID":"P123",
 "Name":"Smith, G.",
 "Address":{"Street":"Regent Street","City":"London"},
 "Phones":[1234,4567]}
```

The result of an SQL **`select`** query will be structured as a single *Json* object representing table **CUSTOMER**. It comprises two *Key-Value* pairs: **`table`**, the value of which is the table name, and **`rows`**, an array which collects the data of the table.

```
{"table":"CUSTOMER","rows":[r1,r2,...,rn]}
```

Each element *ri* of this array is an object made up of the data of a CUSTOMER row:

```
{"CustID":"B112","Name":"HANSENNE",
 "Address":"23, r. Dumont","City":"Poitiers","Cat":"C1",
 "Account":"1250"}
```

In the *Json* format definition, only `selectIntro`, `data` and `selectClose` rows are used. The script setting the generation parameters is shown in Script 17.13 while the generator is shown in Script 17.14.

```
parameter select-align = N;
set selectIntro = {"table":"@0","rows":[@n;;
-- set bar1, head, bar2 and bar3 empty
set data1 = @S2{;
set data2 = "@h":"@v";
set data3 = ,;
set data4 = },@n;
set selectClose = ]}@n;
```

**Script 17.13 -** Settings for the generation of Json files [script UTIL-SELECT-parameters-for-JSON.sql]

```
outputOpen CUSTOMER.js;
set select0 = CUSTOMER
execSQL UTIL-SELECT-parameters-for-JSON.sql;
select * from CUSTOMER;
```

**Script 17.14 -** Generation of a Json file for table CUSTOMER [Generator-for-JSON.sql]

# Chapter 18

# Importing and exporting data

**Objective**: This chapter studies the process of transferring high volumes of data from external data sources to a target database, and conversely. In particular, it describes procedures to exchange data between a relational database on the one hand, and **csv** text files, **dBase** databases, **SQL-DML** scripts and **JSON** files on the other hand.

**Keyword**: data migration, csv format, dbf format, JSON format, legacy data, import, export, data cleaning, data analysis, binary data

**Statements and functions**: import, select, itemLen, item, file-for loop, isNumeric, isInteger, isDate, Xcompliant, fileNames

**System variables and parameters**: dispNull, csv-separator, csv-header, csv-maxerrornbr, ImportErrors

## 18.1 External data import

A database is not an isolated masterpiece. It continuously exchanges data with its environment. Besides interactive data entry and data extraction through dialogue boxes, a database produces on a regular basis high volume of data (**export** function) and receives equally high quantities of external data (**import** function). This chapter studies four representative import/export procedures provided by the SQLfast platform, addressing respectively the exchange **csv** data, **dBase** files, **SQL-DDL** scripts and **JASON** files.

Most RDBMS include import and export functions that can be used to distribute data, to create backup files, to exchange data with other softwares and, in particular to migrate data from a DBMS to another one. The import/export functions of SQLfast, besides being integrated to the SQLfast language, include some refined features that are not provided by these standard processors, such as coping with untrusted, corrupt and untyped source data.

## 18.2 CSV data import / export

The **csv** format has been used for decades by Microsoft Excel spreadsheet software to save or to exchange pure data, but its simplicity and compactness[1] has made it very popular to exchange data between many other programs. It is also used as a save or a data entry format as it is both machine and human readable. We will examine in this section how to produce **csv** data from a database and how to store **csv** data into a table.

A **csv** file is a text file each line of which is a record made up of a list of values separated with a comma, hence the name *comma separated values*, as shown in this example:

```
B062,GOFFIN,rue de la Gare,Namur,B2,-3200
```

The first record can be used as a header that specifies the names of the fields:

```
CustID,Name,Address,City,Cat,Account
B062,GOFFIN,rue de la Gare,Namur,B2,-3200
C400,FERARD,rue du Tertre,Poitiers,B2,350
```

The values in a record are mere character strings, be they actual character strings, numbers, dates or whatever. They **may** be enclosed between double-quotes, even those which represent numbers. However, if a value comprises the separator symbol, it **must** be double-quoted:

```
"V007","Bergson","Regent Street","London","C1","-100"
S328,O'Hara,"27,Napoléon Avenue",Manchester,C1,120
```

Since a **csv** file has no schema, there is no way to know with certainty the intended type of the data fields. Actually, the interpretation of a value is up to the processor that reads the **csv** file. When stored in the row of a table, value `2017` will be considered a number, a date or a mere character string according to the type of the column in which it will be stored.

All the records comprise the same number of values. If a value is missing, its *inexistence* must be represented by an empty string, unless some proprietary conventional value is used (e.g., "N/A", or null):

1. as compared with XML and JSON for example

```
"V007","Bergson","Regent Street","London","","-100"
S328,O'Hara,"27, Napoléon Avenue",Manchester,,120
```

Here too, the interpretation of a missing value depends on the properties of the target column: if the column is *nullable*, the (missing) value translates into null, otherwise it is merely stored as an empty string.

Before discussing import and export tasks, we recall the principles of the **csv** format, specially as understood by SQLfast.

### 18.2.1 CSV syntax rules

**csv** is a *de facto standard*, that is a format that once was proprietary, limited to representing spreadsheet data, but that has since been adopted in many other contexts. Unfortunately, this has lead to variations in its syntax. For example, other separators have been used (such as semicolons (`;`), pipes (`|`) or tabs), the way double-quotes are represented inside a value (by doubling them, or by escaping them), the value of leading and trailing spaces or the way non printable characters are represented.

The following rules are those that define **csv** as understood by SQLfast.

- a record is terminated by an end-of-line control character (typically `CRLF`), except for the last one, for which this character may be missing
- all characters are significant, including leading and trailing spaces
- values *may* be double-quoted, included numbers
- a value that includes separator(s) **must** be double-quoted
- if a value that includes *tabulation* or *end-of-line* control characters **must** be double-quoted
- in double-quoted values, internal double-quotes must be doubled
- a missing value is coded as a 0-length string, double-quoted or not
- apostrophes are mere characters and need no special treatment.

The following examples are valid (and equivalent) **csv** records

```
P995,O'Hara,"27, Napoléon Avenue",Manchester,,1234
P995,O'Hara,27; Napoléon Avenue,Manchester,,1234
P995,O'Hara,"27, Napoléon Avenue",Manchester,,1234
"P995","O'Hara","27, Napoléon Avenue","Manchester","","1234"
```

and, if name `Napoléon` must be quoted:

```
P995,O'Hara,"27, ""Napoléon"" Avenue",Manchester,,1234
```

If we examine a **csv** file with a text editor, an *end-of-line* characters that is part of a value will appear as an *end-of-record*, which it is not, since the value is quoted:

```
P995,O'Hara,"27, Napoléon
Avenue",Manchester,,1234
```

### 18.2.2 Exporting data in csv format

Generating **csv** data is quite straightforward, as we have shown it in Section 16.3. We select the appropriate format, then we extract the data to convert:

```
execSQL UTIL-SELECT-parameters-for-CSV.sql;
select * from CUSTOMER;
```

Of course, it would be nice to send the result set to an external file, e.g., CUSTOMER.csv in folder Files:

```
execSQL UTIL-SELECT-parameters-for-CSV.sql;
outputOpen $fileDirectory$/CUSTOMER.csv;
select * from CUSTOMER;
```

After that, we restore the output channel:

```
outputAppend window;
```

If needed, we can also restore the standard display format:

```
execSQL UTIL-SELECT-parameters-standard-1.sql;
```

Practically, we could build a nice little tool as follows:

- variable **table** contains the name of the table to convert
- variable **condition** specifies which rows are to convert
- the conversion script we have detailed above is rewritten as in Script 18.1
- we save it with name **Save-Table-in-CSV.sql**.

```
execSQL UTIL-SELECT-parameters-for-CSV.sql;
outputOpen $fileDirectory$/$table$.csv;
if ('$condition$' = '');
   select * from $table$;
else;
   select * from $table$ where $condition$;
endif;
outputAppend window;
execSQL UTIL-SELECT-parameters-standard-1.sql;
```

**Script 18.1 -** Saving the contents of a table in a csv file

Now, saving selected rows in **csv** becomes quite easy:

```
set table = CUSTOMER;
set condition = City = 'Toulouse';
execSQL Save-Table-in-CSV.sql;
```

### 18.2.3 Refining export function

However, we can make this process more flexible to cope with different contexts. For this, SQLfast provides three system variables that will allow us to customize the **csv** format:

- **`dispNull`**: since a missing value is represented by an empty string in the **csv** file, we must ensure that *null* values are translated by the **`select`** statement into such empty strings
- **`csv-separator`**: the value separator required by the users of **csv** files must be changed if the standard symbol is not appropriate; if the **tab** control character is chosen, it will be specified by string **`@t`**
- **`csv-header`**: we have to choose to generate a header record (1) or none (0).

These variables can be set in initialization file **SQLfast.ini**:

```
dispNull =
csv-separator = ;
csv-header = 0
```

or, they can be set by the script as ordinary variables:

```
set dispNull = ;
set csv-separator = ;;
set csv-header = 0;
```

We can refine the style script for **csv** data we have developed in Chapter 16.3 (UTIL-SELECT-parameters-for-CSV.sql) to let it take into account variables **`csv-header`** and **`csv-separator`** (Script 18.2).

### 18.2.4 Values including double-quotes

To comply with the **csv** syntax, if a value includes double-quotes, the latter must be doubled (or escaped), as in this example:

Louis XIV, dit "Le Roi Soleil" ⟶ "Louis XIV, dit ""Le Roi Soleil"" "

This modification can be performed by function `replace`, as follows:

```
select ..., replace(Name,'"','""'),...
```

### 18.2.5 Exporting binary values

In Chapter 15, we have studied in some detail various techniques to manipulate binary data. This issue also concerns the way such data must be converted into the **csv** format. One can guess that a picture or a **pdf** document cannot be written as such in the **csv** file, even if we enclose it with double-quotes!

```
...
if ($csv-header$);
   set head1 = @n;
   set head2 = @h;
   set head3 = $csv-separator$;
   set head4 = @n;
else;
   set head1 = ;
   set head2 = ;
   set head3 = ;
   set head4 = ;
endif;
...
set data1 = ;
set data2 = "@v";
set data3 = $csv-separator$;
set data4 = @n;
...
```

**Script 18.2 -** Refinement of **csv** style script **UTIL-SELECT-parameters-for-CSV.sql**

The solution consists in converting a binary value into a character string with such coding as **base64**. This increases a little the size of the value but produces a pure ASCII text. The contents of table DOCUMENT described in Script 15.5 (its column Contents was declared of type **blob**), could be safely saved with this query:

```
select ID,Name,b64encode(Contents,'std')
from   DOCUMENT;
```

Since the values coded in base64 may be fairly large, it is recommended to increase the maximum size of these values in the result set, either in initialization file SQLfast.ini:

```
maxSelectWidth = 999999
```

or, dynamically, in scripts:

```
set maxSelectWidth = 999999;
```

### 18.2.6 Importing rows from a csv file

The simplest way to load **csv** data into a table is through statement **import**. It indicates the name of the target table and the name of the source **csv** file. To load the data of file **CUSTOMER.csv** into table **CUSTOMER**, we simply write:

```
import CUSTOMER from $fileDirectory$/CUSTOMER.csv;
```

To interpret the source data correctly, the operation needs to know two things: the value separator with which the file was written and whether the file comprises a header record. These informations are provided by system variables **csv-separator** and **csv-header**.

As already mentioned, a missing value (represented by an empty string) is translated into a null value if the column is *nullable*, otherwise it is merely stored as an empty string.

Whether the **csv** file includes a header record cannot be easily determined from the analysis of the file. Therefore, such a record is expected if, and only if the value of system variable csv-header is 1. If it is 0, the loader assumes that this record is missing and that the first record comprises data.

The behavior of the loader also depends on whether the target table already exists or not:

– *If the table already exists*, the data are added to this table. If the source file includes a header record, the latter is ignored. In any case, a value must exist for each columns in each data record. The way an empty value is interpreted depends on the *nullability* of the target column.

– *If the target table does not exist*, it is created with the name specified in the `import` statement and with as many columns as there are names in the header record. These columns are not nullable and their common type is (tentatively) `varchar(256)`. If all the values of a column extracted from the data records are not empty, this column is declared `not null`. Otherwise, it is declared nullable when the first empty value is encountered. Depending on the usage planned for this table, it may be useful to analyze its contents to refine the definition of its columns. We will examine this idea in Section 18.2.8.

### Managing csv loading errors

Bulk loading hundreds of thousands of **csv** records is likely to trigger some errors. The **csv** loader considers two kinds of errors, record structure and SQL.

– A *csv related error* arises when the csv file cannot be read correctly, when the new target table cannot be created or in case of character encoding problems.

– *SQL errors* are those that derive from the insertion of incorrect data in the target table (uniqueness, referential, value type, not null, etc.)

Here comes the third system variable that controls the import process: **csv-maxerrornbr**. If no more than csv-maxerrornbr errors have been detected during the operation, the loading is considered successful. Otherwise, it is aborted and all the insertions are rolled back. The number of record structure errors (e.g., not enough or too many values) and SQL errors (e.g., uniqueness, value or foreign key errors)) is provided by system variable **ImportErrors**.

In case of success (no more than csv-maxerrornbr errors), the insertions are committed at the Basic Level but not at the Expert Level 1, in which case a `commitDB` or an `abortDB` must be executed explicitly by the script.

The error limit can be set in initialization file SQLfast.ini:

```
csv-maxerrornbr = 10
```

or, dynamically, in scripts:

```
set csv-maxerrornbr = 10;
```

A summary of the errors is reported on the output channel and their detail is documented in a log file the name of which is that of the source file, with extension log (e.g., CUSTOMER.csv.**log**), located in the folder of the source file. The following extract of a log file shows that two records were invalid (not the right number of fields) and that one row was rejected due to uniqueness violation. When the data are stored in a new target table, the report also includes its SQL-DDL creation code.

```
[Loading CSV data in table "CLIENT_1" from file
         "D:\SQLfast\Files\CLIENT.csv"]
-- source record #3 ignored -- Wrong number of values
        [C123|MERCIER|25, r. Lemaître|Namur|C1|-2300|55]
-- source record #6 ignored -- Wrong number of values
        [K111|VANBIST|180, r. Florimont|Lille|720]
-- Uniqueness error in query "insert into CLIENT_1(NCLI,
         NOM,ADRESSE,LOCALITE,CAT,COMPTE) values('B332',
        'GUILLAUME','14a, ch. des Roses','Paris','B1',0);"

-- Table created --
    <no "create" query found>

-- Loading statistics for table CLIENT_1 --
      Header extracted:          1
      Data records extracted:    16
      Valid records extracted:   14
      Invalid records extracted: 2
      Rows generated:            14
      Rows inserted:             13
      Rows rejected:             1
[Loading completed with 3 error(s)]
```

## The loading workflow

The `import` function does not store the extracted data on the fly. Instead, it creates a script file in which the data are converted into `insert` queries. When this script is completed, the `create table` query is executed, in case of a new target table, then the script is executed to load the data in the target table. This way of proceeding has two advantages:

– it performs a preliminary analysis of the data extracted to detect nullable columns

– in case of abnormal loading, it allows the user to examine the SQL script to identify and manually fix the problems.

When the loading is successfully completed, this file can be deleted:

```
if ('$ImportErrors$' = 0)
   deleteFile $fileDirectory$/CUSTOMER.csv.sql;
```

### 18.2.7 Example: save and restore relational data

Script 18.3 saves the rows of the tables of database ORDERS.db into **csv** files. The list of table names is stored in variable listTables. Function **itemLenS**(L) returns the number of items in list L (suffix **S** indicates the standard separator ';') and function **itemS**(L,n) returns the $n^{th}$ item in list L.

Script 18.4 can be used to restore the database from this set of **csv** files.[2]

```
set dispNull = ;
set listTables = CUSTOMER;PRODUCT;CUSTORDER;DETAIL;
compute tabNb = itemLenS('$listTables$');
for tabId = [1,$tabNb$];
   compute table = itemS('$listTables$',$tabId$);
   outputOpen $fileDirectory$/$table$.csv;
   select * from $table$;
endfor;
outputAppend window;
```

**Script 18.3 -** Saving the data of ORDERS.db database into csv files

```
onError continue;
set listTables = CUSTOMER;PRODUCT;CUSTORDER;DETAIL;
compute tabNb = itemLenS('$listTables$');
for tabId = [1,$tabNb$];
   compute table = itemS('$listTables$',$tabId$);
   import $table$ from $fileDirectory$/$table$.csv;
   write $ImportErrors$ errors for table $table$;
   if ('$SQLdiag$' <> 'OK') exit;
endfor;
```

**Script 18.4 -** Reloading ORDERS.db database from csv files

### 18.2.8 Estimating column types through data analysis

When the source data are stored in a new table, all the columns are declared varchar(256), in the absence of anything better.

However, a more precise data type can be computed, or at least *guessed*, through the analysis of the set of values of each column. For this, SQLfast provides a collection of functions devoted to data analysis, among which:

2. Note that the order of table names in list listTables has been carefully chosen!

- `isNumeric(N)`, `isNotNumeric(N)`: indicates whether string or numeric value `N` represents a **numeric** value (float, integer, double, long, etc.)
- `isInteger(N)`, `isNotInteger(N)`: indicates whether string or numeric value `N` represents an **integer** value
- `isDate(D)`, `isNotDate(D)`: indicates whether string `D` represents a **date** (in any of the six standard representations: YYYY-MM-DD, MM-DD-YYYY, DD-MM-YY, etc.)
- `length(N)`: number of characters of string or numeric value `N`
- `utf8Compliant(S)`, `latinCompliant(S)`, `hexCompliant(S)`, `base64Compliant(S)`: indicates whether string value `S` is compliant with the `utf-8`, `latin`, `hexadecimal` or `base64` coding schemes.

The analysis of temporary table CUSTOMER_TEMP, just loaded from file CUSTOMER.csv, can be written as shown in Script 18.5.[3]

```
set columns = CustID;Name;Address;City;Cat;Account;
set query = ;
for iCol = [1,6];
  compute col = itemS('$columns$',$iCol$);
  set query = $query$ select '$col$' as "Column",;
  set query = $query$ count(*) as "Rows",;
  set query = $query$ count($col$) as "NotNull",;
  set query = $query$ count(distinct $col$) as "Val",;
  set query = $query$ sum(case when isNumeric($col$)
                              then 1 else 0 end) as "Num",;
  set query = $query$ sum(case when isInteger($col$)
                              then 1 else 0 end) as "Int",;
  set query = $query$ sum(case when isDate($col$)
                              then 1 else 0 end) as "Date",;
  set query = $query$ max(length(trim($col$))) as "MaxLen";
  set query = $query$ from CUSTOMER_TEMP;
  if ($iCol$ < 6) set query = $query$ union ;
endfor;
$query$;
```

**Script 18.5 -** Analyzing the value set of each column of table CUSTOMER_TEMP

Its result gives us interesting hints on the properties of the data:

- all the columns have significant values (Rows = NotNull) but Cat, which includes two *null* values (Rows - NotNull)
- the values of columns CustID and Address are unique (Val = NotNull), and are candidate for primary or unique keys

3. This script can be automatically generated from the dictionary tables, that will be studied in Chapters 20 and in Code Generation application

– all the values of column Account are numeric (Num = NotNull), and, more specifically, integers (Int = NotNull)
– all the columns, but Account, include values that are not numeric nor date. They can be declared character string.
– the lengths computed should probably be augmented (e.g., by 50%), due to the small size of the data sample.

| Column | Rows | NotNull | Val | Num | Int | Date | MaxLen |
|---|---|---|---|---|---|---|---|
| CustID | 16 | 16 | 16 | 0 | 0 | 0 | 4 |
| Name | 16 | 16 | 15 | 0 | 0 | 0 | 9 |
| Address | 16 | 16 | 16 | 0 | 0 | 0 | 20 |
| City | 16 | 16 | 7 | 0 | 0 | 0 | 9 |
| Cat | 16 | 14 | 4 | 0 | 0 | 0 | 2 |
| Account | 16 | 16 | 10 | 16 | 16 | 0 | 5 |

**Figure 18.1 -** Result from the data analysis of table CUSTOMER_TEMP

From these observations, we can refine the original tentative schema:

```
create table CUSTOMER_TEMP (
        CustID  varchar(256) not null,
        Name    varchar(256) not null,
        Address varchar(256) not null,
        City    varchar(256) not null,
        Cat     varchar(256),
        Account varchar(256) not null);
```

into this one (still to be validated by users:)

```
create table CUSTOMER (
        CustID  char(6)  not null primary key,
        Name    char(15) not null,
        Address char(30) not null unique,
        City    char(15) not null,
        Cat     char(3),
        Account integer  not null);
```

that defines the table in which we store the data:

```
insert into CUSTOMER select * from CUSTOMER_TEMP;
```

## 18.3 dBase data import

In 1980, Ashton-Tate published database management system *dBase II* for CP/M operating systems on microcomputers[4]. This software included a database manager and a business application programming language. dBase was then ported to Apple

II and IBM PC. This software was very successful, and was said to have triggered, with Visicalc and Lotus 123 (spreadsheet softwares), the popularity of PC's in business. dBase II was extended and redeveloped in C, under the name *dBase III*. dBase has evolved since then to become a modern database application development environment comparable to Microsoft Access. dBase products currently are distributed by dBase LLC (www.dbase.com).

dBase data files still are very common today, either as components of databases or as a store and exchange format for specialized softwares, such as geographic data format *Shapefile*.[5]

### 18.3.1 DBF file format

dBase data are stored in fixed format files that are (fairly) easy to read and write even outside dBase DBMS. For that, this format has become a very popular way to store large record sets. As a matter of fact, many softwares are able to read (and sometimes to update) dBase files (MS Access, Excel, ODBC drivers, among others). Importing data from dBase files is therefore an important function.

A dBase database is a set of files. The main file types are *data files* (*.dbf) and *memo files* (*.dbt). A data file comprises a header that describes the file and the data fields (name, type, length, decimals), followed by a sequence of fixed length records. dBase offers a standard set of data types such as *character*, *numeric* and *date*. The special *memo* data type corresponds to the SQL CLOB type (*character large object*). A *memo* field can store character strings of arbitrary length. These values are stored in a *memo file* the name of which is that of the data file, with extension '*.dbt'.

Though the formats of both file types have been widely published, the long evolution of the software (more than three decades), the multiplicity of publishers and of platforms, the weaknesses of the early versions as far as integrity is concerned and its tolerance to corrupt data and schemas may make the data extraction process more complex than it should be. On the other hand, the fact that column names and types are included in **dbf** files may make data extraction both easier and more precise than it was from **csv** files.

We will analyze the process of **dbf** data **import** but we will not address the inverse **export** function.

Broadly speaking, the **dbf** import process is similar to **csv** import.

### 18.3.2 Importing rows from a dbf file

Like for **csv** files, the simplest way to load **dbf** data into a table is through statement `import`, which indicates the name of the target table and the name of the source **dbf**

4. http://en.wikipedia.org/wiki/DBase
5. https://en.wikipedia.org/wiki/Shapefile

file. To load the data of file **CUSTOMER.dbf** into table **CUSTOMER**, we simply write:

```
import CUSTOMER from $fileDirectory$/CUSTOMER.dbf;
```

In early versions of dBase (dBase III and dBase III plus), the field descriptors do not indicate whether *null* values are allowed or forbidden. So, the **import** process will examine each value of each field to identify empty values and to decide what to do with them: either to leave them as they are or to transform them into *null* values.

The import process copes with four issues that are specific to **dbf** data:

– *the character coding scheme of the source data*: the coding scheme can be different from those commonly used in the SQLfast environment, typically Latin, cp1252 or utf-8 in the Windows environment (Mac roman and utf-8 in Mac OS). In the early versions of dBase III, the data were coded in cp850, the main 8-bit scheme in DOS application. The dBase coding scheme is specified in system variable **incoding_dbf**, that can be set in initialization file SQLfast.ini or in the script itself:

```
incoding_dbf  = cp850
set incoding_dbf  = cp850;
```

– *illegal column names*: the field naming conventions of dBase are different of those of SQL, so that dBase fields may be given SQL invalid column names. When a new target table is created, invalid field names are modified by adding a prefix:

```
-- Invalid DBF fieldname DATE replaced by T_DATE
-- Invalid DBF fieldname IN replaced by T_IN
```

– *deleted source records*: records deleted are not erased from the file but are flagged so that normal operations can skipped them. The `import` function ignores deleted records.

– *corrupt data*: many **dbf** files that are to be migrated to relational database contain data several decades old, created and processed by ageing applications. It is not uncommon that some parts of these files have become corrupt and therefore unreadable. The `import` function identifies these parts and discards them from the conversion process.

A log file is created to report on field name transformation and SQL insert errors. It also gives the import statistics and the schema of the new target table. The statistics report below shows that file BOOK.dbf is claimed (information from the file header) to contain 98,826 records, that this same number have really been extracted, among which 12,847 were in the *deleted* state and 456 were corrupt and discarded. From the valid **dbf** records, 85,523 `insert` queries were generated, all of them being inserted without problem.

```
-- Loading statistics for table BOOK --
      Records claimed:   98826
      Records extracted: 98826
```

```
Records deleted:   12847
Records corrupt:     456
Rows generated:    85523
Rows inserted:     85523
Rows rejected:         0
```

### 18.3.3 Example: migrating a dBase database

A library software, written in the eighties in Clipper (a dBase-compatible compiled language), comprises a database of 54 **dbf** files and 8 **dbt** memo files. Script 18.6 migrates the contents of these files into relational database LIBRARY.db. Statement `fileNames` stores in file dbfFile.txt the full names of all the **dbf** files of directory D:/LIBRARY. For each of these names, the loop extracts the table name (the local file name minus its extension) then imports the data of the file.

```
fileNames dbfList.txt = D:/LIBRARY,dbf;
for dbfFile = [file dbfList.txt];
   compute table = pureNodeName('$dbfFile$');
   import $table$ = $dbFile$;
endfor;
commitDB;
```

**Script 18.6 -** Migrating the **dbf** files of database LIBRARY to a relational database

## 18.4 SQL-DML data import/export

To be written

## 18.5 JSON data export/import

To be written

## 18.6 Summary

# Chapter 19

# Recursive programming

**Objective**: This chapter addresses database problems the solution of which involves the problem itself, that is, recursive solutions. It describes four powerful techniques, namely, recursive SQL queries, SQL recursive triggers, SQLfast recursive scripts and SQLfast loop-based recursion simulation. These techniques are illustrated by two case studies, computing the factorial of a number and managing the staff of a company.

**Keywords**: CTE, recursive CTE, recursive trigger, recursive procedure, tail recursion, tree, hierarchy, acyclic graph, circuit, recursion depth, breadth first traversal, depth first traversal, Turing machine

**Statements and functions**: with, with recursive, execSQL, for-endfor, union (all)

**System variables and parameters**: procLevel, maxProcLevel

## 19.1 Introduction

Solving a problem by dividing it into smaller subproblems, easier to solve, is a common strategy (called *divide and conquer*). We have done so in several examples developed in this tutorial, when a main script called one, two or several secondary scripts (or procedures) that each performs a specific subtask.

In many cases, a task must be carried out for each member of a series, possibly very large, of similar elements. So far, this has been translated into a **loop** structure,

the body of which generally includes calls to secondary scripts. Script 19.1 is an example of this pattern. If the problem to solve is particularly complex, these secondary scripts may themselves include calls to other, lower-level, scripts. Seen from a certain distance, the structure of these scripts is a tree, each node of which is a script with a specific goal. This tree is fixed since it is defined by the cascade of `execSQL` statements.

```
for ord = [select CustOrd from CUSTOMER where City = $cit$];
    execSQL computeTotalOrderAmount.sql;
endfor;
```

**Script 19.1 -** Solving a series of similar subproblems with a loop

However, some problems cannot be solved easily by such loops and secondary scripts. A popular example is that of the **genealogy** of a person. To display the **descendants** of person P (the *descent* of P), we must first display the children of P, if any, then the children of each children of P, then the children of each children of each children of P, and so on until there is no more children to display. Obviously, using a simple loop, or embedded loops, does not help[1], because we do not know the depth of the genealogy. The process also is represented by a tree. However, its shape is dynamic since it is defined by the data and not by the code of the script.

The trick here is to observe that we apply the same operation, *display the children of a person*, to each individual of each generation. So, we can reformulate the *display of the descendants* of P as follows (P is just a nickname for the current person who will be processed):

```
display-the-descendants of P ≡
   for each child C of P:
      display the name and birthdate of C
      display-the-descendants of C
```

That's all. This very simple algorithm can display the descendants of P, whatever the number of the generations originated from P. It is called **recursive** since it **calls itself** to solve one of its subproblem. A recursive solution of a problem comprises three main components:

1. The starting point: selecting source person P, of which we want to display the descent.
2. The processing of the current element: displaying the information of each child of the current person and displaying their descent.
3. The termination condition: the process stops naturally when a person has no children, which prevents the algorithm to call itself forever.

1. Though Section 19.8 will show that loops may sometimes be used to simulate recursion.

The complete solution generally comprises two scripts. The first one starts the process by selecting person P, of which we want to display the descent. The second one computes and displays the descendants of a person.

Many database problems naturally lead to recursive reasonings. Fortunately, SQL provides several way to translate them into simple recursive queries and procedures. We will study two of them, *recursive queries* and *recursive triggers*. When these features cannot be used, whatever the reason, then we will resort to special forms of SQLfast scripts: *recursive scripts* and *loop-based simulation* of recursive procedures.

## 19.2 SQL recursive queries

The recursive approach to problem solving has been addressed by SQL through the *Common Table Expression*, a concept that may look unimpressive at first glance, but a variant of which provides a nice way to recursive programming.

### 19.2.1 Common Table Expressions (CTE)

Recursive queries rely on a special kind of view called *Common Table Expression*, or CTE. When we want to declare an SQL view that is used in **one** SQL query **only**, we can use a CTE instead. Let us consider simple Script 19.2, that displays the number of orders placed by customers. It first creates view CITY_ORDERS that provides the number of orders in each city (where at least one order has been placed), then the **`select`** statement computes the sum of these numbers (don't try to find any logic in this complicated way to proceed).

```
create view CITY_ORDERS(City,Number)
as     select City,count(*)
       from   CUSTOMER C, CUSTORDER as O
       where  C.CustID = O.CustID
       group by City;
select sum(Number) from CITY_ORDERS;
```

**Script 19.2 -** Computing statistics through an SQL view [CTE-Query.sql]

If view CITY_ORDERS has no other usage, we can make it local to the **`select`** query as a CTE, as shown in Script 19.3.

```
with CITY_ORDERS(Number)
as   (select count(*)
      from   CUSTOMER C, CUSTORDER as O
      where  C.CustID = O.CustID
      group by City)
select sum(Number) from CITY_ORDERS;
```

**Script 19.3 -** Computing statistics through a CTE [CTE-Query.sql]

The **with** statement comprises two parts. The first one defines one or several CTE's (`CITY_ORDERS(Number) as (...)`) and the second part is the **main query** (`select sum(Number) from CITY_ORDERS`).

### 19.2.2 Recursive CTE

However, the real interest of the **with** query lies in its *recursive* variant, in which the definition of a CTE (its **as** part) may reference itself in its `from` clause. A recursive **with** query is processed in a very specific way that will be explained with a simple query displaying the factorial of integer number **N**. [2]

The factorial of a non negative integer number **N**, noted **N!**, is the product of all the integer numbers from 1 to N:

$$N! = 1 \times 2 \times 3 \times \ldots \times (N-1) \times N$$

with the particular case of N = 0, the factorial of which is 1. However, there exists a much more elegant way to define the factorial of N. The factorial of N is 1 if N = 0, and is N multiplied by the factorial of N-1 otherwise. More formally:

```
if N = 0 then
   N! = 1
else
   N! = N * (N-1)!
```

This definition is called *recursive* because it relies on itself. It can be easily translated into a recursive procedure, such as that of Script 19.33, but, for now, we will express it as an SQL **with** query. This could seem a bit odd, since this is not a database problem[3], but let us take it as an easy introductory example.

The main query [4] just extracts and displays data from CTE FACTORIAL.

2. Evaluating the factorial of N, though being the most popular introductory example of recursive computing, generally is considered a bad illustration since it can more simply be carried out by a mere loop as shown in Section 19.8.1.
3. All the more so as an SQL query can only be executed when a database is opened. Therefore, all the scripts computing numbers, like this one, must include `openDB` and `closeDB` statements.

FACTORIAL is a local view each row of which contains a number (column Nb) and its factorial (Fact). It is defined as the union of two `select` queries.

– The *initialization query* [1] describes a constant row[4] (0,1), initiating the definition, telling that *the factorial of 0 is 1*.

– The *recursive query* [3] describes how to derive the factorial of Nb + 1 once the factorial of Nb is known: *we add 1 to Nb and we multiply Fact by (Nb + 1)*.

```
ask N = Enter a number (from 0 to 175):;
with recursive FACTORIAL(Nb,Fact)
as  (select 0,1                                          [1]
       union all
     select Nb+1, Fact * (Nb+1)                          [2]
     from   FACTORIAL
     where  Nb < $N$                                     [3]
    )
select * from FACTORIAL where Nb = $N$;                  [4]
```

**Script 19.4 -** Computing the factorial of first N numbers through a CTE [CTE-Factorial.sql]

When the CTE is invoked in the main query, the initialization query is executed once and the recursive query is executed repeatedly until it returns an empty set. The role of condition (`Nb < $N$`) is to stop these executions after N runs, when this condition is no longer satisfied. However, these successive executions are organized in a way that is not as intuitive as it seems to be. We will borrow (and slightly modify) the description of the SQLite documentation[5]. It relies on an additional data structure called the *queue*. This queue is fictitious and is introduced for clarity only.

1. Run the *initialization query* and add the result to the queue.
2. While the queue is not empty
   - Extract a single row from the queue.
   - Insert that single row into the CTE table.
   - Pretend that this single row just inserted is the only row in the CTE table and run the *recursive query* against it, adding its result set to the queue.

### 19.2.3 *Union all* or *union*?

In many problems, the way the recursive query is evaluated prevents the contents of the CTE to include duplicates. This is true for the factorial example but also for tree

4. This particular form of the `select` query, where the `from` and `where` clauses are missing is a simple way to define a constant row. The `values` form can also be used instead, as it is in **`insert`** queries: `values (0,1)`.
5. http://www.sqlite.org/lang_with.html

processing (as in Section 19.3 and in Kings of France application). In these cases, eliminating duplicates through a `union` operator is pure waste of time[6]. Hence the use of `union all`, which is much less time consuming. In The book of which you are the Hero application) we will go back to the duplicate elimination issue.

Now that the mechanics behind the recursive CTE is clarified, we can tackle more interesting problems that cannot be solved easily in other ways.

## 19.3 SQL recursive queries: the *Staff* application

Recursive queries really shine in tree and graph processing, the most popular example being that of the staff of a company, where each person reports to their direct manager. For each person, we record the personal id (PID), the name (Name) and the personal id of the immediate manager of this person (Manager). Of course, the manager of an employee is a person. The database in which these data can be recorded is shown in Script 19.5. At some time, the contents of table PERSON could be that of Figure 19.1. It defines a **tree** structure, the **root** of which is person p01, depicted in Figure 19.2.

These data tell us that p07 (Dermiez, to whom nobody reports) reports to p06 (Dupont), who reports to p04 (another Dupont), who reports to p01 (Mercier) who reports to ... nobody, because Mercier is the CEO. If ps reports to pm, we say that ps is a **subordinate** of pm while pm is the **manager** of ps.

When a person is removed from the database, the rows of their subordinates are preserved, the value of column Manager being set to *null* (`on delete set null`).

```
createOrReplaceDB STAFF.db;
create table PERSON(
    PID     char(4)  not null primary key,
    Name    char(24) not null,
    Manager char(4),
    foreign key (Manager) references PERSON
            on delete set null
            on update cascade);
```

**Script 19.5 -** Creating the staff database [STAFF-create.sql]

```
+-----+---------+---------+
| PID | Name    | Manager |
+-----+---------+---------+
| p01 | Mercier | --      |
| p02 | Durant  | p01     |
| p03 | Noirons | p01     |
```

6. Remember that the `union` operator produces a **set**, even when the source tables include duplicates.

```
| p04 | Dupont  | p01 |
| p05 | Verger  | p04 |
| p06 | Dupont  | p04 |
| p07 | Dermiez | p06 |
| p08 | Anciers | p02 |
+-----+---------+-----+
```

**Figure 19.1 -** The staff [also in script STAFF-create.sql]

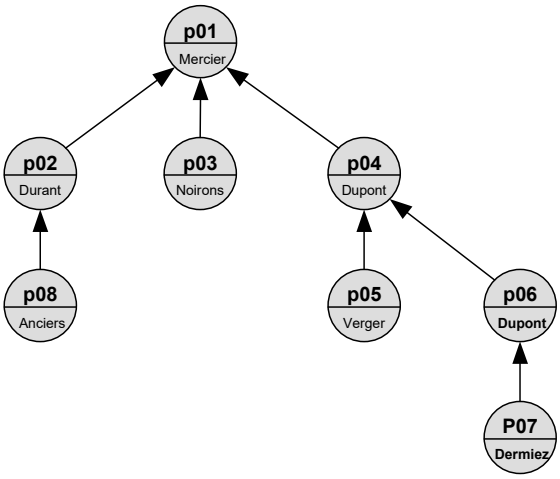



**Figure 19.2 -** The hierarchical structure of the staff

Let us first display all the subordinates of a person. Script 19.6 shows how a recursive **`with`** query can do the job. Its result set for person **p4** is shown in Figure 19.3. For each person processed, the query displays their PID, Name and Manager. In addition, their level in the hierarchy (Lev) is computed: 1 for p04, 2 for the direct subordinates of p04, 3 for the direct subordinates of the direct subordinates of p04, and so on.

```
+-----+------+---------+---------+
| Lev | PID  | Name    | Manager |
+-----+------+---------+---------+
| 1   | p04  | Dupont  | p01     |
| 2   | p05  | Verger  | p04     |
| 2   | p06  | Dupont  | p04     |
| 3   | p07  | Dermiez | p06     |
+-----+------+---------+---------+
```

**Figure 19.3 -** Person p4 and all their subordinates

The initialization query [1] *inserts* in CTE ORG the data of person `$P$` (e.g., person **p04**). The recursive query [2] then adds the data of the subordinates of the last persons inserted. These data are extracted through a join of the current contents of

the CTE with table PERSON. It stops when all the lower level persons have been processed, each providing an empty set of subordinates. If needed, the source person (**p04** here) can be discarded from the result set through a `where` condition in the main query.

```
ask P = PersonID:[!select PID from PERSON];
with recursive ORG(Lev,PID,Name,Manager)
as (select 1,PID,Name,Manager
    from   PERSON where PID = '$P$'                          [1]
      union all
    select O.Lev+1, P.PID, P.Name, P.Manager                 [2]
    from   ORG O, PERSON P
    where  O.PID = P.Manager
    )
select Lev,PID,Name,Manager from ORG;                        [3]
```

**Script 19.6 -** Identifying and displaying data of all the subordinates of a person [STAFF-CTE-subordinates.sql]

Script 19.7 is an extension of Script 19.6. It provides two graphical views of the staff hierarchy. The CTE comprises two new columns:

– **Dist**, which measures the distance in the hierarchy between a person and the root (the source person); Dist = `Lev - 1`, according to the previous query.

– **Path**, which shows the sequence of persons in the hierarchy from the root to each person.

```
ask P = PersonID:[!select PID from PERSON];
with recursive PATHS(Dist,PID,Name,Manager,Path)
as (select 0,PID,Name,Manager,'.'||PID||'.'
    from PERSON where PID = '$P$'
      union all
    select O.Dist+1,P.PID,P.Name,P.Manager,O.Path||P.PID||'.'
    from   PATHS O, PERSON P
    where  O.PID = P.Manager
    )
select substr('------',1,2*Dist)||PID
     ||' ['||Name||']' as Person, Path
from PATHS order by Path;
```

**Script 19.7 -** Showing the hierarchy of the staff [STAFF-CTE-subordinates-paths.sql]

The main query provides these two views (Figure 19.4 for source person **p01**). The first one (column Person) places the persons with an indentation that shows their

relative position *wrt* to their direct manager. The second one (column Path) just writes the values of column Path.

```
+---------------------+-------------------+
| Person              | Path              |
+---------------------+-------------------+
| p01 [Mercier]       | .p01.             |
| --p02 [Durant]      | .p01.p02.         |
| ----p08 [Anciers]   | .p01.p02.p08.     |
| --p03 [Noirons]     | .p01.p03.         |
| --p04 [Dupont]      | .p01.p04.         |
| ----p05 [Verger]    | .p01.p04.p05.     |
| ----p06 [Dupont]    | .p01.p04.p06.     |
| ------p07 [Dermiez] | .p01.p04.p06.p07. |
+---------------------+-------------------+
```

**Figure 19.4 -** Two views of the staff hierarchy

Now, we build a more sophisticated report, in which, for each manager, we count the number of subordinates and we compute their paths (Script 19.8). Here, the CTE is quite simple. It just provides the persons, their subordinates and the paths between them. The main query displays the interleaved result sets of two queries, respectively producing, in addition to the manager id (Head) and subordinate id (Sub),

- the total number of subordinates (Nbr sub)
- the paths between these persons (Path).

The main query discards the initial rows (`Head <> Sub`) and specifies a global `order by` clause which provides the appropriate interleaving (Figure 19.5).

```
with recursive PATHS(Head,Sub,Path)
as (select PID,PID,'.'||PID||'.' from PERSON
       union all
    select O.Head,P.PID,O.Path||P.PID||'.'
    from   PATHS O, PERSON P where O.Sub = P.Manager
    )
select Head,'' as "Nbr sub",Sub,Path
from   PATHS where Head <> Sub
   union all
select Head,count(*) as "Nbr sub",'' as Sub,'' as Path
from   PATHS where Head <> Sub
group by Head
order by Head,Path;
```

**Script 19.8 -** Showing the subordinates of each manager [STAFF-CTE-full-hierarchy.sql]

| Head | Nbr sub | Sub | Path |
|---|---|---|---|
| p01 | 7 | | |
| p01 | | p02 | .p01.p02. |
| p01 | | p08 | .p01.p02.p08. |
| p01 | | p03 | .p01.p03. |
| p01 | | p04 | .p01.p04. |
| p01 | | p05 | .p01.p04.p05. |
| p01 | | p06 | .p01.p04.p06. |
| p01 | | p07 | .p01.p04.p06.p07. |
| p02 | 1 | | |
| p02 | | p08 | .p02.p08. |
| p04 | 3 | | |
| p04 | | p05 | .p04.p05. |
| p04 | | p06 | .p04.p06. |
| p04 | | p07 | .p04.p06.p07. |
| p06 | 1 | | |
| p06 | | p07 | .p06.p07. |

**Figure 19.5 -** A more complete report of the staff hierarchy

We observe that, in these examples, the role of the CTE is to extract raw data while the global query transforms and presents them. Though some processing performed by the global query can be moved to the CTE, this organization tends to make the whole query simpler and easier to write and to understand.

If, instead of displaying the data derived from a CTE, we want to save them, we just have to change the main query for an `insert` query, like in Script 19.9. Similarly, the data of the CTE can also be used to `update` or `delete` data in other tables.

```
create table TMP(PID char(12));
with recursive ORG (PID)
as (select PID from PERSON where PID = '$P$'
      union all
    select P.PID from ORG O, PERSON P where O.PID = P.Manager
    )
insert into TMP select PID from ORG;
```

**Script 19.9 -** Saving the data of a CTE into table TMP [STAFF-CTE-subordinates-insert.sql]

## 19.4 Recursive queries against cyclic data

The scripts of Sections 19.2 and 19.3 work fine because they operate on a data set that really represents a **hierarchy** of persons. More precisely, **nobody in the staff reports to themselves**, neither directly nor indirectly. Let us observe what happens

when this rule is broken. For instance, we change the data so that, from now on, CEO p01 reports to p07:

```
update PERSON set Manager = 'p07' where PID = 'p01';
```

So, p07 reports to p06, who reports to p04, who reports to p01, who reports to ... p07! We have created a circular path, that is, a **circuit** among the staff (Figure 19.6).

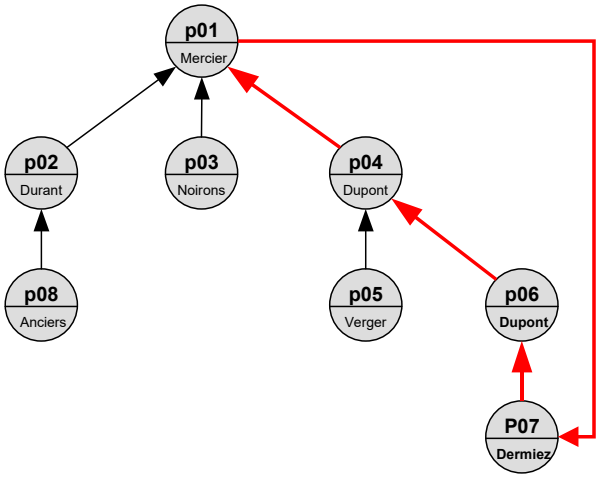



**Figure 19.6 -** Now, the structure of the staff comprises a circuit

Now, we execute Script 19.10, or any other script, with starting point **p01** against this new version of STAFF.

Guess what? **This script never ends**! Indeed, its termination relies on the fact that, at a certain point, all result sets are empty, so that the sequence of executions of the recursive query is limited.

```
with recursive ORG(Lev,PID,Path)
as (select 1,PID,PID from  PERSON where PID = '$P$'
      union all
    select Lev+1,P.PID,Path||'.'||P.PID
    from   ORG O, PERSON P
    where  O.PID = P.Manager
    )
select * from ORG;
```

**Script 19.10 -** A simple (unsafe) script displaying the subordinates of person $P$

When the graph of the hierarchy includes a circuit, then, as soon as one of the person of the circuit is processed, this person, as well as their subordinates, will also be

further processed, again and again, until something (the SQLfast interpreter, the Python runtime, the DBMS, the operating system, the disk) wreaks havoc, possibly killing all the staff by exhaustion.

The termination problem is well known, and has received several solutions. We will describe four of them.

### 19.4.1 Arbitrarily limiting the number of executions

It seems that a staff structured in more than 10 levels is unmanageable and therefore unrealistic. We suggest to limit the executions to a conservative figure of 10 levels (Script 19.11).

```
with recursive ORG(Lev,PID,Path)
as (select 1,'$P$','$P$'
      union all
    select Lev+1,P.PID,Path||'.'||P.PID
    from   ORG O, PERSON P
    where  O.PID = P.Manager
    and    Lev < 10
    )
select * from ORG;
```

**Script 19.11 -** Arbitrarily limiting the number of executions [STAFF-circuits.sql]

Some DBMS offer mechanisms to detect circuits or to limit the number of successive executions of recursive queries. Examples: clause `option (maxrecursion 10)` in SQL Server and clause `cycle` in SQL:1999, available notably in Oracle 11g.

### 19.4.2 Limiting the number of executions to the size of the problem

The largest circuit that can be defined among a set of N persons comprises all these persons. Let us take this number as the limit (Script 19.12).

### 19.4.3 Preventing the query from visiting a person twice

This technique is more general in that it detects abnormal situations when *they are going to occur* and is able to avoid them. When a subordinate is examined, we verify that they have not already been visited as a direct or indirect manager. This condition is quite easy to evaluate for subordinate **S** through the value of Path of the direct manager **M** of **S**. If **S** appears in the Path value of **M**, then they are going to be visited twice, and are skipped. This is what condition

```
(Path not like '.%'||P.PID||'%.')
```

of Script 19.13 translates[7].

As a general rule, relying on the quality of data (e.g., *the data of table PERSON* ***should be*** *acyclic*) to guarantee the safe execution of a program may be bad practice. Writing *defensive* code such as Scripts 19.11 to 19.13 is a much better approach!

```
extract N = select count(*) from PERSON;
with recursive ORG(Lev,PID,Path)
as (select 1,'$P$','$P$'
      union all
    select Lev+1,P.PID,Path||'.'||P.PID
    from   ORG O, PERSON P
    where  O.PID = P.Manager
    and    Lev < $N$
    )
select * from ORG;
```

**Script 19.12 -** Limiting the number of executions to the size of the problem [STAFF-circuits.sql]

```
with recursive ORG(Lev,PID,Path)
as (select 1,'$P$','$P$'
      union all
    select Lev+1,P.PID,Path||'.'||P.PID
    from   ORG O, PERSON P
    where  O.PID = P.Manager
    and    Path not like '%'||P.PID||'%'
    )
select * from ORG;
```

**Script 19.13 -** Preventing the query from visiting a person twice [STAFF-circuits.sql]

### 19.4.4 Discarding duplIcates in the *union* operator

SQL itself can help avoiding visiting a person twice through the `union` operator. Until now, we have added the result sets of the two queries that define the CTE with a `union all` operator. This kind of `union` preserves duplicate rows, so that the result set is no longer a pure *set* but a *multiset*. In acyclic data structures, our queries visit each person only once, so that both variants provide the same result set.

If table PERSON includes circuits and if the CTE adds the result sets through a `union all` operator, then some persons may be visited more than once. On the

7. This condition is quite secure if PID values do not include character '.'. If they do, another delimiter character must be chosen. *Question*: what could we do if no character can be excluded from PID values?

contrary, if we replace the `union all` operator by a simple `union`, then the recursive query will never add a row that has already been inserted in preceeding iterations. Since the last rows inserted never include duplicates, there is no risk to follow circuits. A warning however: duplicate rows have **all** their column values equal, and not only column PID. To avoid them, we must remove columns Lev and Path from the CTE because they will have different values in duplicate rows (Script 19.14).

```
with recursive ORG(PID)
as (select '$P$'
      union
    select P.PID
    from   ORG O, PERSON P
    where  O.PID = P.Manager
    )
select * from ORG;
```

**Script 19.14 -** The **union** operator prevents duplicate rows to be added to the result set

## 19.5 Can recursive queries compute everything?

Difficult question that has no clear answer. Recursive queries can be used to solve many complex problems, sometimes quite unexpected[8], but to what extent?

We can first try to provide a theoretical answer. A *Turing machine* is a mathematical automaton representing a primitive computer[9]. Primitive but able to (somewhat painfully) execute programs and evaluate functions of any complexity. A language is *Turing complete* if it can express all the functions computable by a Turing machine. This is not a big challenge: it must include if and goto statements, sequential read and write statements and a memory to store sequential data. Clearly, SQLfast, as well as all standard programming languages, is Turing complete.

SQL recursive queries implement a subset of *Datalog* predicates. Datalog[10] is a database language based on Prolog (it is little known by practitioners[11] but is a reference among the database scientific community). The interesting point is that *Datalog is not Turing complete*, so that, from the theoretical point of view, SQL recursive queries alone cannot express the solution of all problems.

8. In http://www.sqlite.org/lang_with.html, all the solutions to any Sudoku puzzle are generated by a single SQL query.
9. http://en.wikipedia.org/wiki/Turing_machine
10. http://en.wikipedia.org/wiki/Datalog
11. Though more than a dozen implementations are available, such as pyDatalog for Python.

However, this theoretical development does not help us much. Practically, the question will be coped with through the rules we must follow to write valid recursive queries. These rules are restrictions that ensure that the query will be evaluated efficiently by the DBMS.

### Rule 1: No multiple recursive references

The recursive query may not reference its CTE more than once. Therefore, the queries of Script 19.15 are invalid.

```
with recursive Numbers(N,SN) as
     (select 0,0
        union all
      select N + 1,(select sum(N) from Numbers)
      from Numbers where N < 10)

with recursive Numbers(N,SN) as
     (select 0,0
        union all
      select N1.N + N2.N
      from   Numbers N1, Numbers N2
      where  N1.N + N2.N < 50)
```

**Script 19.15 -** Invalid recursive CTE (the CTE can be referenced only once in the recursive query)

### Rule 2: No mutually recursive CTE

Two recursive queries may not reference each other. Script 19.16,[12] which defines CTE EVEN generating even numbers and CTE ODD generating odd numbers, is particularly elegant, though a bit *snobbish*.

However its form is not allowed in the current implementations of the `with` query. In other words, a CTE can reference itself, but only directly and not through another CTE.

The query can be rewritten as Script 19.16, where CTE EVEN is recursive and does not depend on ODD, while ODD depends on EVEN. This form is valid, and produces the result of Figure 19.7.

12. From an idea suggested by Melton and Simon, *SQL:1999 - Understand Relational Language Components*, Morgan Kaufmann, 2002

```
with recursive
EVEN(N) as
      (select 0
         union all
       select N + 1 from ODD where N < 10),
ODD(N) as
      (select N + 1 from EVEN where N < 10)
select N,'even' as Parity from EVEN
  union
select N,'odd' as Parity from ODD
order by N;
```

**Script 19.16 -** Cross references between two recursive CTE are invalid

```
with recursive
EVEN(N) as
      (select 0
         union all
       select N + 2 from EVEN where N < 8),
ODD(N) as
      (select N + 1 from EVEN where N < 8)
select N,'even' as Parity from EVEN
  union
select N,'odd' as Parity from ODD
order by N;
```

**Script 19.17 -** A CTE can reference a recursive CTE

| N | Parity |
|---|---|
| 0 | even |
| 1 | odd |
| 2 | even |
| 3 | odd |
| 4 | even |
| 5 | odd |
| 6 | even |
| 7 | odd |
| 8 | even |

**Figure 19.7 -** Result of query 19.17

### Rule 3: CTE evaluation must be monotonic

This rule states that no execution of the recursive query may *remove* rows from the current contents of the CTE. Removing existing rows may happen when the recursive part of the CTE includes a negative form (e.g., `except`, `intersect`, `not exists`, etc.) that invalidates some existing rows of the current state of the CTE.

SQL enforces this rule by prohibiting the right part of negative forms to refer to the current CTE.

## 19.6 Tree traversal ordering

When visiting the *nodes of a tree*, for example the persons of the *Staff* application (Figure 19.2), we could want to examine them successively in a certain order, called *traversal ordering*. There are two popular orderings: *breadth first* and *depth first* (Figure 19.8).

*Breadth first* traversal visits nodes level by level, starting from the root of the tree. First the root, then all the children of the root, then all the grandchildren of the root, and so on. So, the persons of the *Staff* application are visited in the following order: first (level 1) p01, then (level 2) p02, p03 and p04, then (level 3) p08, p05 and p06, then (level 4) p07. The order within a level can be arbitrary or according to the direct parent.

*Depth first* traversal visits the children of a node before visiting its siblings. The persons of the *Staff* application are visited in the following order: p01, p02, p08, p03, p04, p05, p06, p07. The order among the children of a node can be arbitrary or according to a definite criterion (PID or Name for instance).

The SQL standard for the **`with`** query includes a **`search`** clause specifying the desired traversal ordering. However, this feature has not been implemented in many DBMS. Fortunately, we can easily enforce both orderings through the standard **`order by`** clause of the main query.

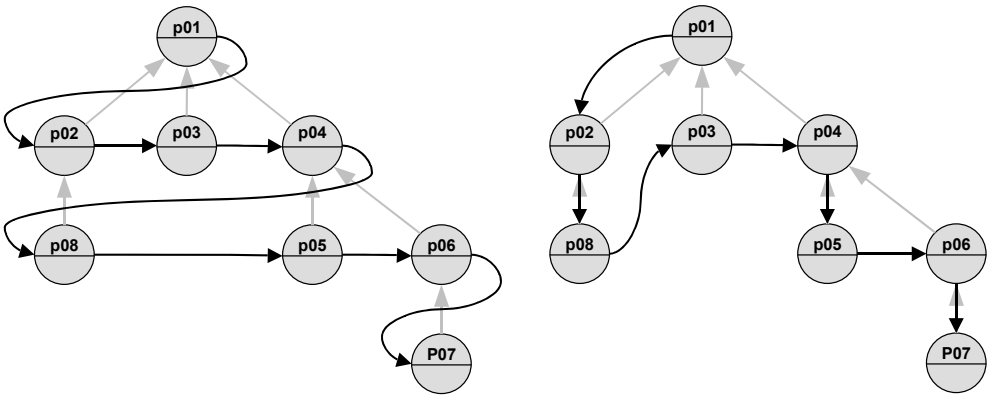



**Figure 19.8 -** Breadth first (left) and depth first traversal (right)

For this, we need a CTE providing the id of the persons (PID), their level number (Lev), their manager (Manager) and their path from the source person (Path) (Script 19.18).

```
with recursive PATHS(Lev,PID,Name,Manager,Path)
as (select 1,PID,Manager,'.'||PID||'.' from PERSON
    where PID = '$P$'
      union all
    select O.Lev+1,P.PID,P.Manager,O.Path||P.PID||'.'
    from   PATHS O, PERSON P
    where  O.PID = P.Manager
    )
select Lev,PID,Manager from PATHS order by Lev,Manager;
```

**Script 19.18 -** Breadth first traversal [STAFF-Traversal-ordering.sql]

The traversal order is defined by the main query as follows:

- **breadth first** (Figure 19.9, left):

  ```
  select Lev,PID,Manager from PATHS order by Lev,Manager
  ```

- **depth first** (Figure 19.9, right):

  ```
  select Lev,PID,Manager from PATHS order by Path
  ```

| Lev | PID | Manager |
|---|---|---|
| 1 | **p01** | -- |
| 2 | **p02** | p01 |
| 2 | **p03** | p01 |
| 2 | **p04** | p01 |
| 3 | **p08** | p02 |
| 3 | **p05** | p04 |
| 3 | **p06** | p04 |
| 4 | **p07** | p06 |

| Lev | PID | Manager |
|---|---|---|
| 1 | **p01** | -- |
| 2 | **p02** | p01 |
| 3 | **p08** | p02 |
| 2 | **p03** | p01 |
| 2 | **p04** | p01 |
| 3 | **p05** | p04 |
| 3 | **p06** | p04 |
| 4 | **p07** | p06 |

**Figure 19.9 -** The two standard traversal orderings for root person p01: breadth first (left) and depth first (right)

## 19.7 SQL recursive triggers

A recursive trigger executes actions that may fire itself. For instance, a trigger for *update on table* ***T*** that performs other updates on table **T** (e.g. its body includes query `update T`) is said to be recursive.

A recursive trigger can be used to repeat similar actions until a definite condition is met, which precisely is the behavior of recursive queries. This is what we will show by rewriting the scripts based on recursive CTE with recursive triggers.

### 19.7.1 Factorial

We still compute couples (Nb,Fact) just like with CTE, but now, we insert them in a **real** table, FACTORIAL (Script 19.19, statement [1]). We initialize it with couple (0,1) by **`insert`** query [4], after which trigger FACT_INS fires [2]. This trigger knows the last row inserted, called **`new`**. So, `new.Nb` is 0 while `new.Fact` is 1. From these values, it computes the next values of Nb and Fact for the factorial of 1 and inserts them as an additional row [3]. This insertion fires the trigger again, leading to the insertion of a third row, defining the factorial of 2, and so on, until the desired number has been processed (new.`Nb < $N$`). Finally, the factorial of this number is displayed by query [5].

This example is shown to illustrate the mechanism of recursive triggers but not as a suggestion to use them to generate series of numbers! On the one hand, the DBMS impose a strict limit to the chain of triggers simultaneously active (1,000 for SQLite but generally much less for other DBMS). On the other hand triggers consumes resources such as memory space and processor time. For such applications, recursive CTE must be preferred.[13]

### 19.7.2 The Staff application

Script 19.20 displays the table of Figure 19.3. It is derived from Script 19.6 as follows.

- The CTE is translated into real table ORG [1].
- The initialization query of the recursive CTE is translated into real **`insert`** query [3].
- The recursive query of the CTE is translated into the body of trigger ORG_INS. This body comprises a single query [2], that builds and inserts the set of rows for the direct subordinates of the row, called new, that fired the trigger. Each such insertion, in its turn, will fire the trigger to insert the next generation of subordinates.
- The main query of the CTE is extracted to form an independent query [4].

Similarly, CTE-based Script 19.7 can be translated into trigger-based procedure of Script 19.21. It also produces the table of Figure 19.4.

Finally, Script 19.22 produces the data of Figure 19.5, as did CTE-based Script 19.8.

13. The generation of a table with the first 1,000 integers takes 0.15 s. with a trigger against 0.015 s. with a recursive CTE on a *typical* machine.

```
ask N = Enter a number (from 0 to 175):;
create table FACTORIAL(Nb integer,Fact integer);                    [1]
create trigger FACT_INS after insert on FACTORIAL                   [2]
for each row
when new.Nb < $N$
begin
  insert into FACTORIAL                                             [3]
  value (new.Nb+1,new.Fact*(new.Nb+1)); end;

insert into FACTORIAL values (0,1);                                 [4]

select Nb,Fact from FACTORIAL where Nb = $N$;                       [5]
```

**Script 19.19 -** Computing the factorial of number N through a recursive trigger

```
create table ORG(Lev integer,PID char(4),
                 Name char(24),Manager char(4));                    [1]
ask P = PersonID:[!select PID from PERSON];
create trigger ORG_INS after insert on ORG
for each row begin
  insert into ORG
  select new.Lev+1,PiD,Name,Manager from PERSON
  where  Manager = new.PiD; end;                                    [2]
insert into ORG select 1,PID,Name,Manager
                from   PERSON where PID = '$P$';                    [3]
select * from ORG order by PID;                                     [4]
```

**Script 19.20 -** Identifying and displaying data of all the subordinates of a person [STAFF-Triggers-subordinates.sql]

Just as CTE scripts, these triggers also fail when the graph of manager/subordinated includes circuits. The best of the three techniques suggested for CTE to avoid infinite firing is that of section 19.4.3 (*Preventing the query from visiting a person twice*). It is implemented in Script 19.23.

```
create table PATHS (Dist integer,PID char(4),Name char(24),
                    Manager char(4),Path char(256));
ask P = PersonID:[!select PID from PERSON];
create trigger PATHS_INS after insert on PATHS
for each row begin
    insert into PATHS
    select new.Dist+1,PID,Name,Manager,
           new.Path||PID||'.'
    from   PERSON where new.PID = Manager; end;
insert into PATHS select 0,PID,Name,Manager,'.'||PID ||'.'
                  from   PERSON  where PID = '$P$';
select substr('------',1,2*Dist)
       ||PID||' ['||Name||']' as Person,Path
from   PATHS order by Path;
```

**Script 19.21 -** Showing the hierarchy of the staff [STAFF-Triggers-subordinates-paths.sql]

```
create table PATHS (Head char(4),Sub char(4),Path char(256));
create trigger PATHS_INS after insert on PATHS
for each row begin
    insert into PATHS
    select new.Head,PID,new.Path||PID||'.'
    from   PERSON
    where  Manager = new.Sub; end;
insert into PATHS select PID,PID,'.'||PID||'.' from PERSON;
select Head,'' as "Nbr sub",Sub,Path
from   PATHS where Head <> Sub
   union
select Head,count(*) as "Nbr sub",'' as Sub,'' as Path
from   PATHS where Head <> Sub
group by Head
order by Head,Path;
```

**Script 19.22 -** Showing the subordinates of each manager [STAFF-Triggers-full-hierarchy.sql]

```
create table PATHS (Lev integer,PID char(4),Path char(256));
ask P = PersonID:[!select PID from PERSON];
create trigger PATHS_INS after insert on PATHS
for each row begin
    insert into PATHS
    select new.Lev+1,PID,new.Path||'.'||PID from PERSON P
    where  Manager = new.PID
    and    new.Path not like '%'||PID||'%'; end;
-- We create a temporary circuit
update PERSON set Manager = 'p07' where PID = 'p01';
insert into PATHS select 1,PID,PID
                  from PERSON where PID = '$P$';
select * from PATHS order by Path;
```

**Script 19.23 -** Preventing the query from visiting a person twice [STAFF-Triggers-circuits.sql]

## 19.8 Loop-based SQLfast implementation of recursive procedures

We may want to design recursive solutions that do not rely on SQL recursive features, for example because they are not available in the underlying DBMS (e.g., MySQL 5 in 2016). In such cases, the SQLfast language can help.

First, many recursive procedures can be simply implemented through a loop.[14] So, in these cases, plain SQLfast scripts using only `if-else-endif`, `while-endwhile` and `for-endfor` statements can do the job.

Secondly, an SQLfast script can call itself, as easily as it would have called any other secondary script. In this sense, SQLfast scripts can be recursive.

In this section, we will explore the loop-based implementation of the recursive procedures we have developed so far.

### 19.8.1 Computing the factorial of a number

Actually, computing the factorial of a number does not require a recursive procedure. A simple loop is the simplest and the most efficient way to get the result. Script 19.24 follows the definition stated in Section 19.2.2. By factoring out the initialization step, a minimal 2-statement procedure can be proposed (Script 19.25).

14. In particular when the (unique) recursive call is the last action of the procedure, a pattern called *tail-recursion*.

```
ask N = Enter a number (from 0 to 175):;
if ($N$ < 1);
   set F = 1;
else;
   set F = 1;
   for Nb = [1,$N$] compute F = $F$ * $Nb$;
endif;
write Factorial of $N$! = $F$;
```

**Script 19.24 -** Computing the factorial of number N through a loop (1) [FACTORIAL-loop.sql]

```
ask N = Enter a number (from 0 to 175):;
set F = 1;
for Nb = [1,$N$] compute F = $F$ * $Nb$;
write Factorial of $N$! = $F$;
```

**Script 19.25 -** Computing the factorial of number N through a loop (2) [FACTORIAL-loop.sql]

### 19.8.2 The *Staff* application

Loop-based translation of recursive CTE or trigger implementations is fairly straightforward, as we will explain for Script 19.26, which extracts the subordinates of a person (see result 19.3 for person **p4**).

As in the trigger-based implementation, we define table ORG, that will collect the result. We initialize it by storing the data of the source person selected [1].

Then, we execute a loop, each iteration of which extracts from PERSON the direct subordinates of the persons inserted during the previous iteration, and stores their data in table ORG [4]. The variable of the loop (`I`) defines the level of the subordinates processed during the previous iteration, which gives us a simple way to identify them (`ORG where O.Lev = $I$`).

The loop is unconstrained (a maximum level of 999 is kind of infinity!) but it is stopped when the previous iteration has not extracted subordinates any more. This is the role of statements [3].

Through a similar translation method of the other CTE and trigger scripts we get:

– Script 19.27, providing two views of the paths between the person selected and all their subordinates.

```
create table ORG (Lev integer,PID char(4),
                  Name char(24),Manager char(4));
ask P = PersonID:[!select PID from PERSON];
insert into ORG select 1,PID,Name,Manager
                from   PERSON where PID = '$P$';                [1]
for I = [1,999];                                                [2]
   extract N = select count(*) from ORG where Lev = $I$;        [3]
   if ($N$ = 0) exit;                                           [3]
   insert into ORG                                              [4]
   select O.Lev+1,P.PID,P.Name,P.Manager
   from   ORG O, PERSON P
   where  P.Manager = O.PID
   and    O.Lev = $I$;                                          [5]
endfor;
select * from ORG order by PID;
```

**Script 19.26 -** Identifying and displaying data of all the subordinates of a person [STAFF-Loop-subordinates.sql]

```
create table PATHS (Dist integer,PID char(4),Name char(24),
                    Manager char(4),Path char(256));
ask P = PersonID:[!select PID from PERSON];
insert into PATHS select 0,PID,Name,Manager,'.'||PID||'.'
                  from   PERSON where PID = '$P$';
for I = [0,999];
   extract N = select count(*) from PATHS where Dist = $I$;
   if ($N$ = 0) exit;
   insert into PATHS
   select O.Dist+1,P.PID,P.Name,P.Manager,O.Path||P.PID||'.'
   from   PATHS O,PERSON P
   where  P.Manager = O.PID and O.Dist = $I$;
endfor;
select substr('------',1,
       2*Dist)||PID||' ['||Name||']' as Person,Path
from   PATHS order by Path;
```

**Script 19.27 -** Showing the hierarchy of the staff [STAFF-Loop-subordinates-paths.sql]

– Script 19.28, showing the subordinates of all the managers of the staff. In this version, we have introduced a level column (Lev), which was not needed in the

former implementation, but that is required here to identify the last persons processed.

– Script 19.29, for non acyclic data (Figure 19.6), based on the third technique.

```
create table PATHS (Lev integer,Head char(4),
                    Sub char(4),Path char(256));
insert into PATHS select 1,PID,PID,'.'||PID||'.' from PERSON;
for I = [1,999];
   extract N = select count(*) from PATHS where Lev = $I$;
   if ($N$ = 0) exit;
   insert into PATHS
   select O.Lev+1,Head,P.PID,O.Path||P.PID||'.'
   from   PATHS O,PERSON P
   where  P.Manager = O.Sub and O.Lev = $I$;
endfor;
select Head,'' as "Nbr sub",Sub,Path
from   PATHS where Head <> Sub
   union
select Head,count(*) as "Nbr sub",'' as Sub,'' as Path
from   PATHS where Head <> Sub
group by Head
order by Head,Path;
```

**Script 19.28 -** Showing the subordinates of each manager [STAFF-Loop-full-hierarchy.sql]

## 19.9 SQLfast recursive scripts

SQLfast has not been designed with recursion in mind. However, when written with some care, an SQLfast script can call itself, either directly (script **A** calls script **A**) or indirectly (script **A** calls script **B**, which calls script **A**) as easily as it would have called any other secondary script[15]. In this sense, SQLfast scripts can be recursive.

The rules according to which SQLfast variables are managed are quite simple:

1. A variable is dynamically created the first time it receives a value; consequently, it is unknown before initialization.
2. Each variable is global. Once it has been created by a main or secondary script, it can be used and modified by any script. The scope of a variable is the current session, so that it can be used to transmit a value from a main script to another

15. The code of precompiled SQLfast scripts is reentrant, so that several simultaneous processes can use this code.

main script. Note however that a variable can be deleted through statement `dropVariable`.

These rules contribute to the simplicity of SQLfast scripts, but, for recursive scripts, variable must be used with a bit more care than in standard programming languages.

```
create table ORG (Lev integer,PID char(4),Path char(256));
ask P = PersonID:[!select PID from PERSON];
insert into ORG select 1,PID,'.'||PID||'.'
                from   PERSON where PID = '$P$';
for I = [1,999];
   extract N = select count(*) from ORG where Lev = $I$;
   if ($N$ = 0) exit;
   insert into ORG
   select O.Lev+1,P.PID,O.Path||P.PID||'.'
   from   ORG O,PERSON P
   where  P.Manager = O.PID and O.Lev = $I$
   and    Path not like '.%'||P.PID||'%.';
endfor;
```

**Script 19.29 -** Preventing the query from visiting a person twice [STAFF-Loop-circuits.sql]

### 19.9.1 Recursive programming in general

The development of the previous three approaches has taught us that a recursive solution comprises three main components[16]:

1. an *initialization step*, executed once,
2. an iterative so-called *recursive procedure* exploiting the result of its previous iteration,
3. a condition that states when iterations must go on or when they must stop.

The most common structure of a recursive solution in standard programming language comprises two procedures.

1. The main procedure initializes the whole process by computing or collecting the initial values, then calls the recursive procedure.

16. My apologies to programming experts for this somewhat biased presentation of recursion (*iteration* is not *recursion*). My concern here is to show that the four programming structures described in this chapter, though very different technically, rely on similar reasoning and that translating from one technique to another one is not particularly tricky, at least for *motivated* users. Whatever this qualifier means!

2. The recursive procedure computes the next set of data, then, if needed, calls itself to exploit this data set to compute the next one, which is then integrated to the solution. Expression *if needed* refers to the condition that decides whether a next call to the recursive procedure will be made or not.

One important mechanism in recursive programming is the way the calling procedure (main or recursive) transmits data to the called recursive procedure, to tell the latter on which data it has to work. In standard programming languages, this transmission generally is performed through the parameter passing mechanism. Since this mechanism does not exist in SQLfast, we must proceed differently. To examine in more detail these subtleties, and contrary to the description of the other three techniques, we will start with the *Staff* application.

### 19.9.2 The *Staff* application - Version 1

We will rewrite Script 19.6 that displays the data of a source person and of all their subordinates. The main script (Script 19.30) opens and closes the database, creates table ORG, gets a person id from the user, inserts the data of this person in table ORG [1] then calls the recursive script to ask it to insert the data of all the subordinates of this person in table ORG [2]. The result is finally written on the output channel (Figure 19.10 for source person p04).

```
openDB STAFF.db;
   create table ORG (Lev integer,PID char(4),
                     Name char(24),Manager char(4));
   ask P = PersonID:[!select PID from PERSON];
   insert into ORG select 1,PID,Name,Manager
                   from   PERSON where PID = '$P$';          [1]
   execSQL STAFF-SQLfast-subordinates-Sub.sql;               [2]
   select * from ORG order by PID;
closeDB
```

**Script 19.30 -** [STAFF-SQLfast-subordinates-Main.sql]

```
+-----+------+---------+---------+
| Lev | PID  | Name    | Manager |
+-----+------+---------+---------+
| 1   | p04  | Dupont  | p01     |
| 2   | p05  | Verger  | p04     |
| 2   | p06  | Dupont  | p04     |
| 3   | p07  | Dermiez | p06     |
+-----+------+---------+---------+
```

**Figure 19.10 -** Person p4 and all their subordinates

The interesting part is recursive script STAFF-SQLfast-subordinates-Sub.sql (Script 19.31). Basically, it comprises two statements only: inserting into ORG the data of

the direct subordinates of the persons just inserted in the calling procedure [3] and calling itself to get the next subordinates [4]. The question is, how does the recursive script knows of which person(s) it must insert the subordinate data?

The answer cannot be simpler: column **Lev** of table ORG indicates the level of the current person in the hierarchy, i.e., 1 for source person p04, 2 for subordinates p05 and p06 of p04, and 3 for subordinates p07 of p06. Therefore, the answer is in the database. To exploit it, we need a means to know the level of the persons of interest, of which the subordinates must be processed. We could imagine to select the persons with the highest value of **Lev**, but this does not work when the lowest subordinates are examined, causing an infinite chain of recursive calling[17].

The means we need is system variable **procLevel**. This variable, which is automatically updated by the SQLfast engine, indicates the calling level of the current script. In a main script, procLevel = 0. In any script called by a main script, procLevel = 1. In general, when script A, with procLevel = n, calls script B, the latter has a procLevel of n + 1.

When main Script 19.30 is run, it has a **procLevel** = 0 and it inserts (so to speak) a person with **Lev** = 1, then calls recursive Script 19.31, which has a procLevel = 1. The rules is now quite obvious: the (supposed) managers of which the script will process the subordinates are those for which,

```
Lev = $procLevel$
```

Recursive Script 19.31 computes in variable **N** the number of these managers [1][18]. If this number is greater than 0, their subordinates, if any, are extracted and their data stored in ORG [2]. Then, the script calls itself to process the potential subordinates of these subordinates [4].

### 19.9.3 Computing the factorial of a number

Though conceptually simpler than the *Staff* scripts, where current data were to be found in the database, computing factorials through an SQLfast script must find another way to transmit data to the recursive script. The main script, called FACTORIAL-SQLfast-Main.sql, starts the process. It does three things (Script 19.32):

1. it asks the user a number and stores it in variable **N**,
2. it calls (recursive) script FACTORIAL-SQLfast-Sub.sql, that computes the factorial of **N** and returns it in variable **F**,
3. it writes the value of **F** on the output channel.

17. Fortunately, such situation will never arise. The SQLfast engine limits the number of cascaded calls to a value stated by parameter **maxProcLevel** in file SQLfast.ini. The default value is 101 but can be set to higher values if needed.
18. Note that there is only one variable called **N** for all the executions of the recursive script. Therefore, when the script calls itself, the value of **N** is replaced by that of the called script, and therefore is lost. Actually this poses no problem since the value of **N** is computed and used just once before the recursive call.

```
extract N = select count(*)
            from ORG where Lev = $procLevel$;                [1]
if ($N$ > 0);
   insert into ORG                                           [2]
   select O.Lev+1,P.PID,P.Name,P.Manager
   from   ORG O,PERSON P
   where  P.Manager = O.PID
   and    O.Lev = $procLevel$;                               [3]
   execSQL STAFF-SQLfast-subordinates-Sub.sql;              [4]
endif;
```

**Script 19.31 -** Extracting subordinate data - Using system variable procLevel [STAFF-SQLfast-subordinates-Sub.sql]

```
ask N = [Enter a number from 0 to 1000] Number:;
execSQL FACTORIAL-SQLfast-Sub.sql;
write Factorial of $N$ is $F$;
```

**Script 19.32 -** Computing the factorial of a number with recursive script [FACTORIAL-SQLfast-Main.sql]

Recursive script FACTORIAL-SQLfast-Sub.sql does the real work (Script 19.33). It distinguishes the case **N** = 0 (it should be `$N$ < 1`, to be sure!), for which it returns **F** = 1. In the other cases, the script decrements the value of **N** by 1, calls itself on this new value, restores the value of **N** (compute `N = $N$ + 1`) then multiplies the value returned by the value of **N**.

```
if ($N$ = 0);
   set F = 1;
else;
   compute N = $N$ - 1;
   execSQL FACTORIAL-SQLfast-Sub.sql;
   compute N = $N$ + 1;
   compute F = $N$ * $F$;
endif;
```

**Script 19.33 -** Computing the factorial of a number with recursive script [FACTORIAL-SQLfast-Sub.sql]

This approach (*modifying* then *restoring*) is fine for some simple cases, where the next value can be easily computed, but may become awkward for more complex problems. So, we will describe a more general technique that works in all cases. To demonstrate it, we will go back to the *Staff* application.

### 19.9.4 The *Staff* application - General approach

The technique we will develop in this section does not rely on the power of SQL, as did Scripts 19.30 and 19.31. In addition, it uses local variables (yes, true local variables, surprising as it seems to be!)

The subordinates of a manager are visited through a `for-endfor` loop. For each of them, a text line is written and the recursive script is called to process their subordinates. We want the result to be as follows, for source person **p01**:

```
Display the hierarchy of p01.
  - p01 is the manager of p02
    - p02 is the manager of p08
  - p01 is the manager of p03
  - p01 is the manager of p04
    - p04 is the manager of p05
    - p04 is the manager of p06
      - p06 is the manager of p07
```

The main script is as expected (Script 19.34). It gets the id of a starting (supposed) manager, writes a text line then calls the recursive script.

```
openDB STAFF.db;
  ask p[0] = PersonID:[!select PID from PERSON];
  write Display the hierarchy of $p[0]$.;
  execSQL STAFF-Sub.sql;
closeDB;
```

**Script 19.34 -** The main script [STAFF-Main.sql]

The id of the current manager at a definite level is stored in a variable that is specific to this level: variable **p[0]** for the starting person at level 0 (see Script 19.34), variable **p[1]** for their direct subordinates at level 1, variable **p[2]** for the direct subordinates of the latter at level 2, and so on. In general, the id of the manager is in variable **p[$procLevel$]**. The main script creates variable **p[0]** while the recursive script creates the series of variables **p[1]**, **p[2]**, etc.

When the recursive script starts one of its executions, it must first access the variable denoting the current manager. The name of this variable is character **p** followed by the value of **procLevel - 1**. The value of **procLevel - 1** is stored in variable **previous**. The id of the manager is in variable **p[$previous$]** and is denoted by expression **$p[$previous$]$**. To clarify (a little bit) the manipulation, this value is stored in variable **mrg[$procLevel$]**. Hence the following sequence in Script 19.35:

```
compute previous = $procLevel$ - 1;
set mgr[$procLevel$] = $p[$previous$]$;
```

The subordinates of this manager are visited through an SQL `for-loop`, which stores the id of each of them in loop variable **p[$procLevel$]**.

```
for p[$procLevel$] = [select PID from PERSON
                      where Manager = '$mgr[$procLevel$]$'];
   <body>
endfor;
```

The body of the loop computes the indentation of the text line, writes the text line (expression `@S$ind$-` produces **ind** spaces followed by a dash character, where **ind** is twice the current level) then calls the script to process the subordinates of manager **p[$procLevel$]**.

```
compute previous = $procLevel$ - 1;
set mgr[$procLevel$] = $p[$previous$]$;
for p[$procLevel$] = [select PID from PERSON
                      where Manager = '$mgr[$procLevel$]$'];
   compute ind = 2*$procLevel$;
   write @Sind$- $mgr[$procLevel$]$ is the manager
         of $p[$procLevel$]$;
   execSQL STAFF-Sub.sql;
endfor;
```

**Script 19.35 -** The recursive script [STAFF-Main.sql]

The recursive script uses two local variables, namely **mgr[$procLevel$]** and **p[$procLevel$]**. Generalization to more local variables, the name of which is indexed by **procLevel**, is straightforward. Variables **previous** and **ind** become useless the first time they are used and need not be preserved. They can be kept global.

### Note on local variables

The term *local variable* requires a bit of explanation. Actually, SQLfast has no such variables, that is, variables created and used in a script but invisible outside because they are destroyed when this script finishes. So, all the variables used in the scripts of this section, that is, p[0], p[1], p[2], mgr[1], mgr[2], etc., are global and known by all the scripts once they have been created. They become *local* due to the discipline we have applied to name and to use them. Standard languages do not require this discipline from the script writer and appear more secure with this respect.

## 19.10 Comparison and conclusions

The choice of the best way to implement a recursive solution of a database problem depends on the techniques provided by the DBMS.

- Whenever possible, **recursive queries** should be used preferably for different reasons: they are the most natural expression of the problem, they impose no limit

to the length of the recursive chain, they generally are fast (particularly with the **`union` all** operator) and they are available in all major DBMS, except MySQL, with (almost) the same syntax.

- **Recursive triggers** are similar to recursive queries but several DBMS do not allow them (MySQL), or impose some drastic constraints on their usage: limited firing chain (all), limited to some patterns (`for each statement` only in Oracle). In addition, some experiments suggest that they are more resource consuming than recursive queries. On the other side, they are much less referenced in the literature, so that their usage may make you look like true SQL experts!
- **Recursive *stored procedures*** are schema components written in one of the procedural languages allowed by the DBMS. Some of them are proprietary (PL/SQL in Oracle, Transact-SQL in SQL Server, MySQL, Firebird, SQL PL in DB2, PL/pgSQL in PostgreSQL) while others are plain programming languages (Java, C, VB, Python, C#). Many DBMS allow procedures to be written in either proprietary or standard languages. Most proprietary languages (and all standard languages) allow recursive procedures to be written, sometimes with restrictions (MySQL). Stored procedures do not exist in SQLite. Besides this approach, ***external* recursive procedures** can be developed in any language that has been given an interface to databases, such as ODBC, JDBC, SQLJ or DB-API (Python). SQLfast scripts belongs to this category. Their main drawback is that they are external to the database contrary to recursive queries and triggers.
- **Loop-based procedures** can be written in all languages for all DBMS. Therefore this technique is the most general. However, because they are a lower level implementation of the recursive procedures, loop-based procedures may appear less natural and more complex than their recursive source. One of their main drawbacks is also that they are not integrated to the database.

In following chapters we will study other applications of recursive database programming:

- tree processing: Kings of France application,
- acyclic graph processing: Bill of Material application,
- general graph processing: The book of which you are the Hero application.

# Chapter 20

# Metadata - Walking on the wild side

**Objective**: This chapter studies the SQLfast *dictionary*, a set of temporary system tables and view containing the description of the schema of the database currently opened. The dictionary contains *data about data*, that is, *metadata*. From these tables, we derive a user-readable version of the database schema and we develop scripts that display the contents of an unknown database.

**Keywords**: metadata, catalog, dictionary, information schema, schema-driven script

**Statements and functions**: createDictionary, deleteDictionary, selectFile, group_concat, group_concat2, item, itemLen, itemIndex

**System variables and parameters**: dictionary

## 20.1 Introduction

When we write queries and database programs we need a precise description of the tables and views that compose this database. In some cases, this documentation is just a little sheet of paper on which we sketch the schema of the database, such as that of Figure 20.1.

The SQLfast environment includes a special function, called *Show schema*, that displays some detail about the schema of a database in a *schema window.* The synthetic view, called **Summary**, is shown in Figure 20.2. If needed, a more

detailed view, as the exact SQL-DDL code, can be displayed by clicking of top left label **Summary**, to replace it by **Full schema**.

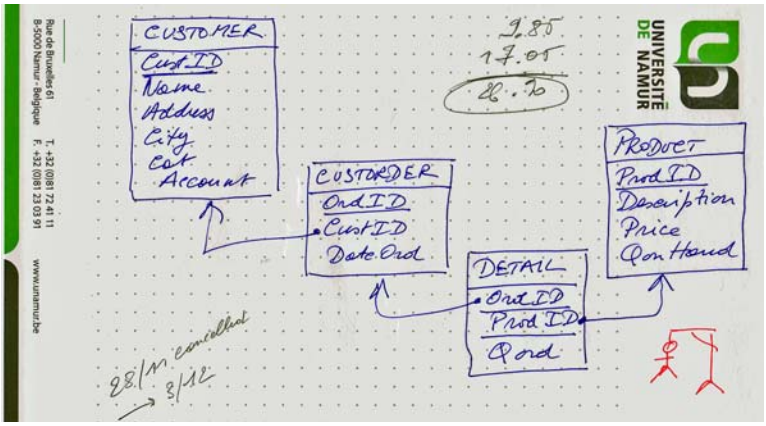



**Figure 20.1 -** The usual way to document a schema

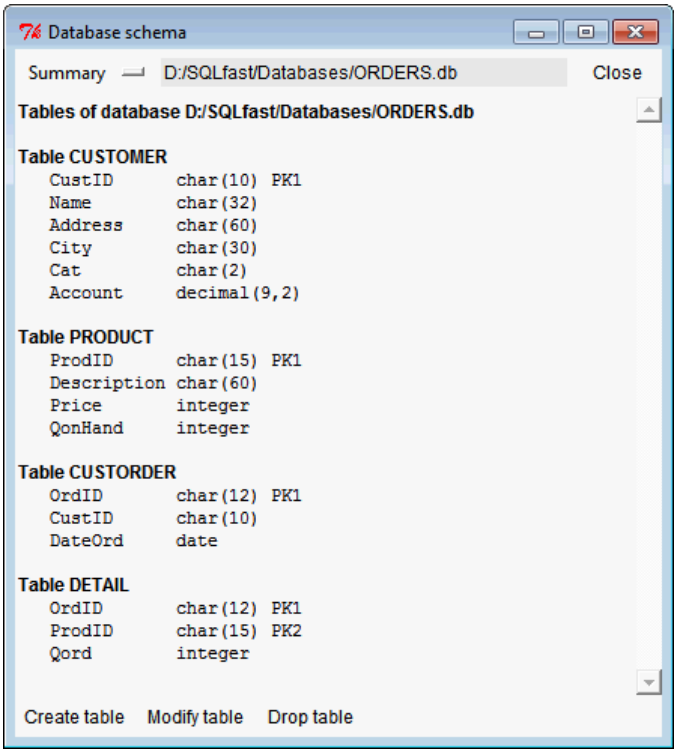



**Figure 20.2 -** Synthetic view of the schema of ORDERS.db

Actually, most database managers also provide such information but quite often according to proprietary interface and format. With many of them, this information is stored in special system tables that, together, are called the ***catalog*** or the (data) ***dictionary***. Such is the case of DB2 and Oracle, the catalog of which comprises more than 100 tables. In others (SQLite for example), this information is provided by specific administration functions that require appropriate manipulations from the developers. The data contained in these tables are not user data but *data about user data*, also called **metadata**.

SQL standards, from SQL:1999, define a specific format for these metadata called **Information Schema**. This schema presents the description of user data as a collection of views that can be consulted through SQL queries like any other table.[1,2]

The SQLfast environment provides a reduced version of the Information Schema comprising four basic tables and a view, named SYS_TABLE, SYS_COLUMN, SYS_KEY, SYS_KEY_COMP and SYS_INTER_TABLE that describe respectively the tables, the columns, the keys of the database and the components of these keys. These tables together form the **Dictionary** of the database.

In this chapter, we will examine the contents of the dictionary and learn to use it to document unknown databases and to produce scripts that are specific to a definite database. In application Schema-based Code Generation, we will find additional, more sophisticated examples of the use of the Dictionary.

The SQLfast dictionary also comprise additional system tables that may be useful to consult or process for advanced applications. These tables contain information of the indexes of the database (SYS_INDEX and SYS_INDEX_COMP), the check constraints (SYS_CHECK), the triggers (SYS_TRIGGER and SYS_TRIGGER_COMP). They will be described in Section 20.3.

## Notes

– Actually, the tables of the dictionary are dynamically created when the database is opened but they disappear when the database is closed. Therefore, they do not physically exist in the database when it is not in use. They are updated in real time when DDL queries are executed. They can be consulted like any user table but cannot be modified by user scripts.[3]

– This material certainly is one of the most exciting in database programming, but it can also be considered a bit complex by novice readers. Indeed, crossing the data/metadata frontier and sometimes mixing these domains in the same SQL queries, require much caution. Clearly, we are *walking on the wild side* of databases!

1. https://en.wikipedia.org/wiki/Information_schema
2. Melton, J., Simon, A., R., *Understanding the new SQL - A complete guide, Chapter 19 Information Schema*, pp. 371-385, Morgan Kaufmann, 1993
3. Well, this is the theory! In SQLfast, nothing prevents a script to update a table, including dictionary tables. This can be useful to use generic column **`Ext`**, that will be described below.

## 20.2 The main SQLfast dictionary tables

The two main system tables of the dictionary provide information on user *tables* and *columns* of the current database. Two additional tables contain information on primary, unique and foreign *keys*. A view derived from these tables shows the dependency graph defined by foreign keys.

**Note**

All the tables have been augmented with generic column **Ext** (for *extension*). It is used to store intermediate results during the creation of the dictionary. It can also be used by the user to serve specific needs when manipulation metadata. We will ignore it in the rest of the chapter.

### 20.2.1 The SYS_TABLE and SYS_COLUMN tables

Table **SYS_TABLE** is very simple. It includes one row per user table or view of the current database. This row gives the unique Id of the table (**TableID**), its name (**TableName**), its type, *base* or *view*, (**TableType**) and whether this table is persistent or temporary (**TablePersist**). The code of Script 20.1 examines the list of the tables of database ORDERS.db, which is shown in Figure 20.3.

```
openDB ORDERS.db;
  createDictionary;
  select * from SYS_TABLE;
closeDB;
```

**Script 20.1 -** Examining the list of tables of database ORDERS.db

| TableID | TableName | TableType | TablePersist | Ext |
|---|---|---|---|---|
| 1 | CUSTOMER | base | Y | -- |
| 2 | PRODUCT | base | Y | -- |
| 3 | CUSTORDER | base | Y | -- |
| 4 | DETAIL | base | Y | -- |

**Figure 20.3 -** Contents of **SYS-TABLE** for the ORDERS.db database

In Script 20.2, we add temporary table CLIENT, persistent view CUST_TOULOUSE and temporary view CUST_PARIS. The contents of SYS_TABLE reflects these additions (Figure 20.4).

```
openDB ORDERS.db;
  createDictionary;
  create temporary table CLIENT(CliID char(4), Name char(24));
  create view CUST_TOULOUSE
         as select * from CUSTOMER where City = 'Toulouse';
  create temporary view CUST_PARIS
         as select * from CUSTOMER where City = 'Paris';
  select * from SYS_TABLE;
closeDB;
```

**Script 20.2 -** Adding table and views to database ORDERS.db

| TableID | TableName | TableType | TablePersist |
|---|---|---|---|
| 1 | CUSTOMER | base | Y |
| 2 | PRODUCT | base | Y |
| 3 | CUSTORDER | base | Y |
| 4 | DETAIL | base | Y |
| 5 | **CUST_TOULOUSE** | view | Y |
| 6 | **CLIENT** | base | N |
| 7 | **CUST_PARIS** | view | N |

**Figure 20.4 - SYS-TABLE** now describes the new objects

Table **SYS_COLUMN** includes one row per column of the current database (Figures 20.5). This row gives the unique Id of the column (**ColID**), the Id of the source table (**TableID**), the sequence number of the column in its table (**ColSeq**), the name of the column (**Cname**), the type of the column (**ColType**), the overall length of the values of the column (**ColLen1**), the number of decimals for *decimal* and *numeric* types (**ColLen2**), whether the values are automatically incremented (**AutoNum**), whether `null` is allowed or not (**ColNull**), the default value (**DefVal**), and whether or not the column appears in the primary key of the table (**PKseq**)[4] and, if ever, its position in the component list. The tables of Figure 20.5 have been obtained by the following queries:

```
select ColID,TableID,ColSeq,ColName,ColType,ColLen1,ColLen2
from   SYS_COLUMN;

select ColID,TableID,ColSeq,ColName,AutoNum,ColNull,DefVal,PKseq
from   SYS_COLUMN;
```

| ColID | TableID | ColSeq | ColName | ColType | ColLen1 | ColLen2 |... |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | CustID | char | 10 | 0 |... |
| 2 | 1 | 2 | Name | char | 32 | 0 |... |
| 3 | 1 | 3 | Address | char | 60 | 0 |... |
| 4 | 1 | 4 | City | char | 30 | 0 |... |
| 5 | 1 | 5 | Cat | char | 2 | 0 |... |

4. This is a redundant information derived from tables `SYS_KEY` and `SYS_KEY_COMP`.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 1 | 6 | Account | decimal | 9 | 2 | ... |
| 7 | 2 | 1 | ProdID | char | 15 | 0 | ... |
| 8 | 2 | 2 | Description | char | 60 | 0 | ... |
| 9 | 2 | 3 | Price | integer | 4 | 0 | ... |
| 10 | 2 | 4 | QonHand | integer | 4 | 0 | ... |
| 11 | 3 | 1 | OrdID | char | 12 | 0 | ... |
| 12 | 3 | 2 | CustID | char | 10 | 0 | ... |
| 13 | 3 | 3 | DateOrd | date | 10 | 0 | ... |
| 14 | 4 | 1 | OrdID | char | 12 | 0 | ... |
| 15 | 4 | 2 | ProdID | char | 15 | 0 | ... |
| 16 | 4 | 3 | Qord | integer | 4 | 0 | ... |

| **ColID** | **TableID** | **ColSeq** | **ColName** | AutoNum | **ColNull** | **DefVal** | **PKseq** |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | CustID | N | N | -- | 1 |
| 2 | 1 | 2 | Name | N | N | -- | 0 |
| 3 | 1 | 3 | Address | N | N | -- | 0 |
| 4 | 1 | 4 | City | N | N | -- | 0 |
| 5 | 1 | 5 | Cat | N | Y | -- | 0 |
| 6 | 1 | 6 | Account | N | N | -- | 0 |
| 7 | 2 | 1 | ProdID | N | N | -- | 1 |
| 8 | 2 | 2 | Description | N | N | -- | 0 |
| 9 | 2 | 3 | Price | N | N | -- | 0 |
| 10 | 2 | 4 | QonHand | N | N | -- | 0 |
| 11 | 3 | 1 | OrdID | N | N | -- | 1 |
| 12 | 3 | 2 | CustID | N | N | -- | 0 |
| 13 | 3 | 3 | DateOrd | N | N | -- | 0 |
| 14 | 4 | 1 | OrdID | N | N | -- | 1 |
| 15 | 4 | 2 | ProdID | N | N | -- | 2 |
| 16 | 4 | 3 | Qord | N | N | -- | 0 |

**Figure 20.5 -** Contents of **SYS-COLUMN** for the ORDERS.db database

## 20.2.2 The SYS_KEY and SYS_KEYCOMP tables

Table **SYS_KEY** describes the primary keys, the unique keys (associated to unique constraint) and the foreign keys of the tables of the database. Each key has a unique Id (**KeyID**), a type (**KeyType**), PK for primary, FK for foreign and UN for unique key. Column **TableID** denotes the parent table of the key and **KeyName** is an artificial unique name. For each foreign key, column **TableTarget** specifies the target table, **KeyTarget** the primary or unique key in this table, **onDelete** the delete mode, **onUpdate** the update mode and **Match** the match clause. Column **KeyNull** specifies whether all the components of the key are *not null* (value **N**), or all the components are *nullable* (value **Y**), or some of the components, but not all, are *not null* (value **?**).[5]

| **KeyID** | **KeyType** | **TableID** | **KeyName** | **TableTarget** | **KeyTarget** | ... |
|---|---|---|---|---|---|---|
| 1 | PK | 1 | T1-PK1 | -- | -- | ... |
| 2 | PK | 2 | T2-PK2 | -- | -- | ... |
| 3 | PK | 3 | T3-PK3 | -- | -- | ... |
| 4 | PK | 4 | T4-PK4 | -- | -- | ... |
| 5 | FK | 3 | T3-FK5-to-T1 | 1 | 1 | ... |
| 6 | FK | 4 | T4-FK6-to-T2 | 2 | 2 | ... |
| 7 | FK | 4 | T4-FK7-to-T3 | 3 | 3 | ... |

5. Redundant information derived from table SYS_KEY_COMP

| KeyID | KeyType | TableID | ... | onDelete | onUpdate | Match | KeyNull |
|---|---|---|---|---|---|---|---|
| 1 | PK | 1 | ... | -- | -- | -- | N |
| 2 | PK | 2 | ... | -- | -- | -- | N |
| 3 | PK | 3 | ... | -- | -- | -- | N |
| 4 | PK | 4 | ... | -- | -- | -- | N |
| 5 | FK | 3 | ... | NO ACTION | CASCADE | NONE | N |
| 6 | FK | 4 | ... | NO ACTION | CASCADE | NONE | N |
| 7 | FK | 4 | ... | CASCADE | CASCADE | NONE | N |

**Figure 20.6 - SYS_KEY**: description of the keys of database ORDERS.db

Table **SYS_KEY_COMP** details the components of each key. Column **KeyCompID** identifies the components, **KeyID** specifies the parent key, **CompSeq** the position of the components within the key, **ColID** identifies the column and **CompTarget** specifies, for a foreign key component, the associated component in the target key. Columns **ColName** and **ColNull** are derived from the properties of column **ColID**, and therefore redundant.

| KeyCompID | KeyID | CompSeq | ColID | ColName | ColNull | CompTarget |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | CustID | N | -- |
| 2 | 2 | 1 | 7 | ProdID | N | -- |
| 3 | 3 | 1 | 11 | OrdID | N | -- |
| 4 | 4 | 1 | 14 | OrdID | N | -- |
| 5 | 4 | 2 | 15 | ProdID | N | -- |
| 6 | 5 | 1 | 12 | CustID | N | 1 |
| 7 | 6 | 1 | 15 | ProdID | N | 2 |
| 8 | 7 | 1 | 14 | OrdID | N | 3 |

**Figure 20.7 - SYS_KEY_COMP**: description of the components of the keys of database ORDERS.db

### 20.2.3 The SYS_INTER_TABLE view

If we extract the couples (KeyID, KeyTarget) from table SYS_KEY, we build the edges of the directed graph formed by the foreign keys of the database. View SYS_INTER_TABLE of the dictionary extends these couples by the Id and the name of the table of the foreign key and the Id and the name of the table of the target primary/unique key. The data of this view are shown in Figure 20.8.

| SourceTable | SourceTableID | SourceKey | TargetKey | TargetTableID | TargetTable |
|---|---|---|---|---|---|
| CUSTORDER | 3 | 5 | 1 | 1 | CUSTOMER |
| DETAIL | 4 | 6 | 2 | 2 | PRODUCT |
| DETAIL | 4 | 7 | 3 | 3 | CUSTORDER |

**Figure 20.8 -** SYS_INTER_TABLE: the graph defined by the foreign keys of database ORDERS.db

## 20.3 The additional system tables

The SQLfast dictionary comprises five additional system tables that describe the check constraints, the indexes and the triggers.

### SYS_CHECK table

Let us suppose that column Qord of table DETAIL has been declared as follows:

```
Qord integer not null constraint CHECK_DET_1 check (Qord > 0)
```

**Script 20.3 -** A check constraint on column Qord

This constraint will be documented in system table **SYS_CHECK** (Figure 20.9).

| CheckID | TableID | ConstrName | Condition |
|---|---|---|---|
| 1 | 4 | CHECK_DET_1 | Qord > 0 |

**Figure 20.9 -** Table SYS_CHECK

Column **CheckID** identifies the constraint in the schema. Column **TableID** denotes the parent table. Column **ConstrName** shows the name assigned to the constraint and column **Condition** contains the expression of the constraint.

### SYS_INDEX and SYS_INDEX_COMP tables

These system tables describe the indexes of the schema and their components.

In table **SYS_INDEX** (Figure 20.10) column **IndexID** assigns to the index a unique identifier in the schema. Column **TableID** gives the name of the parent table, column **IndexSeq** the sequence number of the index in its table and **IndexName** the name of the index. Column **IsUnique** indicates if the index supports a primary or unique key. If the index is partial (that is, indexes a subset only of the rows), column **Partial** provides the condition.

| IndexID | TableID | IndexSeq | IndexName |...
|---|---|---|---|---
| 1 | 1 | 1 | sqlite_autoindex_CUSTOMER_1 |...
| 2 | 2 | 1 | sqlite_autoindex_PRODUCT_1 |...
| 3 | 3 | 1 | sqlite_autoindex_CUSTORDER_1 |...
| 4 | 4 | 1 | sqlite_autoindex_DETAIL_1 |...

| IndexID | TableID | IndexSeq | ... | IsUnique | Partial |
|---|---|---|---|---|---|
| 1 | 1 | 1 | ... | PK | -- |
| 2 | 2 | 1 | ... | PK | -- |

```
| 3         | 3         | 1          | ... | PK         | --         |
| 4         | 4         | 1          | ... | PK         | --         |
+-----------+-----------+------------+-...-+------------+------------+
```

**Figure 20.10 -** Table SYS_INDEX

In system table **SYS_INDEX_COMP**, each row describes a component (a column) of an index (Figure 20.11). Column **IndexCompID** assigns to the component a unique identifier in the schema. Column **IndexID** references the parent index and **CompSeq** is the sequence number of the component in its index. Column **ColID** specifies the column of the component and **ColName** its name.

```
+-------------+---------+---------+-------+---------+
| IndexCompID | IndexID | CompSeq | ColID | ColName |
+-------------+---------+---------+-------+---------+
| 1           | 1       | 1       | 1     | CustID  |
| 2           | 2       | 1       | 7     | ProdID  |
| 3           | 3       | 1       | 11    | OrdID   |
| 4           | 4       | 1       | 14    | OrdID   |
| 5           | 4       | 2       | 15    | ProdID  |
+-------------+---------+---------+-------+---------+
```

**Figure 20.11 -** Table SYS_INDEX_COMP

## SYS_TRIGGER and SYS_TRIGGER_COMP tables

These system tables describe the triggers of the schema and the statements of their body. The body of (fictitious) trigger of Script 20.4 comprises two statements.

System table SYS_TRIGGER provides the information of each trigger.

```
create trigger TRG_CORD_UPD
after update of ItemID,Qty on CUSTORDER
for each row
when new.Price > old.Price
begin
   insert into ARCHIVE values(timestamp,new.OrdID,old.ItemID);
   update ITEM set QonHand = QonHand = new.Qty,
                   Qord = Qord + new.Qord
   where ItemID = new.ItemID;
end;
```

**Script 20.4 -** A typical trigger

```
+--------+---------+-----------+-------------+----------+-----------+---
| TrigID | TableID |TableName  |TrigName     | TrigWhen | TrigEvent |...
+--------+---------+-----------+-------------+----------+-----------+---
| 7      | 3       | CUSTORDER |TRG_CORD_UPD | after    | update    |...
+--------+---------+-----------+-------------+----------+-----------+---

+--------+-...-+------------+-----------+-----------------------+
| TrigID | ... | TrigCol    | TrigScope | TrigCond              |
+--------+-...-+------------+-----------+-----------------------+
```

```
| 7      | ... | ItemID,Qty |row         | new.Price > old.Price |
+--------+...-+------------+-----------+-----------------------+
```

**Figure 20.12 -** Table SYS_TRIGGER

Table SYS_TRIGGER_COMP describes the statements of the body of each trigger.In the current version of SQLfast, which relies on SQLite 3, the body of a trigger is made up of a *sequence* of queries. In Case study **Active databases**, we show that more complex algorithmic constructs, such as variables and conditional statements can be losslessly transformed into a pure sequence of SQL queries.[6]

```
+------------+--------+--------------+---------+-----------+----------+--
| TrigStatID | TrigID | TrigName     | StatSeq | StatTable | StatOper |..
+------------+--------+--------------+---------+-----------+----------+--
| 1          | 1      | TRG_CORD_UPD | 1       | CUSTORDER | update   |..
| 2          | 1      | TRG_CORD_UPD | 2       | CUSTORDER | update   |..
+------------+--------+--------------+---------+-----------+----------+--

+------------+...-+------------+--------------------------------------+
| TrigStatID | ... | StatCol    | StatExpr                             |
+------------+...-+------------+--------------------------------------+
| 1          | ... |            | insert into ARCHIVE [..] ,old.ItemID) |
| 2          | ... | ItemID,Qty | update ITEM set [..] Qord + new.Qord  |
+------------+...-+------------+--------------------------------------+
```

**Figure 20.13 -** Table SYS_TRIGGER_COMP

## 20.4 How to examine the SQLfast dictionary?

The system tables are transient. They disappear when the dictionary is deleted (`deleteDictionary`) or when the current database is closed (`closeDB`). Menu function **Database > Show DB data** does not show the system tables, since the current database has been closed, or, if it is not closed, temporary tables are not visible from outside.

Of course, displaying the content of these tables can be performed through standard select statements:

```
select * from SYS_COLUMN;
```

Actually, the most interesting technique consists in creating a persistent clone of the system tables associated with the current database. Script 20.5 shows a simple way to create this persistent version of the dictionary. The later is named 'Dictionary-of-' + <current database name>. The current dictionary is deleted and a new one is created and attached to the current database. Then, for each system table of the current data-

6. Following the SQLite limitations, we have not described stored procedures either. If needed, they can be simulated by adhoc triggers or UDF.

base, a new table is created in the persistent dictionary with the same structure and content.[7] The dictionary is then detached.

From now on, it can be consulted, augmented and processed independently from its source. In Chapter 23 (*Aid to SQLfast script development*), this persistent dictionary is used to identify potential circuits in the trigger graph of a database.

```
set current_db = RETAIL.db;
set dic-db = $dbDirectory$/Dictionary-of-$current-db$;

deleteDB $dic-db$;

openDB $current_db$;
createDictionary;

generic attach '$dic-db$' as dic;

for sys = [item(,) SYS_TABLE,SYS_COLUMN,...,SYS_TRIGGER_COMP]
   create table dic.$sys$ as select * from $sys$;

commitDB;

generic detach database 'dic';
```

**Script 20.5 -** A procedure creating a persistent dictionary of a database. An extended version of this procedure is available in file SQLfast-Tutorials/Chapter-20/Create-persistent-dictionary.sql.

## 20.5 Dictionary or no dictionary?

The dictionary is a useful resource for skilled script writers, as we will learn in the next sections, but the previous chapters have also shown us that one can write quite valuable scripts without relying on it. Therefore, it would be nice to decide whether SQLfast creates, maintains and exploits a dictionary or simply ignores it. There are two ways to control dictionaries.

1. Initialization file **SQLfast.ini** can include a specific parameter stating whether SQLfast automatically creates and maintains a dictionary whenever a database is opened:

   ```
   dictionary = on
   ```

   or, on the contrary, if it opens databases without creating their directories:

   ```
   dictionary = off
   ```

2. Whatever the status of parameter **dictionary**, a script can explicitly ask for the creation and deletion of the dictionary of the database currently opened:

7. Actually, some additional redundant tables are created, that help users or developers to read system tables. In particular, wherever a table Id (TableID) appears, the name of the table is added.

- statement **`createDictionary`** creates the dictionary of the current database; if changes are applied to the database schema (`create`, `drop`, `alter`), the dictionary is automatically updated.
- statement **`deleteDictionary`** statement drops the dictionary of the current database.

## 20.6 Examining the schema of an unknown database

As a first application of the metadata of the dictionary, let us display the schema of the tables of an unknown database (ORDERS.db for example!) in a layout similar to that of the *Schema window* of SQLfast, as shown in Figure 20.2.

This display comprises two sets of lines, the first one provides the name of each table and the second one describes the columns of these tables. These sets are built through specific `select` queries, the result sets of which are merged with a `union` operator. These queries must select the same number of values, some of them being empty in the first result set and the others being empty in the second one, and conversely.

The tricky aspect of the whole query is to interleave the lines of the two result sets in such a way that each table line is directly followed by its column lines. The solution lies in the `order by` clause, which sorts the lines first on TableID (a column common to both result sets), then on ColSeq. Since the latter does not exist in table SYS_TABLE, it is assigned value 0 so that it appears first in the sequence of lines of its table.

These principles are coded in Script 20.6, which produces the display of Figure 20.14.

```
openDB ORDERS.db;
  createDictionary;
  select TableID,0 as ColSeq,TableName,'' as ColName
  from SYS_TABLE
     union
  select TableID,ColSeq,'' as TableName,ColName
  from SYS_COLUMN
  order by TableID,ColSeq;
closeDB;
```

**Script 20.6 -** Displaying the schema of database ORDERS.db - First version

| TableID | ColSeq | TableName | ColName |
|---|---|---|---|
| 1 | 0 | CUSTOMER | |
| 1 | 1 | | CustID |
| 1 | 2 | | Name |
| 1 | 3 | | Address |

| | | | |
|---|---|---|---|
| 1 | 4 | | City |
| 1 | 5 | | Cat |
| 1 | 6 | | Account |
| 2 | 0 | PRODUCT | |
| 2 | 1 | | ProdID |
| 2 | 2 | | Description |
| 2 | 3 | | Price |
| 2 | 4 | | QonHand |
| 3 | 0 | CUSTORDER | |
| 3 | 1 | | OrdID |
| 3 | 2 | | CustID |
| 3 | 3 | | DateOrd |
| 4 | 0 | DETAIL | |
| 4 | 1 | | OrdID |
| 4 | 2 | | ProdID |
| 4 | 3 | | Qord |

**Figure 20.14 -** Synthetic view of the schema of database ORDERS.db - First version

The first two columns are used to interleave the lines of the result sets but should not be displayed. To get rid of them, we encapsulate the union of `select` queries into an external `from` clause, as shown in Script 20.7, which, in addition, shows the type of the columns (Figure 20.15).

```
select TableName,ColName
from (select TableName,'' as ColName,'' as ColType
      from SYS_TABLE
        union
      select '' as TableName,ColName,ColType
      from SYS_COLUMN
     )
order by TableID,ColSeq;
```

**Script 20.7 -** Removing technical columns and adding column types - Second version

| TableName | ColName | ColType |
|---|---|---|
| CUSTOMER | | |
| | CustID | char |
| | Name | char |
| | Address | char |
| | City | char |
| | Cat | char |
| | Account | decimal |
| PRODUCT | | |
| | ProdID | char |
| | Description | char |
| | Price | integer |
| | QonHand | integer |
| CUSTORDER | | |
| | OrdID | char |

```
|            | CustID      | char    |
|            | DateOrd     | date    |
| DETAIL     |             |         |
|            | OrdID       | char    |
|            | ProdID      | char    |
|            | Qord        | integer |
+------------+-------------+---------+
```

**Figure 20.15 -** Synthetic view of the schema of database ORDERS.db - Second version

In the final version, we add length information for char, varchar, decimal and numeric column types. In addition, we simplify the layout of the display by removing the header line and the horizontal and vertical bars. We leave the select-align parameter set to Y to keep the tabular structure.

Applied to the ORDERS.db database, Script 20.8 produces the result of Figure 20.16, which is a fairly good imitation of the Schema window! Adding primary key indicators is quite easy but is left as an exercise.

```
set bar11,bar12,bar13,bar14 = ,,,;
set bar21,bar22,bar23,bar24 = ,,,;
set bar31,bar32,bar33,bar34 = ,,,;
set head1,head2,head3,head4 = ,,,;
set data1,data2,data3,data4 = ,@v,@s,@n;

select TableName,ColName,ColType
from (select TableID,0 as ColSeq,TableName,
             '' as ColName,'' as ColType
      from SYS_TABLE
        union
      select TableID,ColSeq,'' as TableName,ColName,
      case when ColType in ('numeric','decimal')
           then ColType||'('||cast(ColLen1 as char)
                          ||','||cast(ColLen2 as char)||')'
           when ColType in ('char','varchar')
           then ColType||'('||cast(ColLen1 as char)||')'
           else ColType
      end as ColType
      from SYS_COLUMN)
order by TableID,ColSeq;

execSQL UTIL-SELECT-parameters-standard-1.sql;
```

**Script 20.8 -** Displaying the schema of database ORDERS.db - Final version

```
CUSTOMER
          CustID      char(10)
          Name        char(32)
          Address     char(60)
          City        char(30)
```

```
            Cat          char(2)
            Account      decimal(9,2)
    PRODUCT
            ProdID       char(15)
            Description  char(60)
            Price        integer
            QonHand      integer
    CUSTORDER
            OrdID        char(12)
            CustID       char(10)
            DateOrd      date
    DETAIL
            OrdID        char(12)
            ProdID       char(15)
            Qord         integer
```

**Figure 20.16 -** Synthetic view of the schema of database ORDERS.db - Final version

## 20.7 From metadata to data

Let us push a bit further the idea to analyze an unknown database by examining its contents. At first, we just know the name of the database, which allows us to select it in the Databases directory[8]:

```
selectFile db = $dbDirectory$,Database,db;
```

Then, we open it and we create its dictionary:

```
openDB $db$;
createDictionary;
```

Now, we write a loop that extracts the names of the tables and displays their contents (Script 20.9).

```
selectFile db = $dbDirectory$,Database,db;
  if ('$DIALOGbutton$' = 'Cancel') stop;
openDB $db$;
  createDictionary;
  for tName = [select TableName from SYS_TABLE]
      select * from $tName$;
closeDB;
```

**Script 20.9 -** Examining the contents of all the tables of a database

8. The **selectFile** statement specifies the source directory ($dbDirectory$), the name of file category (Database) and its extension (db).

To be really useful, this procedure must be refined. First, displaying all the rows of a table or of a view may become painful for large result sets! Limiting the list to, say, 20 rows seems more realistic:

```
select * from $tName$ #[1,20];
```

Then, we let the users select the table they want to examine (Figure 20.17):

```
ask tName = Table[!select TableName from SYS_TABLE];
  if ('$DIALOGbutton$' = 'Cancel') stop;
select * from $tName$ #[1,20];
```

Script 20.10 assembles these fragments into a complete script.

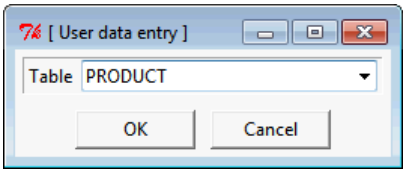



**Figure 20.17 -** Selecting the name of a table to examine

```
selectFile db = $dbDirectory$,Database,db;
  if ('$DIALOGbutton$' = 'Cancel') stop;
openDB $db$;
  createDictionary;
  while (True);
    ask tName = Table[!select TableName from SYS_TABLE];
      if ('$DIALOGbutton$' = 'Cancel') exit;
    select * from $tName$ #[1,20];
  endwhile
closeDB;
```

**Script 20.10 -** Examining the content of a database - Version 1

Though this script seems quite fine for the goal we had in mind, just changing the table selection box from **ask** to **selectOne** (Figure 20.18) allows us to illustrate another use of the metadata.

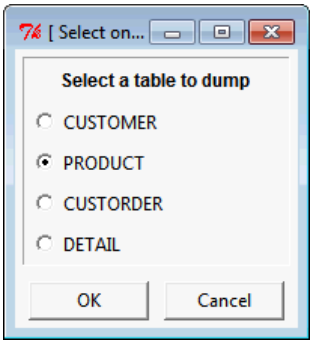



**Figure 20.18 -** Another way to select the table to examine

The new challenge here is to automatically build the list of table names for this statement:

```
selectOne table = [/bSelect a table to dump]
                  CUSTOMER|PRODUCT|CUSTORDER|DETAIL;
```

For this, we will use SQL aggregate function **group_concat**, the effect of which is to concatenate strings from the members of each group of rows, as in the following query, which associates with each city the list of its customers:

```
select City,group_concat(CustID) as Customers
from   CUSTOMER group by City;
```

```
+-----------+--------------------------+
| City      | Customers                |
+-----------+--------------------------+
| Bruxelles | F400                     |
| Genève    | B332                     |
| Lille     | K111                     |
| Namur     | C123,S127,B062,L422      |
| Paris     | S712                     |
| Poitiers  | B112,F010,C400           |
| Toulouse  | B512,C003,K729,F011,D063 |
+-----------+--------------------------+
```

The default separator is the comma. If another separator is preferred, it must be specified explicitly:

```
select City,group_concat(CustID,';') as Customers
from   CUSTOMER group by City;
```

So, going back to the list of table names we have to build, function `group_concat`, applied to table SYS_TABLE, will do the job:

```
extract tList = select group_concat(TableName,'|')
```

```
                    from   SYS_TABLE;
    selectOne table = [/bSelect a table to dump] $tList$;
```

Statement `selectOne` returns a button number, that we have to transform into a table name. Here too, variable **tList** will help us. Its value is a list of table names separated with symbol `'|'`. Function `item(L,i,sep)` returns the $i^{th}$ item of list `L` in which items are separated with symbol `sep`. The name of the table selected will be obtained by expression:

```
item('$tList$',$table$,'|')
```

These principles have been applied in Script 20.11.

```
. . .
extract tList = select group_concat(TableName,'|')
                from   SYS_TABLE;
while (True);
  selectOne table = [/bSelect a table to dump] $tList$;
    if ('$DIALOGbutton$' = 'Cancel') exit;
  compute tName = item('$tList$',$table$,'|');
  select * from $tName$ #[1,20];
endwhile
. . .
```

**Script 20.11 -** Examining the content of a database - Version 2

## 20.8 Complement: SQL *group_concat* function

This function is not standard and both its name and its structure vary among the main DBMS. The name group_concat used in this section is that of MySQL and SQLite.

One of the most complete form of **`group_concat`** is that of MySQL. Besides the expression to concatenate and the separator, it also specifies whether the values are distinct and in which order they appear in the list:

```
group_concat(distinct CustID order by CustID separator ';')
```

The current SQLite version (2020) is incomplete: one can specify, in addition to the expression, either modifier **`distinct`** (`distinct CustID`) or the separator (`CustID,';'`), but not both.

Considering its importance, notably in text generation from the database, a new function, group_concat2, has been implemented in SQLfast to compensate for the weakness of SQLite function:

```
group_concat2(expr,distinct,order,direction,separator)
```

– **expr** is the expression that produces the values to concatenate; *null* values are ignored,

– **distinct** tells whether duplicate values must be preserved (0) or discarded (1),

– **order** is the `order by` key as an SQL expression,

– **direction** specifies whether the order is ascending (1) or descending (2),

– **separator** is the expression the value of which must be used as the value separator; a *null* values is considered an empty string.

For example, the MySQL expression above will be translated as follows:

```
group_concat2(CustID,0,CustID,1,';')
```

**group_concat2** is equivalent to the MySQL function.

## 20.9 Summary

SQLfast provides a **dictionary**, a set of system tables containing the description of the schema of the current database. This description is called *metadata* or *data about user data*. These tables can be queried like user tables. These tables are:

- table **SYS_TABLE** gives the main properties of each table.
- table **SYS_COLUMN** gives the main properties of each column.
- table **SYS_KEY** describes the primary, unique and foreign keys.
- table **SYS_KEY_COMP** describes the components of each key.
- view **SYS_INTER_TABLE** shows the graph defined by the foreign keys.
- table **SYS_CHECK** describes the check constraints.
- table **SYS_TRIGGER** describes the clauses of the header of triggers.
- table **SYS_TRIGGER_COMP** describes the statements of the body of triggers
- The contents of the dictionary tables can be used to examine the schema and the data of an unknown database.
- The dictionary is created automatically when a database is opened if parameter **Dictionary** is set to **on** in file SQLfast.ini.
- Whatever the value of parameter **Dictionary**, the dictionary can be created by statement **createDictionary**. It can be deleted by statement **deleteDictionary**.

- A independent, persistent version of the system tables of database *B* can be created as database Dictionary-of-*B*. It can be generated through script Create-persistent-dictionary.sql.
- A list of values can be built as a character string comprising the concatenation of these values, separated by an appropriate symbol (or, more generally, a character string).
- Lists of values extracted from the database can be computed by SQL aggregate function **group_concat**. **group_concat2**, the SQLfast implementation of this function provides parameters for uniqueness of values, value ordering and separator.
- A specific item can be extracted from a list by function **item(L,i,sep)**. Its arguments are list **L**, index **i** (from 1 to the size of the list) and separator symbol **sep**. Other list processing functions:
  - **itemLen(L,sep)**: number of elements in L.
  - **itemIndex(L,e,sep)**: index of element **e** in L. 0 if missing
  - **itemS(L,i)**: same as **item** when L uses standard separator ';'
  - **itemLenS(L)**: same as **itemLen** when L uses standard separator ';'
  - **itemIndexS(L,e)**: same as **itemIndex** when L uses standard separator ';'.

  Other fonctions perform *set* and *multiset* operations between two lists.

# Chapter 21

# SQLdraw

***Writing in progress***

## 21.1 Vector graphics in SQLfast

The SQLfast environment includes a graphical engine that draws vector graphics defined in a **SQLdraw** script, stored in a file or in a variable. An SQLdraw drawing is made up of shapes of different types: collections of points, polylines, polygons, rectangles, circles, ellipses, bitmaps and texts. An SQLdraw drawing can be still[1] or animated[2], just like **gif** files.

## 21.2 The SQLdraw scripting language

The SQLdraw commands control the behaviour of the SQLfast vector graphic box (statement `showDrawing`). They are stored in a ***.draw** text file. They allow the creation of the drawing area, the display of shapes, their deletion, the display of messages and specifying a delay.

1. Se for instance the *Kings of France* case study, available in the SQLfast web site.
2. Two representative examples: *Conway's Game of Life* and *Agent-based modeling and simulation* case studies, also available in the SQLfast web site.

A shape is a named basic graphic form: image, polyline, polygon, series of points, rectangle, circle. Once displayed, a shape can be deleted. A *shape sequence* is a suite of shapes that are displayed in the drawing area and that cannot be interrupted through the Pause button. A sequence is terminated by a `step` command or by the end of the file. When the SQLdraw box opens, it shows the first sequence. The next sequences can be displayed through buttons Next (only the next sequence) or Play (all the next sequences).

A color can be specified by its RGB components (`#RRGGBB`, where `RR`, `GG`, `BB` are hexa bytes) or by the poetic symbolic name of one of the 754 standard colors of *tkinter* (just google Tkinter colors in RGB).

**`area`** `600,480,"white"`

Specifies the drawing area*:* width, height and background color

**`image`** `"photo1",grid.gif`

Specifies a *.gif file that will be loaded in the drawing area. *Currently disabled.*

**`polyline`** `"Lines-08",1,"grey"`

Header of a polyline (series of connected points): label, line width (in points), line color. Followed by a list of unlabelled points.

**`polyline`** `"Lines-08",1,"grey","yellow"`

Header of a polygon (series of connected points where the starting point is the ending point): label, line width (in points), line color, fill color. Followed by a list of unlabelled points.

**`points`** `"Cities",1,"black","black",15`

Header of a series of unconnected points of the same type: line width, line color, fill color, point type. Followed by a list of labelled points.

**`rectangle`** `"crop",x1,y1,x2,y2,x3,y3,lineW,lineCol,fillCol`

Draw rectangle `x1,y1,x2,y2,x3,y3`, with specified line width, line color and fill color (empty if none).

**`circle`** `"round1",x,y,r,lineW,lineCol,fillCol`

Draw a circle with center `x,y` and radius `r` and with specified line width, line color and fill color (empty if none).

**`wait`** `100`

Delay of 100 ms.

**step**

Specifies a new shape sequence. A script comprising this command describes an animated drawing.

**delete** name

`delete last`: erase the last shape from the graphical area.

`delete all`: erase all the shapes from the graphical area.

`delete Cities`: erase the shape named Cities from the graphical area.

`451,984`

Unlabelled point. Origin `0,0` is the top left corner.

`451,984,"London"`

Labelled point. Origin `0,0` is the top left corner.

**display** message

Write a `message` in the message field of the graphic box.

Figure 21.1 shows the point types available and their code.

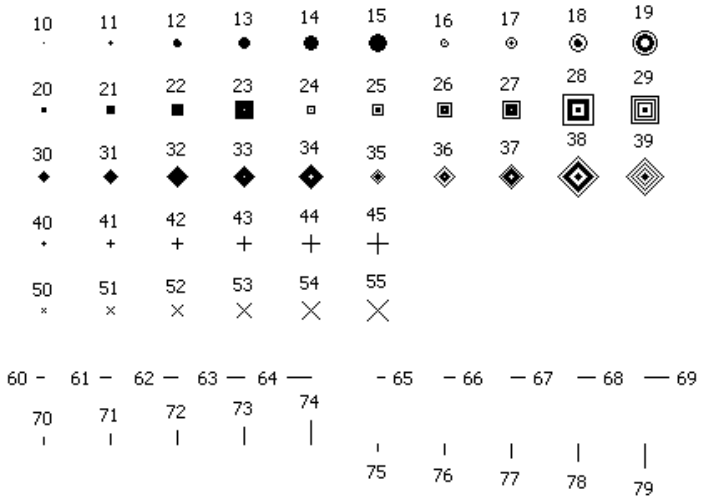



**Figure 21.1 -** The SQLdraw point types

## 21.3 Displaying an SQLdraw script

An *SQLdraw* script file can be executed through the **`showDrawing`** command, as described in Chapter 13:

```
showDrawing Files/LIFEGAME-Glider.draw
            = [/x300/y300/bLIFE Game];

fileToVariable glider = Files/LIFEGAME-Glider.draw;
showDrawing glider = [/b/x395/y245AGENT simulation: ...];
```

Alternatively, a script file can be executed from the SQLfast main window by the menu item `General > Play SQLdraw`.

Both open a graphic box (Figure 21.2).

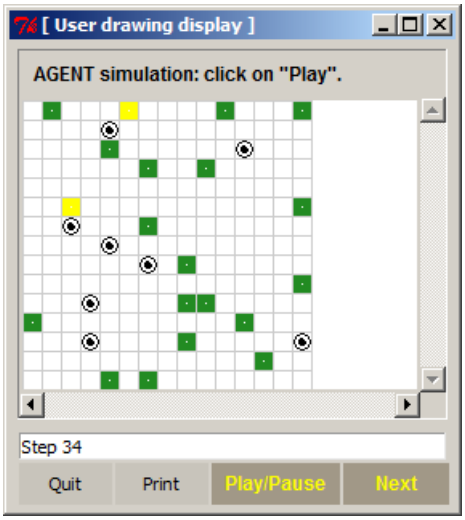



**Figure 21.2 -** Example of contents of the SQLfast graphic box (from an *Agent modelling* simulation)

The set of buttons of the box depends on the nature of the drawing. For still drawing, button **Quit** and **Print** appear, that closes the box. For animated drawing, additional buttons **Play/Pause** and **Next** are provided, allowing the user to start, suspend and resume the animation. An animated drawing is recognized by the presence of commands **`step`** in the SQLdraw script.

Button **Print** generates a vector graphic file from the drawing in **`eps`** (*Embedded PostScript*) format. This format preserves the quality of the drawing whatever the zoom factor. It can be inserted in Word, Powerpoint, Framemaker documents. It can easily be transformed into other formats (e.g. bmp, tiff, jpeg, gif) by standard picture manipulation software, such as Adobe Acrobat, GIMP or Ghostview.

# Chapter 22

# Extending SQLfast

This chapter describes several techniques to enrich the SQLfast environment by creating new custom-made functions and tools.

## 22.1 Introduction

Several approaches allow for SQLfast extension:

- SQLfast utilities: service scripts, functional scripts
- Python external applications and libraries
- SQL-based user-defined functions (UDF)
- SQLfast language extension
- Foreign programs.

Sections 22.2 to 22.6 describe these techniques. In Sections 22.7 to 22.9, we show how to develop custom-made extensions through Python external applications and functions, and through UDF. The last section enumerates the main functions of Python external function libraries.

## 22.2 SQLfast utilities

SQLfast scripts can be used to implement small, general purpose, utilities.

### 22.2.1 Service scripts

A *service script* is an ancilliary SQLfast script that performs some low-level function such as parameter settings. Service scripts have no particular meaning for end users but are called by various functional scripts or by applications. In Chapters 16 and 17 (*Writing data tables*), the format parameters scripts, such as UTIL-SELECT-parameters-for-LATEX.sql for LaTeX, are typical service scripts. Service scripts are executed by **`execSQL`** statements:

```
execSQL UTIL-SELECT-parameters-for-LATEX.sql;
select * from CUSTOMER #[1,10];
```

**Script 22.1 -** Calling utility service UTIL-SELECT-parameters-for-LATEX.sql

### 22.2.2 Functional scripts

A functional SQLfast script performs a well defined user function, such creating and loading a test database, checking the quality of the data in a database or generating a report. Functional scripts are executed by **`execSQL`** statements. They are supposed to be sufficiently generic to meet a wide range of requirements.

```
execSQL Check-DB-Consistency.sql;
```

**Script 22.2 -** Calling functional script Check-DB-Consistency.sql

## 22.3 Python external applications and libraries

SQLfast can invoke and use members of Python modules. Some of them may be full applications while others are libraries of functions.

### 22.3.1 External SQLfast applications

An external application is a standalone program that performs some high level user function. It is written in a standard programming language (Python in this version). An external application is executed through statement **`execApp`**. It may use input arguments and may produce results.

Its results, if any, are stored in SQLfast variables and input arguments are comma-separated character string constants.

Two important recommendations:

– spaces between arguments must be avoided unless they are part of the constants (no space before and after commas)
– if an input argument includes (or may include) commas, it must be enclosed between curly brackets.

**Examples**

```
execApp A,B,C = JsonToSQLconverter SourceFile.txt,$style$;
execApp SQLdrawViewer Life.draw;
execApp res = SendMail $recipient$,{Hello, world},{$body$};
```

A Python application must include an entry point, a function called **Main()**.[1]

### 22.3.2 External SQLfast function libraries

In this version of SQLfast, a *function library* is a Python module made up of function definitions. Typical examples: string manipulation, syntactic analysis, file manipulation, custom dialog boxes, text file transformation, etc. A function is executed through statement **execApp** (*with* function name) or its synonym **function**. Some examples: **LStr.py** library, that includes some basic string manipulation functions, **LFile.py** library, that provides file and directory manipulation functions, **LGeo.py**, that includes basic 2D geometric functions. They are given a short description in the last section.

In the following example, function **KeyValue** from library **LStr** is called to decompose any expression of the form "key = value" into its key and value components:

```
function key,value = LStr:KeyValue $param$;
```

**LStr** is the name of the Python module (actually **LStr.py**) and **KeyValue** is the name of one of its member functions.

Unlike UDFs, that are described below, Python functions cannot appear in expressions but they can return several values. They should naturally be preferred when implementing complex functions, possibly interactive and data intensive.

## 22.4 User-defined functions (UDF)

Functions can appear in computing and condition expressions, in particular in SQLfast **compute**, **if** and **while** statements:

```
compute D = sqrt(1 + int($X$));
compute ad = lower('$n$')||' '||replace('$ad$','str','street');
```

1. **execApp** A = **App** B; is equivalent to **function** A = **App:Main** B;

```
if (not fileExists('$db$')) showMessage Database is missing;
select upper(City) from CUSTORDER where year(DateOrd) = 2016;
```

The functions used in these examples (`sqrt`, `int`, `lower`, `replace`, `fileExists`, `upper`, `year`) are natively built in SQLfast. However, this set of functions can be extended quite easily thanks to a mechanism called *user-defined function* (UDF). Developping custom-made UDFs is a facility provided by most DBMSs to enrich the set of functions that can be used in SQL queries.

SQLfast relies on this mechanism not only to add functions to the SQL language, but also to let users create new functions that can be used in SQLfast **compute**, **if** and **while** statements:

The new user-defined functions are coded in two system libraries

- **SQLfastUDFlib.py**, that extends the function set available in SQLfast,
- **SQLiteUDFlib.py**, that extends the function set available in SQL (more specifically SQLite in the standatd version).

These libraries, which are Python modules, are included in the standard SQLfast distribution and can be modified and augmented by the user. They provide (almost) the same set of functions, but, due to slight difference in the way they are used, they are defined in two different modules.

## 22.5 SQLfast language extension

The SQLfast loader, that is in charge of loading scripts on demand in the SQLfast script store, includes a preprocessor that translates complex statements, such as `for-endfor` and `while-endwhile` into basic SQLfast statements. This translator can be augmented to make it process other statements. In this way, we can modify and augment the SQLfast language itself in an easy and efficient way. Example: changing the name of a statement, new `increment` and `decrement` statements, the `askInsert` statement, adding SQLfast *triggers* associated to some statements (similar to AspectJ for Java), etc.

This feature is not available yet.

## 22.6 Foreign applications

A foreign application is a preexisting commercial, open-source or proprietary software, independent of SQLfast, that can be called from an SQLfast script through statement **`execProg`**, as in the following example, that launches Microsoft Excel program on worksheet CUSTOMER.csv:

```
execProg EXCEL.exe $fileDirectory$/CUSTOMER.csv;
```

The execution of statement **execProg** is asynchronous. It starts the foreign program then continues the execution of the current script. There is no automatic synchronization mechanism, though it could be manually simulated by a `pause` statement.

Foreign programs can be used to display documents that are more complex than mere text files. For example CSV, HTML, XML or RTF files generated by an SQLfast script can be automatically opened by their respective text processors. Similarly, PDF, JPG or AVI files extracted from a database can be opened automatically.

SQLfast provides a collection of system variables that can be used to locate the default file processors for usual file formats:

- **TEXTprocessor**: displaying and processing pure text files (.txt);
- **WSprocessor**: displaying and processing worksheet files (.csv, .xls, etc);
- **DOCprocessor**: displaying and processing document files (.doc, .docx, .odt, etc.);
- **IMAGEprocessor**: displaying and processing images files (.jpg, .bmp, .png, .gif, etc.);
- **PDFprocessor**: displaying pdf files (.pdf);
- **HTMLprocessor**: displaying web pages and other similar files (.htm, .html, .xml, etc.);
- **VIDEOprocessor**: playing video files (.avi, .divx, .mpeg);
- **SOUNDprocessor**: playing audio files (.wma, .mp3, .ogg, etc.);

These variables can be set in the SQLfast.ini initialization file and in any script just like standard variables. Example of usage:

```
execProg $WSprocessor$ $fileDirectory$/CUSTOMER.csv;
```

## 22.7 Developing custom-made external applications and libraries

Now, we examine the procedure to add to the standard SQLfast distribution new Python external applications and function libraries.

### 22.7.1 General procedure

This procedure comprises three steps:

A. **Writing** (or updating) the code into the Python module that codes the application or the set of functions. This step requires the preliminary installation of Python version 2.7.

B. **Compiling** the module.

C. **Integrating** the compiled module into the SQLfast executable.

Let us call **TextProcessing.py** the application we want to create.

### *A. Writing the code of the module*

We create (or we modify, if it already exists) a Python program called **TextProcessing.py**. We save the module and we check it for syntax errors.

If the module implements an application, it must include an entry point called **`Main()`**. This function receives the input arguments of the application and returns its results. This function is not needed for pure function library.

### *B. Compiling the module*

Now, we compile the module into file **TextProcessing.pyc**. Since Python does not save the compiled version of main programs, we create a small main program that calls this module.

We create a file named, say, `Compile_SQLfast_module.py`, which includes just one line of code:

```
import TextProcessing
```

We save this file and we execute it (double-click or key **`F5`** in IDLE). Now we have a fresh version of **TextProcessing.pyc**. If the application module calls other modules, the latter also are automatically compiled into ***.pyc** files.

### *C. Updating the SQLfast executable*

Now, we go back to the SQLfast directory that contains the standard distribution.

- file **`TextProcessing.pyc`** is added to archive **`library.zip`** (drag and drop),
- files **`*.pyc`** resulting from the compilation of the modules called by **TextProcessing.py** are also added to archive **`library.zip`**,
- source file **`TextProcessing.py`** is copied in system directory SQLfast. The source files of the modules called by **`TextProcessing.py`** need not be added.

Now, the new application or functions can be used by SQLfast scripts.

## 22.8 Developing custom-made UDF for SQLfast and SQL

We modify the source code of Python module to add or modify SQLfast UDF functions. To add new SQLfast functions or to modify existing ones, we modify Python source module **SQLfastUDFlib.py**. Similarly, to add functions to SQL, we modify source module **SQLiteUDFlib.py**.

The general procedure is that used for external applications and functions, with some slight differences.

Let us add function **isEven(N)**, that indicates, though result **1** or **0**, whether **N** is an even number or not. This new function will allow us to write such (unsecure!) code:

```
ask n = Enter a number from 1 to 100;
if (isEven($n$));
   write $n$ is an even number;
else;
   write $n$ is an odd number;
endif;

ask A,B,C;
if (isEven($A$) + isEven($B$) + isEven($C$) between 1 and 2)
    write There are at least one even number and one odd number;
```

## *A. Writing the code in the SQLfast library*

The code of all the additional SQLfast functions is collected in Python module **SQLfastUDFlib.py**.

This module contains the definition of a series of functions the names of which are prefixed with `'SQL'` (this is just a personal convention), as for **SQLrandom_f**`(n1,n2)`, which returns a floating point random number between **n1** and **n2**.

The module also includes the definition of function **createFunctions**`(connect)`, the role of which is to associate a list of functions to the database opened with connection `connect`. In this case, we may ignore this database, which is internal to the SQLfast engine.

When SQLfast starts, it opens its internal database, imports module **SQLfastUDFlib** from which it executes function **createFunctions**. From then on, all the functions defined in this module are available to write SQLfast expressions.

Now, let us code our function **isEven**.

We open file **SQLfastUDFlib.py** with Python IDLE code editor or with any text editor. We add the definition of **SQLisEven** function, which could be the following[2]:

```
def SQLisEven(n):
    return n % 2 == 0
```

2. Actually, this code makes a strong assumption that may not be satisfied at execution time: **n** must be an integer value. What if **n** is a floating point number? or if it is a numeric character string? or if it is an arbitrary character string? or if it is None? Obviously, a more *tolerant* but also more *defensive* code should be written. However, this is another point that we will ignore here.

Then, we jump to the body of function **createFunctions**(connect), in which we add the following line:

```
connect.create_function(u"isEven",1,SQLisEven)
```

The first argument is the external name of the new UDF (isEven), the second one is the number of parameters of the function (1) and the third one is a reference to the internal function (SQLisEven) that implements the UDF. Script 22.3 shows what the code now looks like.

We save the module and we check it for syntax errors.

```
import sqlite3
def SQLrandom_f(n1,n2):
    # Returns a pseudo random FLOATING POINT number
    ...
    return rf
def SQLisEven(n):
    # ...
    return n % 2 == 0
    ...

def createFunctions(connect):
    connect.create_function("random_f",2,SQLrandom_f)
    connect.create_function("isEven",1,SQLisEven)
    ...
    return
```

**Script 22.3 -** Defining custom functions **isEven**

## *B. Compiling the library*

Using the procedure described above, we compile the library into file **SQLfastUDFlib.pyc**. Now we have a fresh version of **SQLfastUDFlib.pyc**, which includes the new function.

## *C. Updating the SQLfast executable*

The last step consists in replacing the old version of **SQLfastUDFlib.pyc** with the new one in archive **SQLfast.zip**. To replace it, we just drag the new file and drop it onto this archive icon. The source code need not be stored in this directory to use the new function, but it would be a good practice to replace it for further modification.

Now, the customized version of SQLfast, including function isEven is ready to use.

**Note**

Adding UDF to SQL follows the same procedure, applied to Python module **SQLiteUDFlib.py**

## 22.9 SQL queries in UDF

Can we write UDF that include SQL queries? The answer is positive, but the way to proceed is different for SQL and SQLfast UDF.

### 22.9.1 SQL statements in an SQL UDF

Through function `createFunctions(connect)`, the SQL UDF library, **SQLiteUDFlib.py**, is linked to an opened database denoted by argument **`connect`**, the connection object of this database. In addition, this connection is made persistent in the library through global object **`curConn`**:

```
def createFunctions(connect):
    global curConn
    curConn = connect
```

As a consequence, for each invocation of a function of this library, connection object **`curConn`** is available and can be used to access and even modify (with much caution!) the current database.

Let's show that through an example based on database ORDERS.db. We build the UDF **`custName(cust)`**, which returns the name of the customer given its customer Id.

```
def SQLcustName(cust):
    global curConn
    r = curConn.execute(u'select Name from CUSTOMER '\
        +u'where CustID = "'+cust+u'"').fetchone()
    return r[0] if r is not None else u''
```

Similarly, we define functions `custAddress`, `custCity`, `custCat`, etc. Therefore, we can write the following script, in which the join between CUSTORDER and CUSTOMER is implicit:

```
openDB ORDERS.db;
  select OrdID, DateOrd, custName(CustID) as "Name",
         custAddress(CustID) as "Address"
  from   CUSTORDER
  where  custCity(CustID) = 'Toulouse';
closeDB;
```

With the expected result:

```
+-------+------------+----------+----------------------+
| OrdID | DateOrd    | Name     | Address              |
+-------+------------+----------+----------------------+
| 30185 | 2016-01-02 | PONCELET | 17, Clôs des Erables |
| 30188 | 2016-01-03 | GILLET   | 14, r. de l'Eté      |
+-------+------------+----------+----------------------+
```

## Note

The SQL library already includes functions `custName`, `custAddress`, `custCity` and `custCat`, for experimentation purpose.

### 22.9.2 SQL statements in an SQLfast UDF

Can we apply the same approach for pure SQLfast UDF?

We can, but the procedure is a bit more complex, due to the fact that the global connection object of the module (**`dualConn`**) denotes the SQLfast internal database and not any current user database.

To extract data from a user database, such as ORDERS.db, whether the latter has already been opened by the current script or not, the UDF must open a connexion to the database, then extract the desired data and finally close the database. Clearly, this procedure will generally be more expensive than that of SQL UDF.

Let us rewrite function **`custName`**`(cust)` in SQLfast library **SQLfastUDFlib.py**.

```
def SQLcustName(cust):
   conn = sqlite3.connect(u'D:\\SQLfast\\Databases\\ORDERS.db')
   r = conn.execute(u'select Name from CUSTOMER '\
                    +u'where CustID = "'+cust+u'"').fetchone()
   nam = r[0] if r is not None else u''
   conn.close()
   return nam
```

The following script extracts data from the database without using any SQL statements:

```
compute name = custName('C400');
write $name$;

if (lower(custName('C400')) = 'ferard') write OK;
```

## Note

The SQLfast library already includes functions `custName`, `custAddress`, `custCity` and `custCat`, for experimentation purpose.

## 22.10 Complements: standard external SQLfast function libraries

### 22.10.1The *LStr* library

The LStr.py library provides some basic functions to manipulate, analyze and transform character strings[3]. This library is regularly modified.

### Short description of the functions

All input arguments and returned values are character strings. Positions in a string S range from 1 to length(S)[4]. Both results and input arguments are coma-separated list of elements. Spaces between a comma and the next argument are not ignored but are part of the latter. If the value of an argument includes commas, it must be enclosed between { and }:

```
function n = LStr:InString {Hello, World}, {,}, 1
```

- Ss = LStr:**Capitalize** ss : returns ss where the first character is in upper case and all the others are in lower case.
- ss = LStr:**UnFrench** sf : transforms European accented vowels (including French ones) of character string sf into pure ASCII characters. Accents are removed and placed after the character. E.g., **è** is replaced by **e`** and **ê** by **e^**. Note that **é** is replaced by **e``** and **ë** by **e```**, since neither ´ nor ¨ are ASCII characters. This transformation is reversible through the **French** function provided there is no "`" symbol following a vowel. Extending to other non-ASCII characters is quite easy.
- sf = LStr:**French** ss : restores European accented vowels from pure ASCII character string ss. (Conditionally) inverse of function **UnFrench**.
- ss = LStr:**ExtractLeft** s,n : returns the first n characters of s
- ss = LStr:**ExtractRight** s,n : returns the last n characters of s
- ss = LStr:**RemoveRight** s,n : returns s where the last n characters have been deleted
- ss = LStr:**RemoveLeft** s,n : returns s where the first n characters have been deleted
- pp = LStr:**InString** s,arg,pos : returns the position of string arg in string s, starting from position pos. Returns 0 if no instance was found
- cc = LStr:**NumToChar** i : returns the character with numeric code i mod 256, i < 0 is replaced by 0
- nn = LStr:**CharToNum** s : returns the numeric code of first character of string s, returns 0 if s is empty.

3. Some of these functions duplicate or extend the built-in character string functions described in Chapter 10.
4. This is different from the Python convention.

– ss = LStr:**SQLunquote** s : returns string s unquoted and with double quotes reduced.
– ss = LStr:**SQLquote** s : returns string s quoted in SQL style (external quotes and internal quotes doubled).
– ss = LStr:**SQLquote2** s : returns string s quoted in SQL style (internal quotes doubled but no external quotes).
– sl,sr = LStr:**KeyValue** s : returns the left and right parts of s, a *Key-Value* expression of the form **left** = **right**. Returns two empty strings if s is empty. Returns ( s,'') if no symbol **=**. Leading spaces are removed in **left** and **right**. Trailing spaces are removed in **left**.
– ss = LStr:**ExtractTaggedString** s,h,t : returns the fragment of s between h and first instance of t (excluded). Mainly used in HTML code analysis.
– ss = LStr:**NextSection** s,p,a,h,t : Extract from s, starting at position p, the fragment defined by tags a-t (included). Internal fragments h-t are kepts but skipped. Mainly used in HTML code analysis.
– ss = LStr:**PrettyPrint**(q,b): Returns SQL select query q nicely formatted. Line breaks are specified by b: platform-dependent end-of-line if b = 'eol' and '@n' if b = '@n'

### 22.10.2The *LFile* library

This library provides some special functions to manipulate file and directory paths as well as url.

#### Short description of the functions

– LFile:**neutralizePath** path: returns path where each backslash has been replaced by a slash.
– LFile:**splitPath** path: returns various components and versions of path.
– LFile:**infoFile** path: returns the size and various dates of file path.

#### Warning

Though Windows syntax requires the use of backslash (\) to separate items in a path, it is recommended to uses character "/", according to Unix syntax. SQLfast will adapt the syntax according to the requirements of the platform.

### 22.10.3The *LWeb* library

This library provides some special functions for SMTP, HTTP, MIME and IMAP protocols. Not available yet.

### 22.10.4The *LGeo* library

This library provides basic functions to manipulate geometric objects. These objects can be created by generation scripts, external source transformation (*SRI Shapefiles* for instance) or by the SQLdraw Editor, notably to digitize maps. They can be rendered by SQLdraw, the graphical engine of SQLfast. The representation of geometric objects is derived from the *ESRI Shapefile* format.

According to SQLfast, the elementary geometric objects are the point, the line segment, the label, the text, the circle, the ellipsis and the rectangle. The core geometric object is the shape. A shape is a list of points, a list of independent line segments, a polyline, a polygon, a circle, an ellipsis, a rectangle or a text. A geometric form is a list of shapes. The SQLfast internal representation of a geometric form is a character string structured as a list of list.[5]

The list below describes some basic functions of the LGeo library.

#### Some LGeo functions

- **ShapeFileLoad**: Returns a string that represents the geometric form of a shapefile.
- **ShapeFileAnalysis**: Returns the shape type, the global bounding box, the number of shapes and the number of parts of a shapefile.
- **ShapesExtract**: Returns two strings that contain the name and the contents of a shape of a shapefile.
- **ShapesChangeBbox**: linearly transforms a geometric form from its current bounding box to a new one. An inexpensive version of ShapesMorph when both bounding boxes are rectangles
- **ShapesToDrawFile**: Converts a geometric form into SQLdraw format.
- **ConvertPolylineToPolygon**: Returns a polygon from a polyline.
- **LengthOfSegment**: Returns the length of a segment.
- **DistancePointPoint**: Returns the distance between two points.
- **DistancePointLine**: Returns the distance between a point and a line.
- **DistancePointSegment**: Returns the distance between a point and a segment.
- **AreaOfPolygon**: Returns the area of a polygon.
- **LengthOfPolyline**: Returns the length of a polyline.
- **PerimeterOfPolygon**: Returns the perimeter of a polygon.
- **Colinear**: Indicates whether three points are colinear.
- **PointOnSegment**: Determines whether a point is on a segment.
- **PointInPolygon**: Determines whether a point is in (or on) a polygon.
- **SegmentOutsidePoly**: Determines whether a segment is outside a polygon.

5. This representation is close to that of serialized tuples and lists of Python.

- **SegmentInsidePoly**(p,poly): Determines whether a segment is inside a polygon.
- **SegmentTouchPoly**(p,poly): Determines whether a segment touches a polygon.
- **SegmentIntersectPoly**(p,poly): Determines whether a segment intersects with a polygon.
- **polyOutsidePoly**: Determines whether a polygon is outside another polygon.
- **polyInsidePoly**: Determines whether a polygon is inside another polygon.
- **polyTouchPoly**: Determines whether a polygon touches another polygon.
- **polyIntersectPoly**: Determines whether a polygon intersects with another polygon.
- **ShapesMorph**: Transforms a geometric form by mapping it from its rectangular bounding box into another quadrilateral bounding box. *Example*: see Figure 22.1.

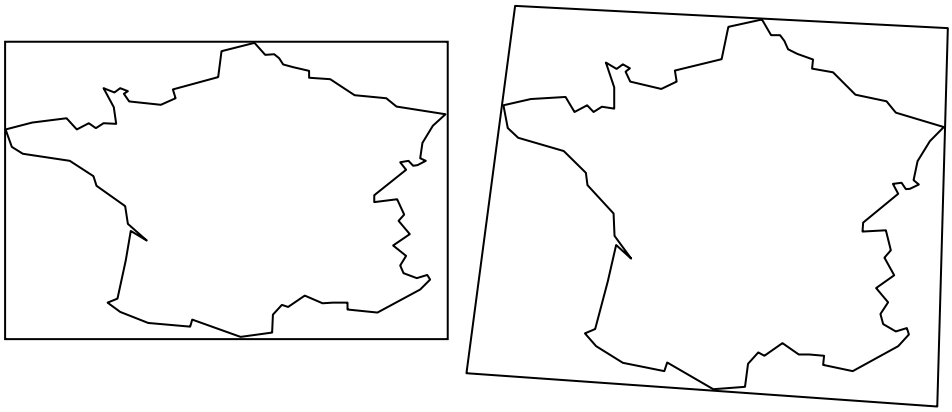


**Figure A22.1 -** Application of ShapesMorph: eliminating the distorsion due to the projection system

### 22.10.5Some recommandations for the development of libraries

- Use a short library name: **LStr** for ***L****ibrary of character* ***Str****ing functions*.
- Use significant function names: **LStr:Replace**
- All input and output arguments are character strings, even numeric ones.
- Apply defensive programming. The environment is particularly hostile: inexperienced programmers, missing input arguments, wrong argument types, inconsistent argument values, non-ASCII arguments, various character coding schemes, etc. By the way, the LStr library is not completely error proof in this respect!
- Decide on intuitive default values for arguments.
- Perform intensive testing, including unlikely and rare usages.

# Chapter 23

# Aid to SQLfast script development

**Objective**: This chapter describes the debugging aids provided by the SQLfast environment. Their purpose is simple: helping users understand the behavior of a script at runtime, particularly when the results of the execution are not what was expected.
Some of these means are simple and straightforward, such as monitoring the state of individual variables or displaying the value of all user or system variables. Others are more sophisticated, such as statement execution tracing, through which each statement executed is reported, included its source and prepared versions, through which the values of the variables can be observed. Tracing of trigger is more complex. It is supported by special UDF functions. Thanks to them a complete trace can be built, in which script statements and trigger statements are seamlessly intertwined.
Since the traces are stored in a system database, several ways to exploit them through SQL queries are suggested, such as computing the value flow of selected variables.

**Keywords**: debugging, tracing, source statement, prepared statement, variable, trigger, SQLfast compilation,

**Statements and functions**: trace on/off, output on/off, writeVariables, writeSystemVariables, Scripts>Tracing settings (Enable tracing, Trace SQL statements only, Append new traces), printMessage, writeTrace

**System variables and parameters**: procLevel, traceActive, traceSQL, traceAppend, loaderDebug

## 23.1 Introduction

Developing a non-trivial application is usually accompanied by multiple program crashes. Each of them must be analyzed in order to identify and fix the source error. Sometimes, a careful examination suffices to identify the problem, but in most cases, more powerful methods are required. This is the purpose of this chapter to describe these methods and to show how they can be used to perform what professionals call *program debugging*.

To fully understand the information provided by the debugging functions of SQLfast, we must first examine a particularity of the way statements are executed. This is the objective of the next section.

## 23.2 Two SQLfast language levels

The SQLfast language as it is known by users includes so-called *complex constructs* built from statements *if*, *else*, *endif*, *while*, *endwhile*, *for*, *endfor*, *next* and *exit*. This level of language can be called *user level SQLfast*.

However, the SQLfast engine, the component that executes the statements of SQLfast scripts, cannot process any of these constructs, but one, the `if-goto` form:

```
if (<condition>) goto <label>;
```

The reason is simple. Interpreting and executing these constructs directly would require a more complex and less efficient engine architecture. The usual way to cope with such state of affair is to preprocess user scripts to transform complex constructs into simpler forms based on elementary `if-goto` statements.

We say *usual way* because it is how *Java* programs, for example, are processed: they are first compiled into low level bytecode statements which are then executed by the *Java Virtual Machine*. The execution of *Python*, *Javascript*, *PHP*, and *C#* programs follows the same scenario.

As an illustration, we consider Script 23.1, which comprises two complex constructs: `if-endif` and `for-endfor`. Before execution, this source script is transformed into that of Script 23.2, in which the source layout is preserved (more detail on the development of the SQL *for-loop* in Section 9.12 of Chapter 9). The complex statements are prefixed with symbol '>>'. They are ignored by the SQLfast engine. Their role is to document the compiled code and to support the generation of execution traces as we will see below.

Complex statements are followed by their compiled code (both source and compiled statements are assigned the same statement number).

The existence of two levels of language can generally be ignored by the users. However, in the current version of SQLfast, error messages and tracing data are related to the compiled code. It is therefore useful, in order to understand these data and traces to examine the exact code that has generated them.

```
[1]  openDB Orders.db;
[2]    ask go;
[3]    if ($go$ = 1);
[4]      write $go$;
[5]      for c,n = [select CustID,Name from CUSTOMER
[6]                 where City = 'Poitiers'];
[7]         write $c$, $n$;
[8]      endfor;
[9]    endif;
[10] closeDB;
```

**Script 23.1 -** A source script as it was written by a user (line numbers added)

```
 0: [1]  openDB Orders.db;
 1: [2]    ask go;
 2: [3]    >> if ($go$ = 1);
 3: [3]    if (not($go$ = 1)) goto _ELSE-1;
 4: [4]      write $go$;
 5: [5]      >> for c,n = [select ... where City = 'Poitiers'];
 6: [5]      SQLfor_header select ... where City = 'Poitiers';
 7: [5]      if ('$SQLdiag$' <> 'OK') goto _EXIT-2;
 8: [5]      label _NEXT-2;
 9: [5]      SQLfor_fetch c,n;
10: [5]      if ('$SQLdiag$' <> 'OK') goto _EXIT-2;
11: [7]         write $c$, $n$;
12: [8]      >> endfor;
13: [8]      goto _NEXT-2;
14: [8]      label _EXIT-2;
15: [9]    >> endif;
16: [9]    label _ELSE-1;
17: [10] closeDB;
```

**Script 23.2 -** The compiled version of Script 23.1 (in blue, the generated statements)

This *examination* is controlled through system variable **loaderDebug**. This variable is set in initializing file SQLfast.ini. It can also be set and looked at from within any script (`set loaderDebug = 2`). Variable **loaderDebug** can take four values:

0: no op (default),

1: print the compiled script in the output window (Script 23.2),

2: print the compiled script in the output window and stores it as an executable script in the *Scripts* directory,

3: print the source script (Script 23.1), the compiled script and the label dictionary in the output window. A nice way to examine how complex statements are compiled.

## 23.3 Examining the contents of variables and parameters

Three SQLfast statements can be used to display the state of all *user variables*, *system variables* and *parameters* at any execution point.[1] The information is printed on the current output channel (the output window by default).

- Display the **user variables** and their values
  The variable name is prefixed with symbol **[u]** for a printable value or **[b]** for a (non printable) binary value.

```
set A = 123;
set ver = First;
writeVariables;
```

*Result*:

```
[u] A = 123
[u] ver = First
```

- Display **system variables** and their values

```
set A = 123;
writeSystemVariables;
```

*Result (excerpts)*:

```
DBMS              = SQLite
DBMSversion       = 3.32.01
DIALOGbutton      =
DOCprocessor      = "C:/Program Files/Microsoft Office/
                     WINWORD.EXE
...
WmaxRowNbr        = 36
WmaxRowWidth      = 255
autoexec          =
...
buttonNO          = Cancel
buttonYES         = OK
coding_db         = utf8
coding_dialog     = utf8
colorBoxFrame     = grey90
colorBoxMessage   = grey85
colorButton       = grey85
```

1. Of course, in addition to the standard `write` statements for displaying selected variables at critical points of the programs under examination.

```
...
colorTextUpdate  = white
csv-header       = 1
csv-maxerrornbr  = 10
...
dispNull         = --
dynaColumns      = 0
dynaMarker       = !
empty            =
...
space            =
startTimer       = 2020-09-23 11:00:54.435000
stdTextX         = 64
stdTextY         = 20
styleBoxFrame    = flat
...
traceSQL         = 0
traceTime        = 0
welcome          = 0
```

- Display **parameters** and their values

```
set A = 123;
writeParameters;
```

*Result*:

```
autoCloseDB     = mainscript
charSize        = 9
commitmode      = transaction
delim2L         = &
delim2R         = &
delimL          = $
delimR          = $
dictionary      = 0
label-separator = |
maxIVV          = 10000
maxProcLevel    = 101
outputmode      = write
outputtype      = window
...
userlevel       = 1
```

In testing and debugging complex scripts, we will probably write these statements as follows:

```
set debug = 1;
...
if ($debug$) write $cn$, $nam$, $cit$, $acc$;
...
if ($debug$) writeVariables;
```

Note that the set of system variables and parameters can change in future SQLfast versions.

## 23.4 Enabling/disabling script output display

The results of the execution of scripts are displayed on the output channel (generally the output window).

The display of results can be disabled by statement

```
output off;
```

or, globally, by menu item **Script > Output off** (see Figure 23.1).

Conversely, this display can be reactivated by statement

```
output on;
```

or, globally, by menu item **Script > Output on**.

## 23.5 Tracing script execution

In general, tracing a software consists in recording, on an appropriate support, information on some of its possible executions. This information, called its *trace*, can then be examined, analyzed and processed in order to extract knowledge about the behavior of the software starting from definite initial conditions. This is particularly useful when one tries to understand an abnormal behavior of this software, for instance when it provides incorrect results or when it crashes.

This is the role of the tracing subsystem of SQLfast. When enabled, the tracing records in the **TRACING.db** database all the available information on each statement execution. For each of them, table **TRACES** includes a row describing the execution of one statement (a *statement trace*), comprising the following information (Script 23.3):

– **TraceID**: the trace identifier. A sequence number automatically assigned to the trace. These values define the temporal ordering.

```
create table TRACES(
       TraceID    integer not null primary key autoincrement,
       Tindex     integer,
       Ttimestamp timestamp,
       Tproc      varchar(1024),
       Tlevel     integer,
       Tnature    integer,
       RawStat    varChar(1024),
       PrepStat   varChar(1024));

create table VARIABLES(
       Variable   varchar(64),
       AssignedIn integer);
```

**Script 23.3 -** Schema of the TRACING.db database

- **Tindex**: the line number of the statement in the source code (more generally the line number of its first fragment).
- **Ttimestamp**: the value of the system clock (down to the millisecond) just before the statement is executed.
- **Tproc**: the name of the procedure (or script) of the statement.
- **Tlevel**: the level in the calling tree of the processes of the procedure. The main script is at level 0. When procedure A, at level N calls procedure B, the level of B is N+1. When tracing a program, the same procedure can appear at different levels. This is particularly true in recursive procedures and triggers.
- **Tnature**: the nature of the statement. The nature of native elementary statements is coded 0. Complex statements have Tnature = 1 and the elementary statements resulting from the compilation of these complex statements have Tnature = 2. A complex statement and all its elementary generated statements share the same Tindex value. So the traces with nature 0 and 1 describe the execution of the statements visible in the source code.
- **RawStat**: the copy of the statement before its preparation. May include variable names.

  *Example*: `write Customers in $city$;`
- **PrepStat**: the statement after it has been *prepared*, that is, when all the variable names have been replaced by their values.

  *Example*: `write Customers in Paris;`

The database also includes table **VARIABLES**, comprising the name of a variable (column Variable) and the Id of a trace where this variable was assigned a value (column AssignedIn). Since this information is critical in many trace analysis processes, it is automatically extracted from the content of table TRACES. This

extraction concerns the variables from all traces, i.e, of native elementary, native complex and generated elementary statements. To reduce the extraction to a subset of the traces, for example, those of elementary statements, or those of source statements, we recommend to execute script **TRACING-Extract-variables.sql**, in which we can limit the extraction to specific natures of statements.

To experiment with the tracing function of SQLfast, let us consider script **TRACING-Main** shown in Script 23.4. It includes two embedded loops from which it calls procedure **TRACING-Write**.

```
-- TRACING-Main.sql --
[1]  openDB ORDERS.db;
[2]  set dir = 0;
[3]  set city = Paris;
[4]  compute pro = upper('pa45');
[5]  for cid,nam = [select CustID,Name
[6]                 from CUSTOMER where City = '$city$'];
[7]    for I = [1,2];
[8]       extract pid,des = select ProdID,Description
[9]                         from PRODUCT where ProdID = '$pro$';
[10]      execSQL TRACING-Write.sql;
[11]   endfor;
[12] endfor;
[13] closeDB;

-- TRACING-Write.sql --
[1] if ($procLevel$ = $dir$);
[2]   write $cid$, $nam$;
[3] else;
[4]   write $pid$, $des$;
[5] endif;
```

**Script 23.4 -** Script TRACING-Main (top) calls Script TRACING-Write.sql (bottom) from within two *for* loops

We enable the tracing through menu item **Script > Tracing settings** (Figure 23.1). A small box pops up, in which we can check three buttons (Figure 23.2):

- **Enable tracing**: enable the tracing of the execution of the main script, and of all the procedures it calls. When the main script finishes, the tracing is automatically disabled.
- **Trace SQL statements only**: when tracing is enabled, record only the traces of the SQL queries, as well as all database related statements (`openDB`, `commitDB`, `SQL-for` loop, `extract`, etc.)
- **Append new traces**: by default, enabling the tracing *resets* (empties) the TRACING.db database. If this button is checked, the future traces will be *appended* to the current contents of the tracing database. To use with care, since the traces and variables of several executions will be mixed.

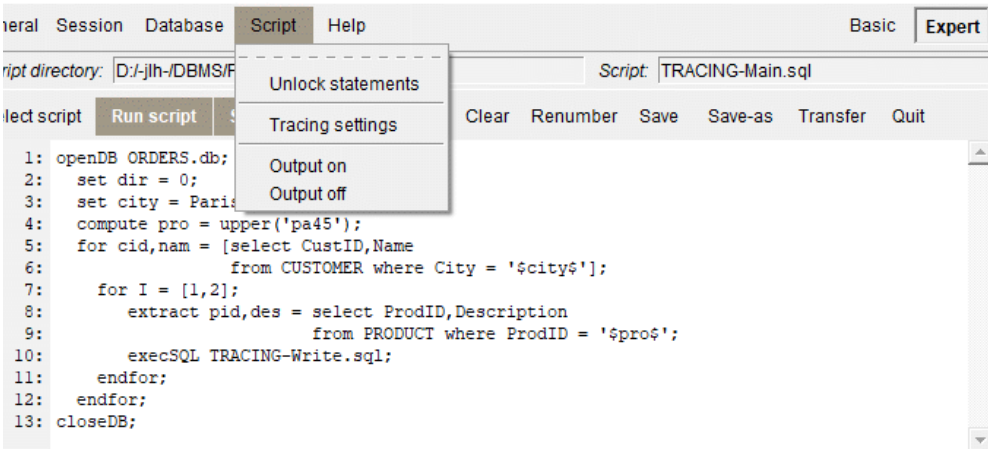



**Figure 23.1 -** The Script menu

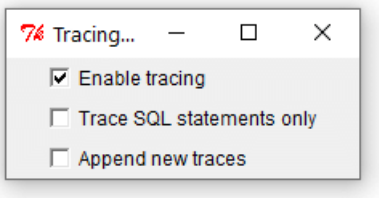



**Figure 23.2 -** Enabling the tracing of the next script execution

Having checked the enabling button, we execute script TRACING-Main.sql through button **Select & Run** of the main window.

The Show data function (menu **Database > Show DB data**) allows us to examine the contents of the TRACING.db database (Figure 23.3). Table TRACES comprises 51 rows. Reducing these traces to those of the source statements (select the rows for which `Tnature < 2`, to exclude generated statements), we count 20 traces that all refer to the statements of Script 23.4 exclusively.

## 23.6 Tracing selected code sections

The mechanism described in the previous section applies to all the statements of all the procedures executed. For complex and long running programs, this will generate quite a high volume of traces. When we are interested in the behavior of small parts of the program only, we would better manually specify these parts with SQLfast

statements `trace off` and `trace on`. The first one suspends the tracing while the latter resumes it.

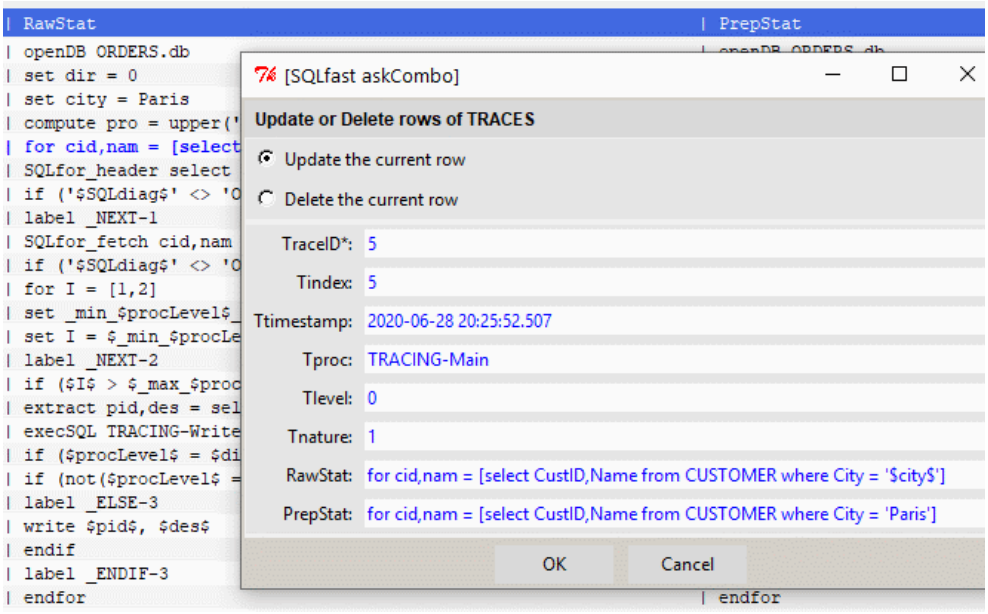
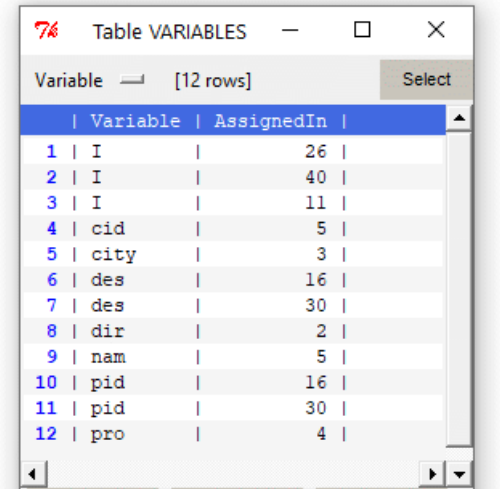



**Figure 23.3 -** Examining the trace the first execution of statement [5] in procedure TRACING-Main.sql




| | Variable | AssignedIn |
|---|---|---|
| 1 | I | 26 |
| 2 | I | 40 |
| 3 | I | 11 |
| 4 | cid | 5 |
| 5 | city | 3 |
| 6 | des | 16 |
| 7 | des | 30 |
| 8 | dir | 2 |
| 9 | nam | 5 |
| 10 | pid | 16 |
| 11 | pid | 30 |
| 12 | pro | 4 |

**Figure 23.4 -** Table VARIABLES shows the variables that have been set during the execution of the script, together with the traces in which they have been set

Let us suppose that we are only interested in the execution of procedure TRACING_Write (Script 23.4). We modify the main script as shown in Script 23.5. Then, after enabling the tracing, we execute the main script. The traces comprise those of the `execSQL` statement, followed by those of procedure TRACING-Write.

Note that `trace` statements themselves are not traced.

```
trace off;
openDB ORDERS.db;
set dir = 0;
set city = Paris;
compute pro = upper('pa45');
for cid,nam = [select CustID,Name
               from CUSTOMER where City = '$city$'];
  for I = [1,2];
     extract pid,des = select ProdID,Description
                       from PRODUCT where ProdID = '$pro$';
     trace on;
     execSQL TRACING-Write.sql;
     trace off;
  endfor;
endfor;
closeDB;
```

**Script 23.5 -** Limiting the tracing to the execution of procedure TRACING.Write

## 23.7 Tracing trigger execution

Understanding how a *trigger system* (the set of triggers of a database) works at execution time may be particularly hard. Indeed, the execution model of SQL triggers combines event-based, logic-based (through the `when` clause as well as the `where` clauses of the body queries) and procedural paradigms, all depending on the execution flow of application programs, itself driven by the contents of the database and on the data collected through the GUI.

When a script fails, and when a careful examination of its source code appears insufficient to detect the faulty sections, tracing some of its representative executions often is a practical way to follow the progress of its execution. We have discussed that for SQLfast scripts in Section 23.3 to 23.6.

Tracing the execution of triggers must bring the same valuable information on what happens when these triggers fire. Some big DBMS include functions to generate traces of SQL queries, including those in trigger bodies. This is not the case of SQLite on which the current version of SQLfast relies. Unfortunately, due to the architecture of SQLite triggers, the SQLfast engine cannot control their components the way it does for plain SQLfast statements.

To generate the desired trace of trigger execution, we will use a somewhat primitive technique consisting in *explicitly* writing information in the output window and/or in the TRACING.db database from trigger bodies. This approach is generally called *code instrumentation*. For this, SQLfast provides two functions:

- `printMessage(s)`: write message **s** in the output window.
- `writeTrace(idx,tim,pro,lev,nat,raw,prep)`: insert into table TRACES a row formed by the values of the arguments.

The case study on active databases (entitled, as expected, *Active Databases*), provides a nice application of these procedures to two tracing variants: *event* (logging the events that fire triggers) and *query* (logging the data modification queries of trigger bodies).

### About SQLite triggers

Ideally, we should be allowed to write, in the body of a trigger, a procedure call such as:

```
execute procedure printMessage(s);
```

as in Postgresql, or, simply, as in Oracle PL/SQL:

```
printMessage();
```

Unfortunately, the body of an SQLite trigger must be made up of a sequence of SQL data modification queries only (`insert`, `update`, `delete`) plus a simple form of `select` query initially intended to (conditionally) raise an exception. So, no explicit procedure nor function call.

We will consider the following syntax of this variant of `select` query:

```
select <function> [where <condition>];
```

Where `<function>` denotes a UDF function that executes some desired action. So, instead of

```
printMessage('Step 1');
```

we will write:

```
select printMessage('Step 1');
```

In the same way, instead of:

```
if (<condition>) printMessage('Step 1');
```

we will write:

```
select printMessage('Step 1') where <condition>;
```

If we find this syntax a bit weird, we can improve it as follows:

```
set message = select printMessage;
```

Then, wherever in the body of a trigger we want to write `<something>` in the output window, we simply write:

```
$message$(<something>);
```

Similarly, defining

```
set trace = select writeTrace;
```

allows us to write:

```
$trace$(...);
```

In Script 23.6, we have used the `$message$` function to log events in the output window and the `$trace$` function to store query traces in the TRACING.db database.

```
set message = select printMessage;
set trace   = select writeTrace;

create trigger TRG_CUSTOMER_INS
begin
   ...
end;

create trigger TRG_PRODUCT_INS
after insert on PRODUCT
begin
   $message$('Enter TRG_PRODUCT_INS: ('
           ||new.ProdID||','||new.Description||','
           ||new.Price||','||new.QonHand||')' );
   $trace$(1,current_timestamp_full(),'TRG_PRODUCT_INS',1,0,
         'insert into ARCHIVE values (new.OrdID,new.DateOrd)',
         'insert into ARCHIVE values ('
         ||new.OrdID||','||new.DateOrd||')' );
   insert into ARCHIVE values (new.OrdID,new.DateOrd);
   $trace$(2,current_timestamp_full(),...);
   update ACTIVITY set N = N + 1 where CustID = new.CustID;
   $message$('Exit TRG_PRODUCT_INS');
end;
```

**Script 23.6 -** Inserting tracing statements in the body of a trigger

In `$trace$` function calls, the process level of the trigger execution has been arbitrarily set to 1. The actual level, considering the level of the source event (whether it comes from an SQLfast script or from a parent trigger), can be computed but it would require a special technique that has been applied in the Case study on ***Active databases***, which the reader is referred to. Therefore we will not develop it in this chapter.

## 23.8 Analyzing and processing traces

The primary goal of generating the trace of the execution of a complex script is to understand why the latter has produced unexpected results and in particular to identify where the script has failed. This is an elegant way to just say *debugging this script*.

So far, the traces can be examined through the Show Data function of SQLfast (Figure 23.3). They can be sorted on any criterion and filtered in order to focus our examination on critical trace sequences.

This is quite fine to cope with short executions of small scripts, but this will not help us much understand what happened in large and complex applications all the more if they include many triggers. Indeed, such a trace will generally comprise several thousands of statement traces, a volume that is impossible to inspect visually or to analyze with the elemntary sort/select operations of Show Data windows.

We will show, through some examples, how SQL queries against the TRACING.db database can be used to analyze and extract information from execution traces.

### 23.8.1 Elementary queries

The query of Script 23.7 computes, for each procedure, the number of distinct source statements executed and the total number of their traces.

```
select Tproc,count(distinct Tindex) as Statements,
       count(*) as Traces
from   TRACES
group by Tproc;
```

**Script 23.7 -** Counting the statements and traces of each procedure

```
+---------------+------------+--------+
| Tproc         | Statements | Traces |
+---------------+------------+--------+
| TRACING-Main  | 11         | 39     |
| TRACING-Write | 4          | 12     |
+---------------+------------+--------+
```

The next query (Script 23.8) counts the traces according to their nature.

```
select case Tnature when 0 then 'Native elementary'
                    when 1 then 'Native complex'
                    else 'Generated elementary' end as Nature,
       count(*) as Traces
from   TRACES
group by Tnature;
```

**Script 23.8 -** How many traces of each nature?

```
+----------------------+--------+
| Nature               | Traces |
+----------------------+--------+
| Native elementary    | 11     |
| Native complex       | 9      |
| Generated elementary | 31     |
+----------------------+--------+
```

Identify the (potentially) *sequential procedures*, those in which no loop has been executed (Script 23.9).

```
select distinct Tproc
from   TRACES T
where  Tproc not in (select Tproc from TRACES
                     where  RawStat like 'for %'
                     or     RawStat like 'while %');
```

**Script 23.9 -** Looking for sequential procedures

```
+---------------+
| Tproc         |
+---------------+
| TRACING-Write |
+---------------+
```

List of the source statements executed in procedure 'TRACING-Main.sql', together with the Ids of their traces (Script 23.10).

```
select Tproc,RawStat, group_concat(TraceID) as Traces
from   TRACES
where  Tproc like '%TRACING-Main' and Tnature in (0,1)
group by Tproc,Tindex
order by Tproc,Tindex;
```

**Script 23.10 -** List of the source statements executed in procedure TRACING-Main.sql.

| RawStat | Traces |
|---|---|
| openDB ORDERS.db | 1 |
| set dir = 0 | 2 |
| set city = Paris | 3 |
| compute pro = upper('pa45') | 4 |
| for cid,nam = [select CustID,Name from CUSTOMER [..] | 5 |
| for I = [1,2] | 11 |
| extract pid,des = select ProdID,Description from [..] | 16,30 |
| execSQL TRACING-Write.sql | 17,31 |
| endfor | 24,38 |
| endfor | 45 |
| closeDB | 51 |

### 23.8.2 Analyzing variables and values flow

Observing the evolution of variables during code execution is one of the most important way to understand how a program behaves. The tracing function of most IDE (Integrated Development Environment) include the monitoring of a selected set of variables. The tracing function of SQLfast, which only records the raw and prepared versions of the statements before their execution, does not include such a mechanism.[2] However by analyzing both versions of the statements, it is fairly easy to extract useful information on variable usage and evolution; the raw version shows the variables while the prepared version shows their values:

- raw version:

  ```
  compute A = $B$ + $C$;
  ```

- prepared version:

  ```
  compute A = 18 + 4;
  ```

The analysis of raw statements identifies two kinds of variables: those that are assigned a value and those the value of which is used. In the compute statement shown above, the substring between verb `compute` and symbol `'='` yields the name

2. This may change in future versions.

of the variable that receives a value. This is the way table VARIABLES is filled once all the traces have been recorded.[3]

As soon as all the variables that have been assigned a value at least once have been identified, the raw statements can be analyzed once again to detect the presence of these variables in substitution patterns. In the example above, provided that variables B and C have already been identified, patterns $B$ and $C$ indicates that these variables are *used* in this `compute` statement.

Now, let us examine some interesting ways to play with variables.

We start with a simple query based on table VARIABLES. Let us show the variables that are assigned a value by a *source statement* (Tnature = 0 or 1) together with the list of traces of these statements (Script 23.11). Joining VARIABLES with TRACES is necessary to get the nature of the statements.

```
select Variable,
       group_concat(distinct AssignedIn) as "AssignedIn"
from   VARIABLES V,TRACES T
where  V.AssignedIn = T.TraceID
and    T.Tnature in (0,1)
group by Variable
order by Variable;
```

**Script 23.11 -** Showing, for each variable, the traces of the statements that assign them a value

```
+----------+------------+
| Variable | AssignedIn |
+----------+------------+
| I        | 11         |
| cid      | 5          |
| city     | 3          |
| des      | 16,30      |
| dir      | 2          |
| nam      | 5          |
| pid      | 16,30      |
| pro      | 4          |
+----------+------------+
```

Now, we would like to know *where* the variables are *used* (Script 23.12).

3. Statements such as `set`, `ask`, `selectOne`, etc. also are processed in this way. Some of them may provide a list of variable names. This list must then be decomposed into individual names. Note that, in the current implementation of the tracing function, only user variables (so, excluding system variables) can be analyzed with this technique.

```
select distinct V.Variable,T.Tproc,T.Tindex,T.TraceID,T.RawStat
from   TRACES T,VARIABLES V
where  Tnature in (0,1)
and    T.RawStat like '%$'||V.Variable||'$%'
order by T.TraceID;
```

**Script 23.12 -** Showing the statements in which each variable has been used

In the result shown below, due to limited space in this page, some column names and values have been abbreviated.

```
+------+---------------+----+-----+----------------------------------------+
| Var  | Tproc         | Ix | Tid | RawStat                                |
+------+---------------+----+-----+----------------------------------------+
| city | TRACING-Main  | 5  | 5   | for cid,nam = [..] City = '$city$']    |
| pro  | TRACING-Main  | 8  | 16  | extract pid,des = [..] ProdID = '$pro$'|
| dir  | TRACING-Write | 1  | 18  | if ($procLevel$ = $dir$)               |
| pid  | TRACING-Write | 4  | 21  | write $pid$, $des$                     |
| des  | TRACING-Write | 4  | 21  | write $pid$, $des$                     |
| pro  | TRACING-Main  | 8  | 30  | extract pid,des = [..] ProdID = '$pro$'|
| dir  | TRACING-Write | 1  | 32  | if ($procLevel$ = $dir$)               |
| pid  | TRACING-Write | 4  | 35  | write $pid$, $des$                     |
| des  | TRACING-Write | 4  | 35  | write $pid$, $des$                     |
+------+---------------+----+-----+----------------------------------------+
```

If we complement each raw statement with its prepared version, we can know the actual value of each of these variables by visual inspection. Automatically extracting this information by comparing both versions would be possible but fairly tricky for large and complex statements, such as GUI dialogues and SQL queries.[4]

Since no statement explicitly assigns it a value, system variable procLevel (which provides the level of the current process in the call graph) has not been identified as a variable. This will be true for most system variable, which are implicitly set by SQLfast. To take it into account, either we add somewhere in a script a *nop* statement such as: `set procLevel = $procLevel$`,[5] or we add this variable explicitly to table VARIABLES.

Finally, by combining the information on the setting and usage of variables, we get a complete history of the variables. The result, expressed as a view, is shown in Script 23.13.

4. In any technical book, at this point of such a discussion, the author, who has unsuccessfully tried to solve the problem, generally writes: *This is left as an exercise for the reader.*
5. Actually, `if (0) set procLevel = $procLevel$`, since this system variable cannot be modified by a script.

```
create temp view VARIABLE_FLOW(Variable,TraceID,
       Set_in,Set_by_statement,Used_in,Used_by_statement) as
select Variable,TraceID,Tproc||':'||Tindex,RawStat,'','' 
from   VARIABLES, TRACES
where  AssignedIn = TraceID and Tnature in (0,1)
    union
select Variable,TraceID,'','',Tproc||':'||Tindex,RawStat
from   VARIABLES V, TRACES T
where  RawStat like '%$'||Variable||'$%' and Tnature in (0,1);
```

**Script 23.13 -** Computing the variable flow within statement traces

For each trace in which variable **v** (i.e., Variable = **v**) appears, the values of columns Set_in (procedure name and statement index) and Set_by_statement (the raw source statement) denote where, in the source code, **v** receives a value. Similarly, columns Used_in and Used_by_statement indicate where the value of **v** is used.

This information can be examined and processed in various ways. As an illustration, Script 23.14 displays the history of a selected variable in a readable format (Figure 23.5).

```
ask var = variable:[!select distinct Variable
                     from VARIABLES order by Variable];
if ('$DIALOGbutton$' = 'Cancel') stop;
for tid,setin,setby,usedin,usedby =
      [select TraceID,Set_in,Set_by_statement,
              Used_in,Used_by_statement
       from   VARIABLE_FLOW
       where  Variable = '$var$' order by TraceID];
   parameter delimiters = { };
   if ('{setin}' <> '');
      write Variable "{var}" is set by statement:
            @n@S6" {setby}" in {setin} [{tid}];
      set used = 0;
   else;
      if ({used} = 0);
         write ... then used in statement(s):;
         set used = 1;
      endif;
      write @S6"{usedby}" in {usedin} [{tid}];
   endif;
   parameter delimiters = default;
endfor;
```

**Script 23.14 -** Displaying the history of a selected variable

```
Variable "dir" is set by statement:
      "set dir = 0" in TRACING-Main:2 [2]
... then used in statement(s):
      "if ($procLevel$ = $dir$)" in TRACING-Write:1 [18]
      "if ($procLevel$ = $dir$)" in TRACING-Write:1 [32]
```

**Figure 23.5 -** History of variable **dir**

To show the actual content of the statements, including the substitution operators, as in `if ($procLevel$ = $dir$)`, it is necessary to disable this operator by changing the variable delimiters to, say, { and }.

### 23.8.3 Auditing business activities

The trace includes data on the operations that require human involvement, mainly data entry and operation selection through GUI. We can derive from them useful statistics such as:

- how many customer orders have been entered,
- how many supplier orders have been issued,
- how many operations have been canceled (`'$DIALOGbutton$' = 'Cancel'`)
- how much time have employees spent in each data entry operation (Script 23.15 and Figure 23.6)

```
select T1.TraceID,
       durationSeconds(T1.Ttimestamp,T2.Ttimestamp) as Time,
       T1.PrepStat as Statement
from   TRACES T1, TRACES T2
where  (T1.PrepStat like 'ask%'
        or T1.PrepStat like 'selectOne%')
and    T2.TraceID = T1.TraceID + 1
order by Statement;;
```

**Script 23.15 -** Computing the time spent on each data entry operation

| TraceID | Time | Statement |
|---|---|---|
| 246 | 8.591 | ask inv = [/bRegister a payment][..] |
| 212 | 5.406 | ask itm = [/bSelect an item to [..] |
| 216 | 5.112 | ask sup = [/bSelect a supplier [..] |
| 130 | 10.481 | ask-u oid,dat,cus,itm,qty = [..] |
| 173 | 81.894 | ask-u oid,dat,cus,itm,qty = [..] |
| 9 | 6.298 | selectOne op = [/bA RETAIL SHOP[..] |
| 47 | 3.835 | selectOne op = [/bA RETAIL SHOP[..] |
| 118 | 7.843 | selectOne op = [/bA RETAIL SHOP[..] |

```
| 161      | 3.518   | selectOne op = [/bA RETAIL SHOP[..] |
| 203      | 15.224  | selectOne op = [/bA RETAIL SHOP[..] |
| 237      | 5.749   | selectOne op = [/bA RETAIL SHOP[..] |
| 263      | 3.189   | selectOne op = [/bA RETAIL SHOP[..] |
+---------+--------+-------------------------------------+
```

**Figure 23.6 -** Where are the employees wasting their time?

### 23.8.4 Auditing computer activity

Similar information on computing resources can be derived from the traces:

- time spent in each procedure[6],
- number of `insert` operations executed per hour,
- cost of data modification operations. In the standard mode of SQLite, we observe that `insert` operations take no time but that the `commit` operation seems to be quite costly. Such a cost pattern generally means that the modifications are first applied in the DBMS cache, then copied on the external storage at commit time.
- identification of performance problems.

We must be aware that the timing information includes the tracing operations, some of which may be proportionally costly. As an example, inexpensive statement `set k = 0,` when traced, involves additional computing time plus a database `insert` operation. The proper tracing time is independent from the nature of the statement, so that we can get a fairly correct estimate of the tracing overhead by comparing the execution time of a representative sequence of operations with and without tracing enabled.[7]

### 23.8.5 Auditing SQL usage

SQL specialists and teachers often complain that many programmers use relational databases as mere data store (like an elementary direct access file), ignoring all the power and expressiveness of the SQL language.[8]

The examination of the traces of pure SQL queries can show the most frequent query patterns, and, therefore, the degree of *professionalism* of the progammer!

From another point of view, trace analysis can be used to check the compliance of the source code to standard valid programming patterns. For instance, between data modification queries and the closing of the database, the trace of either `commitDB` or `abortDB` must be encountered.

---

6. This information is not as easy to compute as it seems. Indeed, it is a particular case of *temporal projection*, as described in detail in the Case study ***Temporal databases - Part 2***.
7. Some preliminary experiments suggest that the average overhead never exceeds 10% of the script execution time.
8. See https://sigmodrecord.org/2019/10/19/richardhipp-speaks-outon-sqlite for example.

## 23.9 Checking trigger circuits

One of the major concerns of developers when they build a database the behavior of which is controlled by triggers is *whether the trigger system will terminate whatever the initial event that triggers it*. This property is difficult to evaluate since it depends, not only on the trigger global structure but also on values extracted from the database or provided by application variables.

Let us consider event **e0** that fires trigger **t0**, the body of which includes data modification queries. If these queries create events that are catched by other triggers, initial event **e0** starts a chain of data modification operations that can be of any length. This is what Figure 23.7 illustrates. When fired through an `insert into R` query, trigger TRG_R_INS executes two data modification queries that fire triggers TRG_S_DEL and TRG_T_UPD. These triggers, if their body also include data modification queries, will create events that are likely to fire other triggers, and so on.

This event-trigger pattern creates a *graph* in which nodes represent triggers and directed arcs represent the direct firing relation between two triggers. The initial question on trigger termination reduces to the nature of this graph. If the graph is *acyclic*, that is, if it contains no circuits, we are sure that, once started, the trigger system will terminate. If the graph contains circuit(s), then its execution *could* cause termination problems. Checking whether a directed graph is acyclic is a standard graph theory problem. Solving it is particularly simple and intuitive. However, *finding* these circuits may be a bit more difficult.

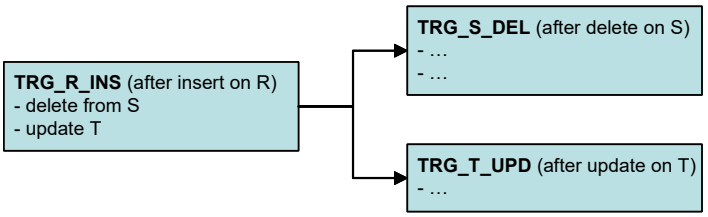



**Figure 23.7 -** When fired through an *insert into R* query, trigger TRG_R_INS executes data modification queries that fire triggers TRG_S_DEL and TRG_T_UPD

A study of the acyclicity of the trigger graph is developed in case study ***Active databases***, which the reader is referred to.

## 23.10 Summary

The SQLfast environment provides several techniques to help a developer examine the way scripts execute.

- The simplest technique consists in adding **`write`** statements that display the state of selected variables at critical points of the scripts.

- Similarly, statements **`writeVariables`**, **`writeSystemVariables`** and **`writeParameters`** can be used to list all the variables and their values.
- Tracing the execution of a script (together with the procedures it calls) consists in recording in system database **TRACING.db** a copy of all the statements executed. Two versions are stored: **RawStat**, the source code (as it has been written by the user) and **PrepStat**, where all the variables have been replaced by their value. The database also includes the execution timestamp (**Ttimestamp**), the identification of the statement (its script path, **Tproc**, its line number, **Tindex**, and the process level, **Tlevel**).
- Tracing is controlled by system variable **traceActive**. This variable can be set in three ways: in initialization file SQLfast.ini (`traceActive = 1`), by statement `trace on/off` and through menu item `Scripts>Tracing settings> Enable tracing`.
- Tracing can be limited to database-related statements (SQL queries, `openDB`, `commitDB`, `extract`, `SQL-for` loops, etc.) through system variable **traceSQL**. It can be set in two ways: in initialization file SQLfast.ini (`traceSQL = 1`) and through menu item `Scripts>Tracing settings>Trace SQL stat ements only`.
- By default, when starting the tracing of a script, the previous content of the TRACING.db database is deleted. To preserve it and to let the future traces be appended, system variable **traceAppend** must be set to 1, either from file SQLfast.ini (`traceAppend = 0`) or through menu item `Scripts> Tracing settings> Append new traces`.
- To improve the readability of the variable and statement traces sent to the output window, it may be useful to disable the output of the script results through statement **`output off/on`** or through menu items `Scripts>Output on/off`.
- Since tracing data are stored in an SQL database, it is easy to extract useful informations such as when and where selected variables have been set and have been used.

## **C**hapter **24**

# Storing transient data - The SQLfast lists

**Objective**: This chapter describes an alternative way to store temporary data (that, usually, are stored in temporary tables), namely SQLfast list. An SQLfast list is a character string made up of a sequence of values separated by a definite symbol, such as comma or semi-colon. These lists can be processed by a collection of functions including set operators similar to those offered by SQL. Three examples of list application are developed and discussed.

**Keywords**: value list, list of lists, list processing, set operator, set inclusion, set equality

**Statements and functions**: list-based loop, group_concat, group_concat2, item, itemLen, itemSlice, itemSort, itemNSort, itemDelete, itemJoin, itemUnion, itemUnionAll, itemExcept, itemExceptAll, itemInter, itemInterAll, itemEqual, itemEqualAll, itemInclude, itemIncludeAll, select-Many, set inclusion

**System variables and parameters**: dynaColumns, dynaMarker, listSeparator, DColumns, itemSpeed

## 24.1 Transient data: temporary tables or SQLfast lists?

The temporary storage of a list of values is ideally done in ... a temporary table:

```
create temp table LOC as
    select distinct City from CUSTOMER order by City;
```

They can be easily manipulated using SQL queries. In some cases, however, one may find it useless to use all the power of this language to store and manipulate simple lists. Moreover, this storage technique may also prove unsuitable or inefficient for certain processes. *SQLfast lists* can then be used.

## 24.2 SQLfast lists

An SQLfast list is a character string made up of a sequence of items separated by a special symbol, just like **csv** format (but not quite identical). Thus, the contents of table LOC could be more simply replaced by the list:

```
Bruxelles;Genève;Lille;Namur;Paris;Poitiers;Toulouse
```

The separator here is the semicolon (';'). This is the standard default separator but any string can be used to separate items.

The items of a list are character strings, and more specifically numerical character strings when these items denote numbers:

```
Paris;123;4.50;-68;1.25E-8
```

To manipulate a list we need to know the separator with which it has been built. This must therefore be specified when using the list functions. To obtain the 3rd item of the $L$ list, we use the item function, for which we specify the list ('$L$'), the position of the item (3) and the separator (';'):

```
item('$L$',3,';')
```

If the separator is empty, then each character in the list is a distinct item. The following expression returns letter 'c':

```
item('abcdefghijklmnopqrstuvwxyz',3,'')
```

An item can be empty, such as the 3rd one in the following list:

```
Bruxelles;Genève;;Namur;Paris;Poitiers;Toulouse
```

When an item includes instance(s) of the separator, it will be protected by single quotes (or apostrophes). In the example below, the separator is the comma:

```
Bruxelles,...,Namur,'Paris, ville lumière',Poitiers, Toulouse
```

More generally, any item *may* be surrounded by quotes but *must* be if it includes the separator. If a protected item includes quotes, these will be treated as usual in SQL constants, by doubling them:

```
Bruxelles,...,Paris,Poitiers,'Villeneuve d''Ascq',Toulouse
```

## 24.3 Creating an SQLfast list

Like any value of a variable, a list can be created by assignment (`set`), computation (`compute`), read in a file or extracted from a database (`extract`). In the latter case, the aggregate function group_concat will be used, with or without duplicates, with or without sorting:

```
extract L = select group_concat(distinct City)
            from   CUSTOMER;
```

To better control the source of the list, we use the special SQLfast function `group_concat2` that extends the SQlite native function. It requires five parameters:

```
group_concat2(L,sort,dir,unique,sep)
```

where **l** is the item expression, **sort** indicates whether the values must be sorted (0 = no, 1 = yes), **dir** the *direction* of the ordering (0 = ascending, 1 = descending), **unique** the uniqueness (0 = keep duplicates, 1 discard them) and **sep**, the separator symbol (actually any character string).

So, we will write:

```
extract L = select group_concat2(City,1,1,0,',')
            from   CUSTOMER;
```

## 24.4 Manipulation of SQLfast lists

SQLfast lists are manipulated by specific functions, most of them being recalled here after. All these functions must be provided the specific item separator needed to decode the list string. These functions also exist in a version where the default separator is assumed. Their name is that of the normal function suffixed by letter **S**. They have no separator parameter.

Examples: `itemLenS('$L$')`, `itemSliceS('$L$',3,5)`.

### *Elementary manipulations*

– Return the element of a given rank in a list.

```
compute elem = item('$L$',3,',');
```

– Return the number of elements in a list, including empty elements.

```
compute long = itemLen('$L$',',');
```

– Return a sublist of a list containing elements of rank 3 to 5, including these.

```
compute sublist = itemSlice('$L$',3,5,',');
```

– Return the position in a list of the first instance of a value.

```
compute position = itemIndex('$L$','Toulouse',',');
```

– Sorting the elements of a list, each element being considered as a string of **alpha-numeric** characters. For example, '5' > '12'. Parameter D indicates the sorting direction: 0 = ascending, 1 = descending. The U parameter indicates whether duplicates are eliminated: 0 = no, 1 = yes.

```
compute sorted = itemSort('$L$',$D$,$U$,',');
```

– Sorting the elements of a list, each element being considered as a string of **numeric** characters representing a number. For example, '5' < '12'. Parameters D and U as above.

```
compute sorted = itemNSort('$L$', $D$, $U$, ',');
```

– Removing the element of a given rank from a list.

```
compute reduced = itemDelete('$L$', 3, ',');
```

### *Joining lists (from version 3.0)*

– Return the list resulting from the concatenation of elements of the same rank in two lists, separated by the specified separator. The 1st separator is that of the two lists, the 2nd one separates the two source values in the result and the 3rd one separates the elements in the result.

Example: `itemJoin('ABCD','1234','','-',',')`
*result*: `'A-1,B-2,C-3,D-4'`

```
compute L12 = itemJoin('$L1$','$L2$', ',', ';', ',');
```

## 24.5 Set-theoretic manipulations of SQLfast lists

It is possible to build a new list by set-theoretic operations applied to two source lists. You can also check whether two lists are equal or if one is a subset of the other. These operations exist in two variants: without duplicates (pure sets) and with duplicates (multisets). Some of these operators are similar to those in SQL.

– Union of two lists considered as sets of elements. The resulting list includes all the elements of source lists L1 and L2, but in a single instance, duplicates excluded. Similar to SQL operator `union`.

```
compute L = itemUnion('$L1$','$L2$', ',');
```

– Union of two lists considered as multisets of elements. The resulting list includes all the elements of source lists L1 and L2, duplicates included. Similar to SQL operator `union all`.

```
compute L = itemUnionAll('$L1$','$L2$', ',');
```

– Difference between two lists considered as sets of elements. The resulting list includes the elements of L1 that are not in L2, but in a single instance, duplicates excluded. Similar to SQL operator `except`.

```
compute L = itemExcept('$L1$','$L2$', ',');
```

– Difference between two lists considered as multisets of elements. The resulting list includes the elements of L1 that are not in L2, duplicates included. Similar to SQL operator `except all`.

```
compute L = itemExceptAll('$L1$','$L2$', ',');
```

– Intersection of two lists considered as sets of elements. The resulting list includes the elements of L1 that are also in L2, but in a single copy, duplicates excluded. Similar to the SQL operator `intersect`.

```
compute L = itemInter('$L1$','$L2$', ',');
```

– Intersection of two lists considered as multisets of elements. The resulting list includes the elements of L1 that are also in L2, duplicates included. Similar to the SQL operator `intersect all`.

```
compute L = itemInterAll('$L1$','$L2$', ',');
```

– Equivalence of two lists considered as sets of elements. Indicates whether lists L1 and L2 include the same elements, duplicates excluded. There is no such operator in SQL.

```
compute equal = itemEqual('$L1$','$L2$', ',');
```

– Equivalence of two lists considered as multisets of elements. Indicates whether lists L1 and L2 includes the same elements, duplicates included. There is no such operator in SQL.

```
compute equal = itemEqualAll('$L1$','$L2$', ',');
```

– Inclusion of a list into another list considered as sets of elements. Indicates whether items in list L1 also belong to L2, duplicates excluded. There is no equivalent operator in SQL.

```
compute included = itemInclude('$L1$','$L2$', ',');
```

– Inclusion of a list into another list considered as multisets of elements. Indicates whether items in list L1 also belong to L2, duplicates included. There is no equivalent operator in SQL.

```
compute included = itemIncludeAll('$L1$','$L2$', ',');
```

– Translation of a character string. Returns the string S where each instance of an item of L1 has been replaced by its corresponding item in L2. Similar to the translate SQL function but also works with substring translation. The intersection of lists L1 and L2 should be empty (otherwise ...).

```
compute conv = itemTranslate('$S$','$L1$','$L2$', ',');
```

## 24.6 List-based *for* loop (from version 3.0)

There is a particular form of loop that iterates over the successive elements of an SQLfast list. There are two ways to specify the *item separator* of the list, either the default separator (keyword **itemS**) or by specifying explicitle (**item(,)**) :

```
for city = [itemS Bruxelles;Genève;Lille;Namur]
    write $ville$;

for city = [item(,) Bruxelles,Genève,Lille,Namur]
    write $ville$;
```

## 24.7 Standard versions of SQLfast list functions

The value separator symbol is explicitly specified in list functions. However, there exists a *default*, or *standard*, separator that can be set in file SQLfast.ini (or, dynamically, in a script) through system variable **listSeparator**. This separator is *implicitly*

specified in special versions of the list functions, the names of which are suffixed by letter **S**. For instance, this statement

```
compute L = itemUnionS('$L1$','$L2$');
```

is equivalent to this one:

```
compute L = itemUnion('$L1$','$L2$','$listSeparator$');
```

## 24.8 Light versions of list functions

When executing list functions, the SQLfast engine checks the validity of the arguments and, when needed, performs the appropriate conversions. For instance where a number is expected, SQLfast allows it to be expressed as a numeric string. All these preprocessing activities can be considered too expensive. Actually, they are controlled by system variable **itemSpeed**, the value of which can be 0 or 1. If itemSpeed = 0, the checking and conversions are performed. If itemSpeed = 1, they are disabled, which can speed up the execution of the functions considerably. This variable can be set in the SQLfast.ini initialization file:

```
itemSpeed = 1
```

or, dynamically, by the script:

```
set itemSpeed = 1;
```

## 24.9 Three application examples

In this first example, this script asks the user to select, via a `selectMany` box, one or several cities, then displays the list of the customers of each of these cities.

This box should appear as in Figure 24.1, in which each check box is labelled with the name of a city.

We also decide to also give the target variables (one per check box) the name of the cities, so that the `selectMany` statement must look like:

```
selectMany Bruxelles,Genève,Lille,Namur,Poitiers,Toulouse
         = Bruxelles|Genève|Lille|Namur|Poitiers|Toulouse;
```

The structure of the dialog box (list of variable names and list of labels) does not depend on the schema of the database, as is usual, but on the current state of the data. therefore, this statement must be built dynamically, at run time.

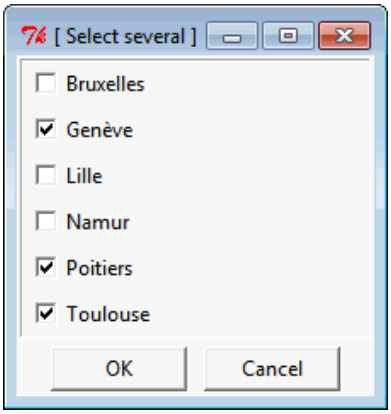



**Figure 24.1 -** Selecting cities

The two lists that form the statement have the same contents, except for the value separator. Let us store the list of variables names in variable (named) ... **variables** and the list of labels in variable **labels**. Therefore, the `selectMany` statement can be simply assembled as follows:

```
selectMany $variables$ = $labels$;
```

These lists are quite easy to build:

```
extract variables,labels
      = select group_concat2(City,1,1,0,','),
               group_concat2(City,1,1,0,'|')
        from CUSTOMER;
```

The customers of each selected city are displayed through a loop extracting each value from the city name list, which is stored in variable **variables**.

```
for city = [item(,) $variables$]
    if ($$city$$ = 1) select CustID,Name,Address,City
                      from   CUSTOMER
                      where  CITY = '$city$';
```

Loop variable **city** contains the successive values of the list, that is, the name of a city. This name is denoted by expression **$city$**. this name also is the name of one of the target variable, so that **$$city$$** is the value assigned to the check box of this city by the user. If this value is 1, then the customers of the corresponding city are displayed.

Hence the complete algorithm of Script 24.1.

```
openDB ORDERS.db;
extract variables,labels
      = select group_concat2(City,1,1,0,','),
               group_concat2(City,1,1,0,'|')
        from CUSTOMER;
selectMany $variables$ = $labels$;
   if ('$DIALOGbutton$' = 'Cancel') stop;
for city = [item(,) $variables$]
    if ($$city$$ = 1) select CustID,Name,Address,City
                      from   CUSTOMER
                      where  CITY = '$city$';
```

**Script 24.1 -** Displaying the customers of selected cities

In the second application, we implement the concept of *list of lists*. The main list contains a sequence of records separated by semi-colons (supposed to be the default separator). Each record is a list of values, separated by commas.

As an example, let us represent the data of two rows from the CUSTOMER table as a list of lists:

```
B062,GOFFIN,'72,r. de la Gare',Namur;C400,FERARD,'65,r. du
Tertre',Poitiers
```

To process this list, we build two nested loops, as shown in Script 24.2.

```
set customers =  B062,GOFFIN,'72,r. de la Gare',Namur
                ;C400,FERARD,'65,r. du Tertre',Poitiers;
for customer = [itemS §customers§]
    for attribute = [item(,) §customer§]
        write $attribute$;
```

**Script 24.2 -** Processing lists of lists

The third example illustrates the power of SQLfast list on a fairly difficult problem, namely computing the inclusion relation among a set of sets.

We consider table BUY(City,ProdID), each row (c,p) of which tells that city c buys product p, or, more precisely, that, in city c, at least one customer has ordered product p. The rows of BUY are derived from our ORDERS.db database as follows:

```
create temp table BUY as
   select distinct City,ProdID
   from   CUSTOMER C, CUSTORDER O, DETAIL D
   where  O.CustID = C.CustID and D.OrdID = O.OrdID;
```

Now, we want to create all the couples (c1,c2) such that cities c1 and c2 buy exactly the same set of products.[1] We represent a set of products by the list of its ProdID values, separated by commas. The list of products of each city is created through the `group_concat` agregate function:

```
select City,group_concat(ProdID,',') from BUY group by CustID
```

For each couple of (distinct) cities, we compare their product list through function `itemEqual`. Since the product list must be computed twice, we define it in a CTE (Script 24.3).

```
with CP(City,PSet) as
    (select City,group_concat(ProdID,',')
     from   BUY group by City)
select C1.City,C2.City
from   CP C1, CP C2
where  C1.City <> C2.City
and    itemEqual(C1.PSet,C2.PSet,',');
```

**Script 24.3 -** Finding couples of cities that buy the same product set

Finally, we display the groups of cities that all buy the same product set (Script 24.4). For this, we add the product list (column PSet) to each couple of *equal* cities in table EQUAL, derived from Script 24.3. The city groups are then built by concatenating the city names grouped by their product set. To order and to avoid duplicate city names, we use the special SQLfast function `group_concat2`.

The result is shown in Figure 24.2. It has been computed on a variant of ORDERS.db in which several cities have assigned the same product set.

The complete script of this application is available in file **SQLfast-List-3.sql** in directory Scripts/SQLfast-Tutorials/Chapter-24.

1. A special case of the inclusion relation: city c1 buys all the products of city c2 and city c2 buys all the products of city c1.

```
openDB ORDERS.db;

create temp table BUY as
   select distinct City,ProdID
   from   CUSTOMER C, CUSTORDER O, DETAIL D
   where  O.CustID = C.CustID and D.OrdID = O.OrdID;

create temp table EQUAL
with CP(City,PSet) as
    (select City,group_concat(ProdID,',')
     from   BUY group by City)
select C1.City as City1,C2.City as City2,
       itemSort(C1.PSet,0,0,',') as PSet
from   CP C1, CP C2
where  C1.City <> C2.City
and    itemEqual(C1.PSet,C2.PSet,',');

select PSet as Products,
       group_concat2(City1,1,0,1,',') as Cities
from   EQUAL
group by PSet;
```

**Script 24.4 -** Creating groups of cities buying the same product sets

```
+------------------+------------------------+
| Products         | Cities                 |
+------------------+------------------------+
| CS262,PA45       | Bruxelles,Genève,Paris |
| CS262,PA60,PS222 | London,Toulouse        |
| PA45             | Lille,Poitiers         |
+------------------+------------------------+
```

**Figure 24.2 -** The groups of cities buying the same product sets

Chapter **25**

# Dynamic columns and schema-less tables

**Objective**: This chapter explores the concept of *schema-less table*. Such a table is made up of columns dynamically created by `insert` and `update` SQL-DML queries, and for that, called *dynamic columns*. They opposed to static tables that comprise columns (called *static columns*) created by `create table` and `alter table` SQL-DDL queries. Dynamic columns are particularly fit to store data about entities with no standard structures and the properties of which cannot be determined in schema creation time. The SQLfast environment offers a simple and intuitive extension to create and use dynamic columns. This extension is compared with the dynamic columns of MariaDB.

**Keywords**: dynamic column, schema-less table, JSON, MariaDB

**Statements and functions**: getDColNames, group_DColNames, getDCol, setDCol

**System variables and parameters**: DynaColumns, dynaMarker, DColumns

## 25.1 Introduction

The usual procedure for creating a table uses the `create table` query of the SQL-DDL sub-language:

```
create table ITEM(ItemID integer not null primary key,
                  Name   varchar(20) not null);
```

The insertion of data must conform to the type of the columns that make up the table:

```
insert into ITEM(ItemID,Name) values
       (123,'SCREW (45-2)',
       (124,'NAILS (120-3.5)');
```

If a new column is required, it must be created by an SQL-DDL query that modifies the schema of the table:

```
alter table ITEM add column Color char(20);
```

After that, it is possible to insert lines according to this new format:

```
insert into ITEM(ItemID,Name,Color) values
       (138,'BRUSH AD8','BLUE');
```

In all pre-existing rows, the new column has a *null* value.

## 25.2 Dynamic columns

For some application domains, however, this disciplined procedure is considered excessively strict. When the facts to be recorded in a table have properties which are very variable and impossible to identify in advance, one can no longer construct the schema of this table a priori. Practically, we would like the table schema to evolve dynamically according to the data we insert. This flexibility in the evolution of the scheme is the objective of *dynamic columns*.

This type of column allows us to write and execute the following script, whose `insert`, `update` and `select` queries reference undeclared columns:

```
create table ITEM(ItemID integer not null primary key,
                  Name   varchar(20) not null);

insert into ITEM(ItemID,Name,Color,Price)
       values (138,'BRUSH AD8','BLUE',2.50),
              (150,'TOOLBOX','GREY',18);
```

```
update ITEM
set    Color = 'BLACK',
       Price = 10
where  Color is null;

select ItemID,Name,Color,Supplier
from   ITEM
where  Color = 'BLUE';
```

Columns Color, Price and Supplier are *dynamic columns*. They are automatically created the first time they are mentioned in an SQL query. The columns created by the `create table` and `alter table` statements are called *static*.

A table consisting of a static column declared `primary key` and in which all other columns are dynamic can be considered as a *table without schema* or *schema-less table*.

## 25.3 Practical syntax

Dynamic SQLfast columns are implemented as a layer developed above the SQL engine and not as an extension of this engine. This architecture imposes some constraints:

– For preprocessing efficiency reasons[1], it is necessary to *mark* dynamic column names with a specific prefix. Thus, we will write **!Color** instead of **Color**. The default marker '**!**' can be changed if needed: it is defined by system variable dynaMarker, which can be set in initialization file SQLfast.ini or, dynamically, in scripts.

– Dynamic columns are supported (preprocessed) if system variable dynaColumns is set to 1. They are ignored if this variable is set to 0, which is its default value. This variable is assigned in SQLfast.ini file:

```
dynaColumns = 1
```

or in the script itself:

```
set dynaColumns = 1;
```

1. For some parts of some queries, identifying dynamic column names is fairly straightforward: they simply do not appear in dictionary table SYS_COLUMN. For other parts however, their identification would require a complete syntactic analysis of the extended SQL query. Considering the richness and complexity of the SQL language, this analysis would be quite expensive, all the more so as this analysis will also be performed by the SQL engine. This would contradict with the objectives of a modest scripting language such as SQLfast!

– SQLfast does not store metadata for dynamic columns. References to a dynamic column in a query are identified as the word following the '**!**' marker (or its substitute). This technique can cause ambiguities if the marker also appears in SQL constants. In this case, it is recommended to replace the marker with an appropriate symbol.

The script above should therefore be written as follows:

```
create table ITEM(ItemID integer not null primary key,
                  Name   varchar(20) not null);
insert into ITEM(ItemID,Name,!Color,!Price)
       values (138,'BRUSH AD8','BLUE',2.50),
              (150,'TOOLBOX','GREY',18);
update ITEM
set    !Color = 'BLACK',
       !Size = 'MEDIUM'
where  !Color = 'BLUE';
select ItemID,Name,!Color,!Supplier
from   ITEM
where  !Color = 'BLACK';
```

The syntax of the `set` clauses of the `update` query is a bit more limited than that of pure SQL:

the right part of each clause denotes either a string constant (starting with an apostrophe), or a column name, or a numeric constant or any kind of expression between parentheses.

*Example*

```
update ITEM
set    !Color = 'BLACK',
       !Model = 12.5,
        Name  = (upper(Name) || '!'),
       !Size  = (!Size * 1.05),
       !Price = (select Price from PRODUCT
                 where  ProdID = ITEM.ItemID)
where  !Color = 'BLUE';
```

## 25.4 Metadata of dynamic columns

The list of dynamic column names of a row is obtained by (UDF) SQL function getDColNames(sep), where sep is the name separator. Following the execution of the script above, the query:

```
select getDColNames(',') as Columns
from   ITEM where ItemID = 150;
```

will provide the following result:

```
+-------------+
| Columns     |
+-------------+
| Color,Price |
+-------------+
```

To obtain the list of names of *all* the dynamic columns of a table, we will use the group_DColNames(sep) aggregate function:

```
select group_DColNames(',') as Columns from ITEM;
```

the result of which will be:

```
+------------------+
| Columns          |
+------------------+
| Color,Price,Size |
+------------------+
```

The idea of a small application immediately comes to mind: listing the contents of a table that includes dynamic columns. This is what the script below does:

```
extract dCol = select '!'||group_DColNames(',!') from ITEM;
select ItemID,Name,$dCol$ from ITEM;
```

`dCol` is the comma-separated list of the names of all the dynamic columns in table ITEM, prefixed with the dynamic marker:

```
dCol = !Color,!Price,!Size
```

Result of this script:

```
+--------+-----------------+-------+-------+--------+
| ItemID | Name            | Color | Price | Size   |
+--------+-----------------+-------+-------+--------+
| 123    | SCREW (45-2)    | BLACK | 10    | --     |
| 124    | NAILS (120-3.5) | BLACK | 10    | --     |
| 138    | BRUSH AD8       | BLUE  | --    | MEDIUM |
| ...    | ...             | ...   | ...   | ...    |
+--------+-----------------+-------+-------+--------+
```

This small exercise shows how to display static and dynamic columns when the latter are not known by the programmer. The procedure is of course much simpler when the columns to display are known in advance:

```
select ItemID,Name,!Color,!Price,!Size from ITEM
```

## 25.5 Technical note

The way dynamic columns are implemented deserves a word of explanation. As soon as a dynamic column is mentioned in an `insert` or `update` query, a technical column named DColumns is added to the table by an `alter table ITEM add column` ... query. This column contains a Json object whose components represent the values of all the dynamic columns of the row. For example, the value of DColumns of row ItemID = 123 is expressed as follows:

```
{"Color":"BLACK","Price":10}
```

Let us consider *schema-less* table CUST defined as follows:

```
create table CUST(CustID char(12) not null primary key);
```

We insert in this table some rows made up of data of table CUSTOMER of our reference database, formatted as dynamic columns.

```
insert into CUST(CustID,!Name,!City,!Cat,!Account)
       select CustID,Name,City,Cat,Account from CUSTOMER;
```

To visualize the contents of table CUST, we just execute this query, to get the result of Figure 25.1:

```
select CustID,!Name,!City,!Cat,!Account from CUST;

+--------+-----------+-----------+-----+---------+
| CustID | Name      | City      | Cat | Account |
+--------+-----------+-----------+-----+---------+
| B112   | HANSENNE  | Poitiers  | C1  | 1250    |
| C123   | MERCIER   | Namur     | C1  | -2300   |
| B332   | MONTI     | Genève    | B2  | 0       |
| F010   | TOUSSAINT | Poitiers  | C1  | 0       |
| K111   | VANBIST   | Lille     | B1  | 720     |
| S127   | VANDERKA  | Namur     | C1  | -4580   |
| B512   | GILLET    | Toulouse  | B1  | -8700   |
| B062   | GOFFIN    | Namur     | B2  | -3200   |
| C400   | FERARD    | Poitiers  | B2  | 350     |
| C003   | AVRON     | Toulouse  | B1  | -1700   |
| K729   | NEUMAN    | Toulouse  | --  | 0       |
| F011   | PONCELET  | Toulouse  | B2  | 0       |
| L422   | FRANCK    | Namur     | C1  | 0       |
| S712   | GUILLAUME | Paris     | B1  | 0       |
| D063   | MERCIER   | Toulouse  | --  | -2250   |
| F400   | JACOB     | Bruxelles | C2  | 0       |
+--------+-----------+-----------+-----+---------+
```

**Figure 25.1 -** Contents of table CUST viewed through the dynamic column interface

The following query will show the implementation of these dynamic columns (Figure 25.1):

```
select * from CUST
```

| CustID | DColumns |
|---|---|
| B112 | {"Name":"HANSENNE","City":"Poitiers","Cat":"C1","Account":[..] |
| C123 | {"Name":"MERCIER","City":"Namur","Cat":"C1","Account":-2300} |
| B332 | {"Name":"MONTI","City":"Genève","Cat":"B2","Account":0} |
| F010 | {"Name":"TOUSSAINT","City":"Poitiers","Cat":"C1","Account":0} |
| K111 | {"Name":"VANBIST","City":"Lille","Cat":"B1","Account":720} |
| S127 | {"Name":"VANDERKA","City":"Namur","Cat":"C1","Account":-4580} |
| B512 | {"Name":"GILLET","City":"Toulouse","Cat":"B1","Account":-8700} |
| B062 | {"Name":"GOFFIN","City":"Namur","Cat":"B2","Account":-3200} |
| C400 | {"Name":"FERARD","City":"Poitiers","Cat":"B2","Account":350} |
| C003 | {"Name":"AVRON","City":"Toulouse","Cat":"B1","Account":-1700} |
| K729 | {"Name":"NEUMAN","City":"Toulouse","Account":0} |
| F011 | {"Name":"PONCELET","City":"Toulouse","Cat":"B2","Account":0} |
| L422 | {"Name":"FRANCK","City":"Namur","Cat":"C1","Account":0} |
| S712 | {"Name":"GUILLAUME","City":"Paris","Cat":"B1","Account":0} |
| D063 | {"Name":"MERCIER","City":"Toulouse","Account":-2250} |
| F400 | {"Name":"JACOB","City":"Bruxelles","Cat":"C2","Account":0} |

**Figure 25.2 -** Contents of table CUST viewed through the standard SQL interface

When system variable DynaColumns is set to 1, the SQLfast engine processes SQL queries that include instances of dynamic marker dynaMarker. Through this process, the queries are converted into pure SQL queries. Setting and extracting the values of dynamic columns are done through two functions **setDCol** and **getDCol**, that work on a flat Json object, here below denoted by parameter `container`:

- `setDCol(container,col,val)`: returns Json object `container` in which an element with name `col` and value `val` has been added. If such an element already exists, changes its value. If `val` is *null*, the element is deleted.
- `getDCol(container,col)`: returns the value of the element named `col` in Json object `container`. Returns *null* if such an element does not exist.

## 25.6 Performance: static *vs* dynamic columns

To evaluate the penalty of managing data dynamically, we load 160,000 rows of data in a temporary table created in database ORDERS.db. The source data are the rows of table CUSTOMER, loaded 10,000 times.

In the first experiment (Script 25.1), temporary table CUST is declared with static columns CustID, Name, City, Cat and Account. In the second experiment (Script 25.2), table CUST is made up of one static column, CustID, which serves as its primary key. Columns Name, City, Cat and Account are created dynamically.

```
openDB ORDERS.db;

create temp table CUST(CustID  char(10) not null primary key,
                       Name    char(20) not null,
                       City    char(20) not null,
                       Cat     char(2),
                       Account decimal(10,2) not null);
start-timer;
for N = [1,10000]
    insert into CUST(CustID,Name,City,Cat,Account)
    select '$N$'||CustID,Name,City,Cat,Account from CUSTOMER;
write Static time = $timer$ s.;
```

**Script 25.1 -** Loading 160,000 rows in static table CUST

```
openDB ORDERS.db;

create temp table CUST(CustID char(10) not null primary key);

start-timer;
for N = [1,10000]
    insert into CUST(CustID,!Name,!City,!Cat,!Account)
    select '$N$'||CustID,Name,City,Cat,Account from CUSTOMER;
write Dynamic time = $timer$ s.;
```

**Script 25.2 -** Loading 160,000 rows in dynamic table CUST

Both scripts run in about 8 seconds, with a slight difference:

```
Static time = 7.867 s.
Dynamic time = 7.909 s.
```

This difference is not really significant (0.5%) and does not depend on the volume of the data loaded.

## 25.7 Dynamic columns in DBMS

Dynamic columns are getting popular, particularly in NoSQL data managers. Let us examine how they are implemented in MariaDB.

MariaDB is, in some sense, the current Open source version of MySQL. Dynamic columns values are collected in a BLOB column that must be explicitly declared at table create time.

```
create table ITEM(ItemID   integer not null primary key,
                  Name     varchar(20) not null
                  DColumns BLOB);
```

*SQLfast equivalent:*

```
create table ITEM(ItemID integer not null primary key,
                  Name   varchar(20) not null);
```

They are created, updated and queried through eight special functions.

- Dynamic columns are created through function CREATE_COLUMN:

```
insert into ITEM(ItemID,Name,DColumns)
       values(138,'BRUSH AD8',
              COLUMN_CREATE(DColumns,'Color','BLUE',
                                     'Price',2.50));
```

*SQLfast equivalent:*

```
insert into ITEM(ItemID,Name,!Color,!Price)
       values(138,'BRUSH AD8','BLUE',2.50));
```

- Dynamic values can be extracted with function COLUMN_GET:

```
select ItemID,
       COLUMN_GET(DColumns,'Color' as char)  as Color,
       COLUMN_GET(DColumns,'Price' as float) as Price
from ITEM;
```

*SQLfast equivalent:*

```
select ItemID,!Color,!Price from ITEM;
```

- The existence of a dynamic value in a row is checked with function COLUMN_EXISTS:

```
select ItemID,Name
from   ITEM
where  not COLUMN_EXISTS(DColumns,'Color')
```

*SQLfast equivalent:*

```
select ItemID,Name from ITEM where !Color is null;
```

- To add or replace a dynamic value, one uses function `COLUMN_ADD`:

```
update ITEM
set    DColumns = COLUMN_ADD('DColumns,Color','BLACK',
                             'Price',2.5)
where  COLUMN_GET(DColumns,'Color') = 'BLUE';
```

*SQLfast equivalent:*

```
update ITEM
set    !Color = 'BLACK', !Price = 2.5
where  !Color = 'BLUE';
```

- Function `COLUMN_DELETE` allows a dynamic value to be removed from selected rows:

```
update ITEM
set    DColumns = COLUMN_DELETE('DColumns','Price')
where  COLUMN_GET(DColumns,'Color') = 'BLUE';
```

*SQLfast equivalent:*

```
update ITEM set !Price = null where !Color = 'BLUE';
```

- Function `COLUMN_LIST` returns the list of dynamic column names of the current row:

```
select ItemID,COLUMN_LIST(DColumns) as Columns from ITEM;
```

*SQLfast equivalent:*

```
select ItemID,getDColNames(',') as Columns from ITEM;
```

Function `COLUMN_CHECK` checks the validity of the BLOB container (here, DColumns) and `COLUMN_JSON` returns the contents of the container converted into a Json object.

Note that there is no equivalent of aggregate function `group_DColNames` in MariaDB.

## Suggested references

- Dynamic Columns in MariaDB [https://mariadb.com/kb/en/library/dynamic-columns/]
- Dynamic Columns Tutorial – Part 1: Introduction [https://mariadb.com/fr/node/469]

– Dynamic Columns Tutorial – Part 2: Searching and Updating [https://mariadb.com/fr/node/455]

## 25.8 Summary

SQLfast provides features to allow scripts to add *dynamic columns* to a table, that is, columns that are dynamically created through DML queries

*Static columns* are created by DDL queries: `create table` and `alter table add column`. *Dynamic columns* can be created by `insert` queries, in both `insert-values` and `insert-select` variants. They can also be created by `update` queries.

Dynamic columns can be referenced by (i.e., their name may appear into) the following statements:

– **`extract`**: data can be extracted from dynamic columns

– **`create view`**: a view can be defined on a table containing dynamic columns

– **`insert values`**: values can be assigned to dynamic columns; note that only constants may appear in the value list

– **`insert select`**: the source select expression may provide values derived from static and dynamic columns

– **`update`**: values can be assigned to dynamic columns; the source data can be derived from static and dynamic columns

– **`select`**: data can be extracted from both static and dynamic columns; if an element of the select-list is a dynamic column reference mentioned without alias, an alias is automatically added.

Any reference to a dynamic column must be prefixed with a special dynamic marker. The default marker is the exclamation character (!). It can be changed by assigning a new value to system variable `dynaMarker`.

Dynamic columns are processed by the SQLfast engine if system variable `dynaColumns` is set to 1. Otherwise, their references are ignored.

For performance sake, dynamic column references are preprocessed through a simplified syntactic analysis. Therefore, the processing of some complex expressions (notably those the constants of which comprise dynamic markers) may fail and lead to execution errors. In such cases, it is recommended to temporarily change the dynamic marker.

# Chapter 26

# Developing a tutorial

**Objective**: This chapter describes SQLtuto, the SQLfast sub-language devoted to the development of tutorials. A tutorial is a formatted text that can include pictures, hyperlinks, audio and video messages as well as executable code. This language allows the creation of various kinds of tutorials, ranging from purely static texts to multimedia documents and to documents merging a fully featured application and its (technical or user) documentation.

**Keywords**: tutorial, help document, SQLtuto language, tutorial engine, tutorial window, multimedia, formatted text, bitmap picture, audio message, video message, code snippet, script, embedded script, wrapper

**Statements and functions**: .titleN:, .text0:, .text:, .bold:, .italic:, .bold-italic:, .center:, .itemN:, itemN:, .indentN,:, .code:, .titlecode:, .link:, .image:, .tutoImageMaxSize, .figure:, .script:, .endscript:, .scriptMaxHeight:, .char-Size:, .loadscripts:, displayHelp, closeHelp, execSQL, execProg

## 26.1 Introduction

Tutorials (and help documents twins) are an essential part of the SQLfast environment. Not only do they help users to master the various components of SQLfast, in particular its languages, functions and interfaces, but they also support the didactic objective of SQLfast. To clarify these two categories of objectives, let us say that the

first one includes the so-called *help documents*, while the others are the *actual tutorials*, in the traditional sense of the word.

Each *help document* is assigned a proper name that is mentioned in the **Help** menu: *Getting started*, *Survival guide*, *SQLfast commands*, etc. There are structural components of the SQLfast environment and there is no way to add or remove a help document but anybody can modify them.

*Tutorials* normally are created to teach students to write SQL queries or to write database application programs. Considering the scope of the SQLfast environment, a series of tutorial have already been written and integrated in the distribution. Of course, they can be modified or removed and other tutorials can be created at will. Beyond their primary objective to teach SQL and database programming, tutorials can be developed to help users to run application programs written in SQLfast.

The only difference between help documents and tutorials is the extension of their source files: respectively *.help and *.tuto. From now on, the term **tutorial** will denote both **help documents** and **tutorials**.

Tutorials are written in a dedicated language, *SQLtuto*. This chapter is an introduction to this language and to the various usages one can make of it. In particular, we will learn how to develop complete, documented, applications in SQLtuto.

## 26.2 Opening a tutorial

Opening a tutorial consists in displaying the contents of an SQLtuto file in a tutorial window. It appears as a formatted text that may include pictures, code snippets, embedded scripts and links to other tutorials. The source of a tutorial is a text file that controls its appearance through SQLtuto commands. These commands specify the format of the text, the location of the pictures, the code (snippets and embedded scripts) and the links.

The tutorial window comprises several buttons, among which, **Close**, the effect of which is obvious. The others will be described later.

A tutorial can also be opened from the SQLfast main window: through the Help button at the Basic level and the Help menu at the Expert level. In addition, a tutorial can be opened from within a script through command **`displayHelp`** and closed either manually or through command **`closeHelp`**.

## 26.3 A tutorial is a formatted text

The source text of a tutorial is a sequence of **paragraphs**. A paragraph can be prefixed with a **command**, that specifies how it will appear in the tutorial window. Most commands comprise a keyword enclosed between a dot (`.`) and a semi-colon (`:`), plus zero, one or more parameters separated by a semi-colon.

Some examples:

```
.italic:This text will appear in italic.
.center:This one will be centered.
.item1c:SQL: is the most widespread database language.
```

A line the first two characters of which are `'--'` is a comment and therefore is ignored.

### General formats

Let symbol `<text>` denote any paragraph.

**`<text>`**
: same as **`.text:<text>`**. A blank or empty line is a valid paragraph.

**`.text:<text>`**
: Displays `<text>` as a simple text including blank lines. Left aligned. Font verdana 9, 1 mm before.

**`.text0:<text>`**
: Same as **`.text:`** but no space before.

**`.bold:<text>`**
: Same as **`.text:`** in bold.

**`.italic:<text>`**
: Same as **`.text:`** in italic.

**`.bolditalic:<text>`**
: Same as **`.text:`** in bold and italic.

**`.center:<text>`**
: Same as **`.text:`** centered.

**`.code:<text>`**
: Text displayed in fixed width font (courier 9) with a left margin of 6 mm, no space before. Generally used to display program code and query result set.

**`.charSize:n`**
: **`n`** is a number from 0 to 3 specifying the character sizes. The default value is 1, which corresponds to the sizes indicated in the format descriptions (that is, 9, 10, 11 and 12 points).

  Other values: 0 = minus one point (8, 9, 10, 11); 2 = plus 1 point (10, 11, 12, 13); 3 = plus 2 points (11, 12, 13, 14).

  The size specified by this command is global for the whole document. It is best to place it at the beginning of the document.

**`.`**
: A single dot creates a small blank line (courier 6). Size not modified by **`.charSize:`**. Used to create small space between paragraphs.

### Titles

SQLtuto provides four levels of titles that are used to give the text of the tutorial its gross structure.

**`.titleN:<text>`**

where **`N`** is a number from 1 to 4. Displays `<text>` as a *level-**N*** title.

**`.title1:`** helvetica 12, bold, centered, 3 mm before, 3 mm after.

**`.title2:`** helvetica 11, bold, left aligned, 3 mm before, 1 mm after.

**`.title3:`** helvetica 10, bold, left aligned, 3 mm before, 1 mm after.

**`.title4:`** helvetica 10, bold italic, left aligned, 3 mm before, 1 mm after.

**`.titlecode:<text>`**

Text displayed in fixed width font (courier 9) in bold. 2 mm before. No left margin. Generally used to start the description of a language element.

**Remark**: there is no automatic numbering.

### Indented paragraphs

SQLtuto provides four levels of indented paragraphs, useful to create a hierarchy among a sequence of paragraphs.

**`.indentN:<text>`**

where **`N`** is a number from 1 to 4. Displays `<text>` with a left margin of Nx3 mm. 2 mm before.

### Itemized paragraphs

SQLtuto provides four levels of itemized paragraphs, that is, lists of items.

**`.itemN:<text>`**

Same as **`.indentN:`** preceded by a dash character.

**`.itemNc:<header>:<text>`**

Same as `.itemN:`. The text comprises `<header>` in fixed width font (courier 9) in bold, followed by `<text>`.

### Example

Script 26.1 illustrates the use of some of the formatting commands described above. It creates the document of Figure 26.1

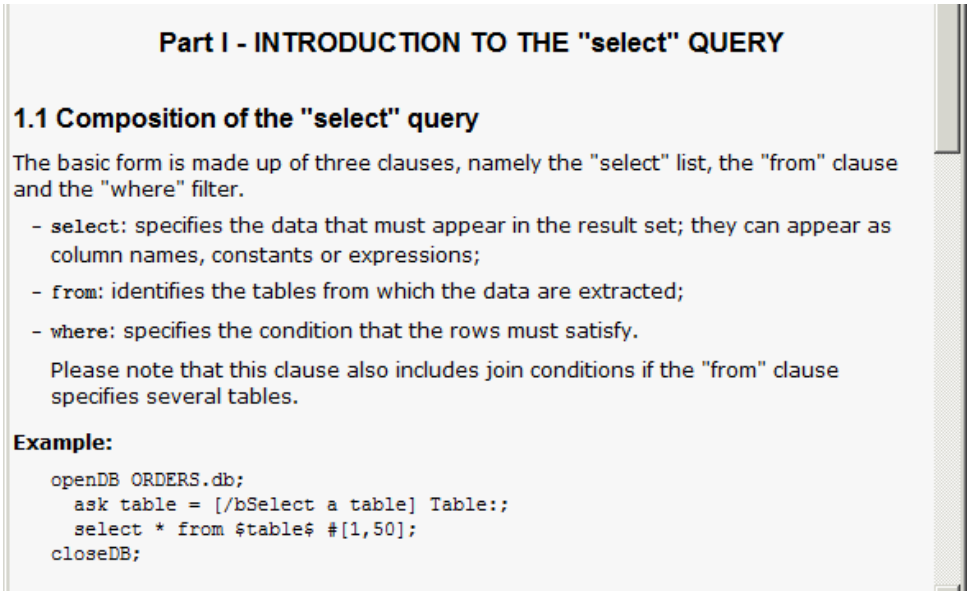
```
.title1:Part I: INTRODUCTION TO THE "select" QUERY

.title2:1.1 Composition of the "select" query
The basic form is made up of three clauses, namely the "select"
list, the "from" clause and the "where" filter.
--
.item2c:select:: specifies the data that must appear in the
result set; they can appear as column names, constants or
expressions;
--
.item2c:from:: identifies the tables from which the data are
extracted;
--
.item2c:where:: specifies the condition that the rows must
satisfy.
.indent2:Please note that this clause also includes join
conditions if the "from" clause specifies several tables.

.bold:Example:

.code:openDB ORDERS.db;
.code:  ask table = [/bSelect a table] Table:;
.code:  select * from $table$ #[1,50];
.code:closeDB;
```

**Script 26.1 -** A script creating a structured document



Part I - INTRODUCTION TO THE "select" QUERY

1.1 Composition of the "select" query

The basic form is made up of three clauses, namely the "select" list, the "from" clause and the "where" filter.

- select: specifies the data that must appear in the result set; they can appear as column names, constants or expressions;
- from: identifies the tables from which the data are extracted;
- where: specifies the condition that the rows must satisfy.

  Please note that this clause also includes join conditions if the "from" clause specifies several tables.

**Example:**

```
openDB ORDERS.db;
  ask table = [/bSelect a table] Table:;
  select * from $table$ #[1,50];
closeDB;
```

**Figure 26.1 -** A structured document

## 26.4 Adding images

An image is inserted into a tutorial through command `.image:` that specifies by `<address>` the address of a file containing a bitmap image in jpeg (or jpg) or gif format:

**`.image:<address>`**

Displays the image contained in file `<address>`; the default directory for images is `<hdir>/Images`, where `<hdir>` is the help directory.

**`.tutoImageMaxSize:n`**

To reduce the images that are too large, a max size of `n` pixels can be specified (default is set by system variable `tutoImageMaxSize`). The image will be forced to be inscribed in an `n`x`n` square.

**`.figure:<text>`**

To add a caption, the insertion of the picture will be followed by command `.figure:` that specifies the text of the legend (`<text>`):

These commands are illustrated in Script 26.2, that creates the tutorial fragment of Figure 26.2.

```
.bolditalic:Remark
.indent2:Each menu item can be detached to form a floating tool
bar (Figure 4).

.image:Lev1-Help/Lev1-Detachable-Menu.gif
.figure:4 - Detachable menu
```

**Script 26.2 -** Inserting an image and its caption

## 26.5 Navigating within and between tutorials

The tutorial subsystem of SQLfast provides easy ways to navigate within a tutorial and to connect (and navigate between) the documents of a collection of tutorials.

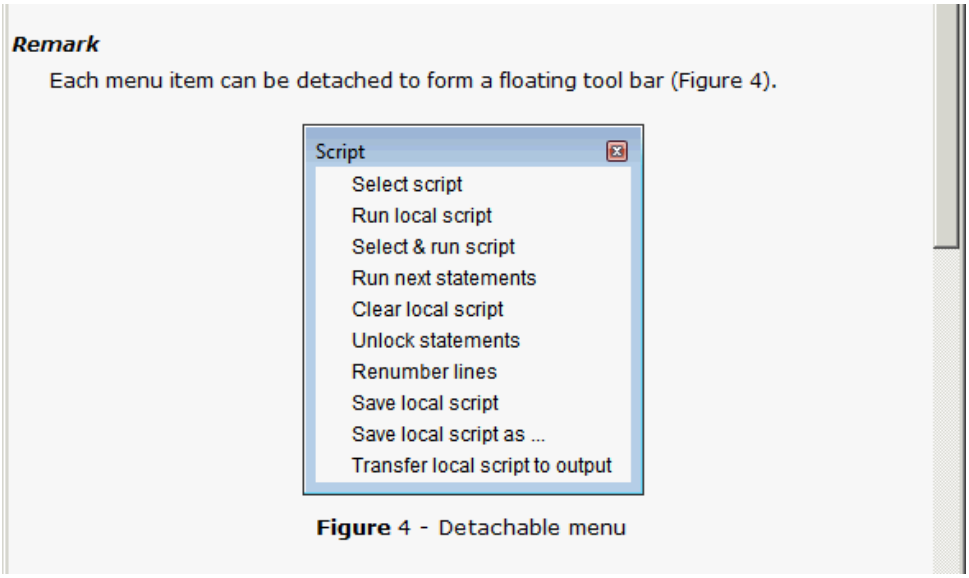



**Figure 26.2 -** Image inserted in a tutorial

## Internal navigation

Like web pages, a tutorial may be quite large, so that users may want to get quick access to some of its part. For example, the **Survival guide** tutorial that can be opened from the Help menu is more than sixty page long (if printed) and comprises about fifty sections. Browsing through the whole document just to read Section 32 (32. Error management) would be discouraging. The SQLtuto window allows us to jump to a definite part of the document by a simple and direct action: by *double-clicking anywhere into a paragraph*, the window jumps to the *next instance of this paragraph*.

A tutorial that includes a series of exercises (as do most of the SQL tutorials included in the SQLfast distribution) will often be made up of two parts: the first one presents the statements of the exercises while the second one suggests and discusses their solutions. If both the statement and the solution are identified by a title such as the following:

**Exercise 4.17 - Grouping data**

double-clicking on the title of the statement of exercise 4.17 will immediately show its solution. Then, double-clicking of the latter title will bring us back to the statement title in the first part.

The **Survival guide** tutorial starts with a *table of contents* that collects all the section titles (e.g., **32. Error management**). It allows us to jump to any section with the same technique.

### External navigation

A tutorial can include references to other tutorials through an external link mechanism.[1] The link is created through the `.link:` command:

```
.link:<address>:<text>
```

Parameter `<address>` specifies the path of the external tutorial and parameter `<text>` is the text of the link that will appear in the tutorial. The link is clearly identified: its text appears in *bold* and in *blue colour*.

The use of the link command is illustrated in Script 26.3, that creates the fragment of Figure 26.3.

By clicking on the text of the link, the current tutorial is replaced in the tutorial window by the tutorial referenced by the link (in our example, file Lev1_Tutorials/9._More-on-loops.tuto). Backward navigation is performed through button **Back** of the tutorial window, that displays the source document. This button can be activated several times if necessary.

```
This concludes this lesson. Now, on the way to the next chapter:
.
.link:Lev1_Tutorials/9._More-on-loops.tuto: 9. Looping - The
final chapter
```

**Script 26.3 -** Creating a link to another tutorial

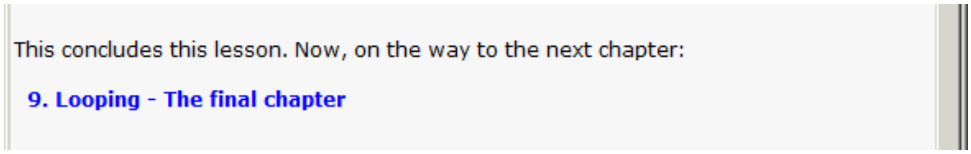



**Figure 26.3 -** A link to another tutorial

## 26.6 Code snippets

All the tutorials and most help documents of the SQLfast distribution include *code snippets* intended to illustrate the use of SQL or SQLfast languages. They generally are created by a sequence of `.code:` commands that give the code a clear format, as shown by Figure 26.1 for example.

Beyond their use as mere illustration, code snippets enjoy a very interesting property: *they can be executed*. When a sequence of code lines is selected (the first and

1. Similar to HTML hyperlinks created with the `<a> </a>` tags.

last lines can be partially selected), two actions are proposed through buttons of the tutorial window:

– button **Transfer**: the selected lines are appended to the contents of the script area of the main SQLfast window, where they can be executed.

– button **Run**: the selected lines are directly executed.

This is an important feature of SQLfast tutorials: they now can be called **active**, since they include **executable code**.

Moreover, a tutorial is **updatable**: lines can be dynamically added, modified and deleted by the user. This is also true for code snippets, that can be modified or completed in order to get different results. For instance, the user can be invited to replace a line left blank in an SQL query by a clause that makes this query valid. However, these modifications are transient, since they are not propagated to the source text of the tutorial.

## 26.7 Embedded scripts

Like code snippets, *embedded scripts* are executable sequences of SQL or, more generally, SQLfast statements. Their differences lie in their properties and in the way they are executed.

The central section of Figure 26.4 shows an embedded script as it is rendered in an SQLtuto document. Its code is that of the snippet of Figure 26.1 but its layout is quite different.

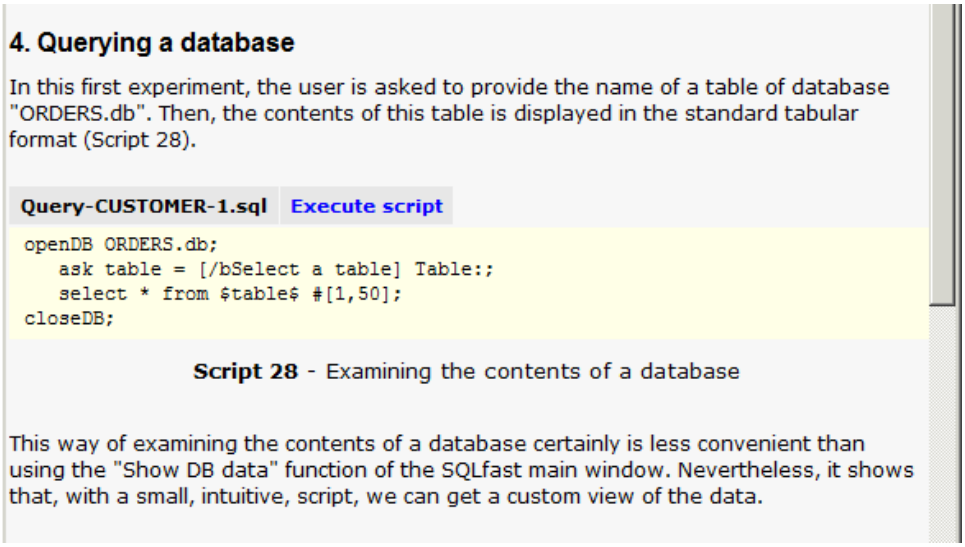



**Figure 26.4 -** An embedded script in its environment

As any embedded script, this one is made up of five components:

- the script identifier (Query-CUSTOMER-1.sql);
- the execution button (with label Execute script);
- the body, comprising a sequence of four statements;
- a reference (e.g., a number) through which it can be referenced from within the document (28), just like a figure;
- a caption describing the function of the script (Examining the contents of a database).

This script is specified by the SQLtuto code of Script 26.4. The code of the script is enclosed between commands `.script:` and `.endscript:`.

```
.script:Query-CUSTOMER-1.sql:Execute script
openDB ORDERS.db;
   ask table = [/bSelect a table] Table:;
   select * from $table$ #[1,50];
closeDB;
.endscript:28:Examining the contents of a database
```

**Script 26.4 -** General structure of an embedded script

**`.script:<scriptId>:<execLabel>`**

This command has two optional parameters, the *script identifier* (`<scriptId>`) and the label of the *execution button* (`<execLabel>`).

The ***script identifier*** (script Id) is unique among all the embedded scripts of the document. If it is specified, it can be used to call the script from another embedded script, provided it ends with characters `.sql`. A length can be specified as follows: `/25Query-CUSTOMER-1.sql`. With a length of `0` (`/0Query-CUSTOMER-1.sql`), the Id is defined but is not displayed.

If the script Id is missing, the engine creates an internal Id. In such a case, the script Id is not displayed.

This ***execution button***, when clicked on, submits the body to the SQLfast engine for execution. The second parameter (`<execLabel>`) specifies the label of the execution button. If this label is missing, the default label **Run** is used. A length can also be specified: `/18Execute script`. With a length of `0` (`/0Execute script`, or simpler, `/0`), the button is not created.

**`.endscript:<scriptRef>:<caption>`**

This command closes the script body and generates the legend of the script. It has two optional parameters, the *script reference* (`<scriptRef>`) and the *caption* (`<caption>`). Both are arbitrary characters strings that need not be unique. If both of them are missing, no legend is generated.

The rendering of an embedded script is also controlled by command `.scriptMaxHeight:`, that specifies the number of lines of the window in which the body of the script is displayed. The default height is 50. For larger scripts, a scrollbar appears. If a length of 0 is specified, the script body itself is created but is not displayed.

## 26.8 Embedded script variants

With appropriate parameter settings, several embedded script variants can be built.

In most cases, the simplified presentation of embedded scripts shown in Script 26.5 will be considered sufficient. It generates the view of Figure 26.5, in which the script Id and the legend are missing.

```
.script:
openDB ORDERS.db;
   ask table = [/bSelect a table] Table:;
   select * from $table$ #[1,50];
closeDB;
.endscript:
```

**Script 26.5 -** Creating a simplified display of an embedded script

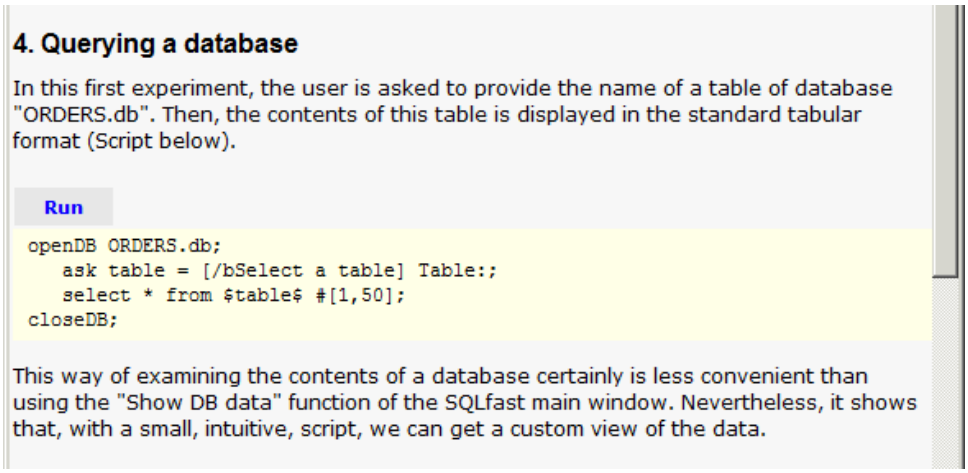


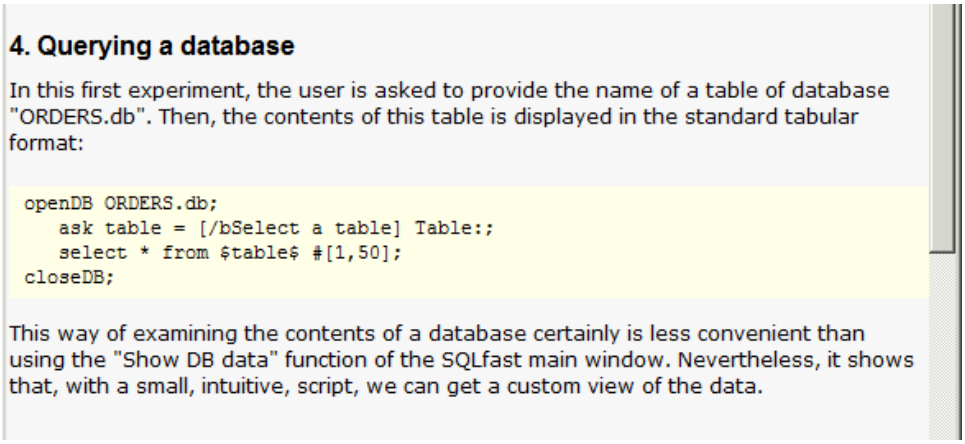

**Figure 26.5 -** Simplified display of an embedded script

If a script is shown as a pure illustration that is not intended to be executed, no execution button is generated thanks to the zero-length parameter (Script 26.6).

```
.script:/0
ask <target_variable> = [<message>] <label>;
select * from $<target_variable>$ #[<range>];
.endscript:
```

**Script 26.6 -** Static script illustration: only the script body is visible

The code of Script 26.7 creates an embedded script in its ultimate presentation: it does not appear in the document at all (Figure 26.7). It is given Id `Query-CUSTOMER-1.sql`, through which other scripts of the document can call it. This pattern can be used to create a library of procedures that need not to be visible by its users.



**4. Querying a database**

In this first experiment, the user is asked to provide the name of a table of database "ORDERS.db". Then, the contents of this table is displayed in the standard tabular format:

```
openDB ORDERS.db;
   ask table = [/bSelect a table] Table:;
   select * from $table$ #[1,50];
closeDB;
```

This way of examining the contents of a database certainly is less convenient than using the "Show DB data" function of the SQLfast main window. Nevertheless, it shows that, with a small, intuitive, script, we can get a custom view of the data.

**Figure 26.6 -** Static embedded script

```
.scriptMaxHeight:0
Between this text line,
.script:/0Query-CUSTOMER-1.sql:/0
openDB ORDERS.db;
   ask table = [/bSelect a table] Table:;
   select * from $table$ #[1,50];
closeDB;
.endscript:
and this one, there is a hidden embedded script.
```

**Script 26.7 -** This script is *hidden* but still exists and can be called by other embedded scripts

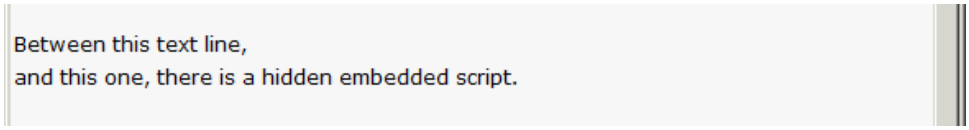



**Figure 26.7 -** *Where is Wally?* Find the hidden script!

## 26.9 Embedded scripts as a programming system

An embedded script can comprise any SQL or SQLfast statement, included those that call - that is, that *ask for the execution of* - another script (`execSQL`), a Python application (`execApp`) or function (`function`), or an external program (`execProg`).

### 26.9.1 Calling an embedded script

An `execSQL` statement calls a script generally stored somewhere in the SQLfast/Scripts directory.

An important feature of embedded scripts is that they also can call other embedded scripts of their tutorial document.[2] This can be done under the following conditions:

– an embedded script can be called if it has been assigned a script Id which ends with characters `.sql` (Figure 26.4);

– when an embedded script is called through an `execSQL` statement, its name must be prefixed with name '`local.`' to avoid name confusion with external scripts stored on disk. For instance, calling embedded script `Query-CUSTOMER-1.sql` must be written:

```
execSQL local.Query-CUSTOMER-1.sql;
```

– to be *callable*, the embedded scripts of the tutorial must have been *prepared* through command `.loadscripts:`. This command, that appears at the end of the SQLtuto script, materializes the callable embedded scripts by storing them as temporary files in directory SQLfast/Scripts. These files are automatically deleted when the tutorial is closed.

2. Inter-document calls are not allowed (yet)

The code of Script 26.8 creates a fragment of an introductory tutorial to *Data analytics*. It comprises three embedded scripts:

- The main script creates a database, then calls local script `CreateSchema.sql` and finally calls local script `LoadData.sql`. Since it is not intended to be called, it has no explicit identifier but is given an execution button.
- Local scripts `CreateSchema.sql` creates the tables of the database. It has no execution button (it is not intended to be executed by the user).
- Local scripts `LoadData.sql` insert some initial data into these tables. It also doesn't have an execution button.

The aspect of the resulting tutorial is shown in Figure 26.8.

The `.loadscripts:` command creates a script file for each of the last two embedded scripts.

```
.title2:5. Creating the ANALYTICS database
The data collected from the sales transactions will be stored
in database "ANALYTICS.db" that describes the "regions" and the
"transactions". To create this database and to store some
representative data, just execute the Script 12.

.script::Create ANALYTICS database
createOrReplaceDB ANALYTICS.db;
  execSQL local.CreateSchema.sql;
  execSQL local.LoadData.sql;
closeDB;
.endscript:12

.script:CreateSchema.sql:/0
create table AN_REGION (
  RegID integer not null primary key, REGName varchar(64));
create table AN_TRANSACTION (
  TransID integer not null primary key, TransDate date not null);
commitDB;
.endscript:

.script:LoadData.sql:/0
insert into AN_REGION values (8,'East Midlands');
insert into AN_REGION values (10,'South East');
commitDB;
.endscript:

.loadscripts:
```

**Script 26.8 -** Creating a system of interconnected embedded scripts

### 5. Creating the ANALYTICS database

The data collected from the sales transactions will be stored in database "ANALYTICS.db" that describes the "regions" and the "transactions". To create this database and to store some representative data, just execute Script 12.

**Create ANALYTICS database**

```
createOrReplaceDB ANALYTICS.db;
  execSQL local.CreateSchema.sql;
  execSQL local.LoadData.sql;
closeDB;
```

**Script 12**

**CreateSchema.sql**

```
create table AN_REGION (
  RegID integer not null primary key, REGName varchar(64));
create table AN_TRANSACTION (
  TransID integer not null primary key, TransDate date not null);
commitDB;
```

**LoadData.sql**

```
insert into AN_REGION values (8,'East Midlands');
insert into AN_REGION values (10,'South East');
commitDB;
```

**Figure 26.8 -** Excerpts from the Data analytics tutorial

### 26.9.2 Minimizing script internals

Two comments that potential users could formulate on this tutorial fragment:

- "few of us will be interested by *the body* of the embedded scripts,"
- moreover, "many of us will probably be disturbed by *the very existence* of the two called scripts."

Hence the idea of a *minimized view* in which only the components relevant to the users' need are shown (i.e., the execution button of the main script) and all the technical components are hidden. Actually, the users will be certainly quite happy with the simplified view of Figure 26.9. The modified SQLtuto code is shown in Script 26.9.

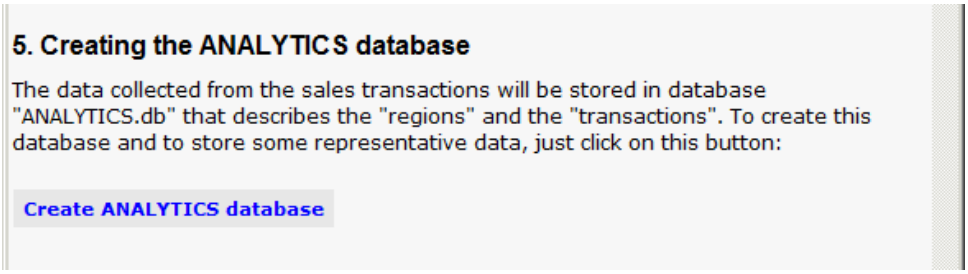



**Figure 26.9 -** A minimized, user-friendly version of the Data analytics tutorial

```
.title2:5. Creating the ANALYTICS database
The data ..., just click on this button:

.scriptMaxHeight:0

.script::Create ANALYTICS database
 ...
.endscript:

.script:/0CreateSchema.sql:/0
 ...
.endscript:

.script:/0LoadData.sql:/0
 ...
.endscript:

.loadscripts:
```

**Script 26.9 -** The SQLtuto code that creates the tutorial of Figure 26.9

## 26.10 Inserting video and audio messages

SQLtuto does not comprise specific commands to launch a video clip or an audio message, as the SQLfast language can do. However, the same result can easily be achieved by embedding the corresponding SQLfast instructions (Script 26.10 and Figure 26.10)

```
.scriptMaxHeight:0
.script::Date's Introduction to SQL (click)
 execProg $VIDEOprocessor$/SQLintro.avi;
.endscript:
```

**Script 26.10 -** A minimal script to start a video clip

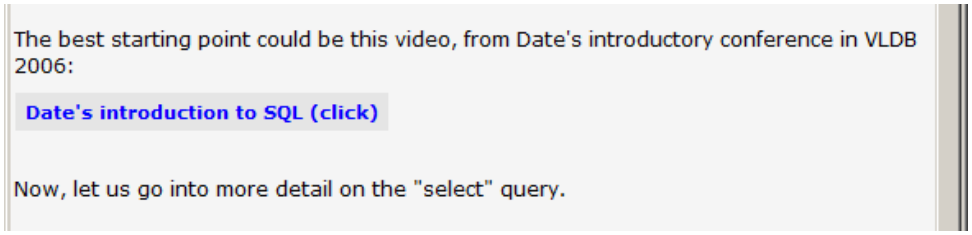



**Figure 26.10 -** Launching a video clip (the reference to Date's keynote is fictitious!)

## 26.11 Embedded wrappers

In all the examples of embedded code examined in the previous section, we can say that *what you see is what is inside*, except in one variant, when `.scriptMaxHeight:0` has been specified to hide the script body.

The fact that all the parts of an embedded script can be shown or hidden can lead to very interesting programming constructs, among them, *wrappers*.

Before defining what wrappers are, and showing how they can be exploited, we will examine the basic mechanism on which they rely. Let us consider the embedded script as it appears in Figure 26.11. It apparently comprises an execution button controlling a body that displays the rows of table CUSTOMER. If we click on this button, the contents of table CUSTOMER should be displayed in the output window of SQLfast. And indeed, it works that way!




**Figure 26.11 -** A (seemingly) simple embedded script

Actually this figure has been build as **two embedded scripts**. Of the first one, only the *execution button* is visible while we only see the *body* of the second one.

The second script has been given a script Id, `selectFromCUSTOMER.sql`, that makes it callable but this Id is hidden. Since this is the script the user sees, we will call it the *user script*.

The body of the first script, called the *wrapper*, is quite simple: just calling the second script. Hence the SQLtuto code of this construct shown in Script 26.11.

In this elementary example the wrapper does nothing really useful but it illustrates the architecture pattern that we will exploit here below.

```
-- The wrapper script --
.scriptMaxHeight:0
.script::List CUSTOMER data
execSQL local.selectFromCUSTOMER.sql;
.endscript:
--
-- The user script   --
.scriptMaxHeight:50
.script:/0selectFromCUSTOMER.sql:/0
openDB ORDERS.db;
   select * from CUSTOMER;
closeDB;
.endscript:
```

**Script 26.11 -** Building an elementary wrapper

Now, let us assign more responsibility to our wrapper. We suppose that the reader of the tutorial just starts learning databases and is not yet aware of the fact that before extracting data from a table, one must first open the database[3], and that, once the job done, the database must be closed. Practically, what the reader should see is nothing more than Figure 26.12. Of course, opening and closing the database now is the responsibility of the wrapper (Script 26.12).

This short example illustrates one of the objectives of SQLtuto wrappers: to present to the user a simplified view of a task by hiding its real complexity.




**Figure 26.12 -** Simplified user's view of data extraction

The second example is more elaborate. It shows how a wrapper can protect inexperienced users by catching their errors. The idea is to let users insert data in a table and to react in a *friendly* way to the mistakes they might have made.

The fragment of Figure 26.13 is an excerpt of a tutorial on database concepts specifically addressing data integrity. It includes two scripts: the first one creates a database and a table while the second script inserts some sample rows in this table.

3. All the more so since the opening command often must specify such complex parameters as isolation mode, locking policy, transaction technique (WAL or shadow paging for example) or cache management.

```
-- The wrapper script --
.scriptMaxHeight:0
.script::List CUSTOMER data
openDB ORDERS.db;
  execSQL local.selectFromCUSTOMER.sql;
closeDB;
.endscript:
--
-- The basic script   --
.scriptMaxHeight:50
.script:/0selectFromCUSTOMER.sql:/0
select * from CUSTOMER;
.endscript:
```

**Script 26.12 -** Now, the task of the wrapper is more complex!

This is quite simple and intuitive, until we observe that *the second script includes an error*: the second row insertion will violate the uniqueness constraint induced by the primary key of the table.

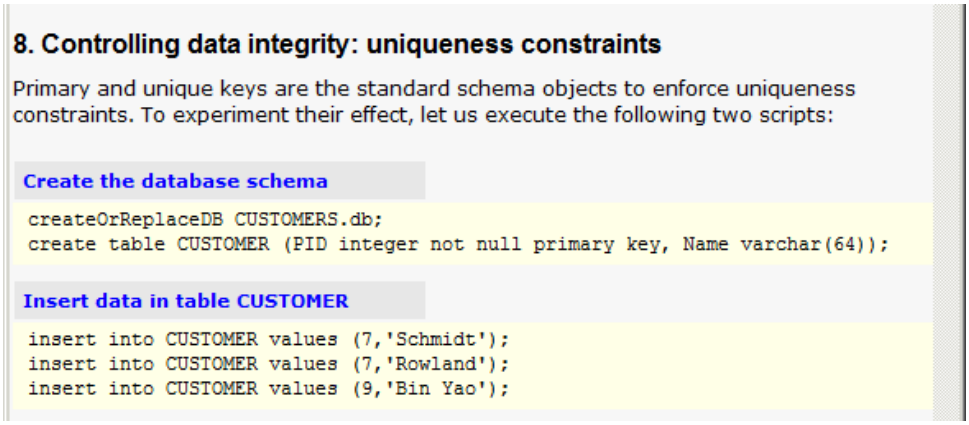



**Figure 26.13 -** In this exercise, the user is invited to provide data to insert in table CUSTOMER

When processing the second `insert` query, the execution will stop, a detailed error message will pop up, the same message will also appear in the output window and a rollback operation will cancel all the data modification carried out in this script, including the (successful) insertion of the first row. This is the standard way the SQLfast engine copes with script errors at the *Expert* level.[4]

We, the developers of this tutorial, consider that this complex reaction is not particularly appropriate for novice readers who may feel disoriented by this excessively *technical* and *verbose* behaviour. On the contrary, we would like to take the lead when an error is detected in order to give the user *more friendly* information.

While the code that creates the first script is quite simple (Script 26.13), the code of second one is structured as a wrapper (Script 26.14, hidden) followed by the user script it controls (Script 26.15, visible).

```
.script:/22Create the database schema
createOrReplaceDB CUSTOMERS.db;
create table CUSTOMER (PID integer not null primary key,
                       Name varchar(64));
.endscript:
```

**Script 26.13 -** The code of the first script: creating the database schema

The wrapper follows a particular approach: it does not execute the user script through an `execSQL` command, but it reads its instructions one at a time and interprets them in a secure way. We decide that this objective will be achieved by implementing the following rules:

- only uniqueness violations are coped with, the other errors being ignored (processing them in the same way is left as an exercise),
- if an `insert` does not raise a uniqueness error, it is automatically and definitively committed, whatever the current commit mode,
- when a uniqueness error occurs, the user is informed in precise but non technical English words,
- when all the statements of the user script have been executed, the (valid) contents of table CUSTOMER is displayed,
- the diagnostic is not displayed in the output window but in a custom text window depending on the script (this means that both the main and output windows, that may disturb novice users, can be minimized),

The wrapper of Script 26.14 translates these rules.

4. The default commit mode is *transaction*, that requires the execution of a `commitDB` instruction to actually perform the data modifications.

```
.scriptMaxHeight:0
.script:/22Insert data
openDB CUSTOMERS.db;
outputOpen diag.var;
for insert = [file $scriptDirectory$/local.CUSTOMER-DATA.sql];
  onError continue;
  $insert$;
  if ('$SQLdiag$' <> 'OK');
     if ('$SQLdiag$' = 'ID');
        write-b Primary key uniqueness error in query:
                @n@t"$insert$";
     else;
        write-b "$SQLdiag$" error in query:
                @n@t"$insert$";
     endif;
  else;
     commitDB;
  endif;
  onError stop;
endfor;
write;
select * from CUSTOMER;
showText diag = [/x76/y10];
outputAppend window;
closeDB;
.endscript:
```

**Script 26.14 -** The (hidden) code of the wrapper

The result of the execution of both scripts is shown in Figure 26.14

```
.scriptMaxHeight:50
.script:/0CUSTOMER-DATA.sql:/0
insert into CUSTOMER values (8,'Schmidt');
insert into CUSTOMER values (8,'Rowland');
insert into CUSTOMER values (10,'Bin Yao');
.endscript:

.loadscripts:
```

**Script 26.15 -** The (visible) code of the user script

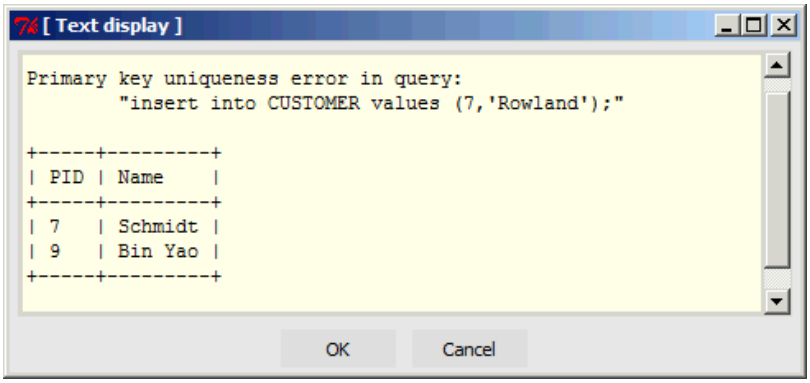



**Figure 26.14 -** Result of the execution of user script 26.15 through wrapper 26.14

## 26.12 What is the role of a tutorial, after all?

The tutorial subsystem of SQLfast can be used to develop a wide variety of products, ranging from simple tutorials to complete, standalone applications embedded in their user interface. Let us briefly describe some of them.

### 26.12.1 The tutorial as ... a tutorial!

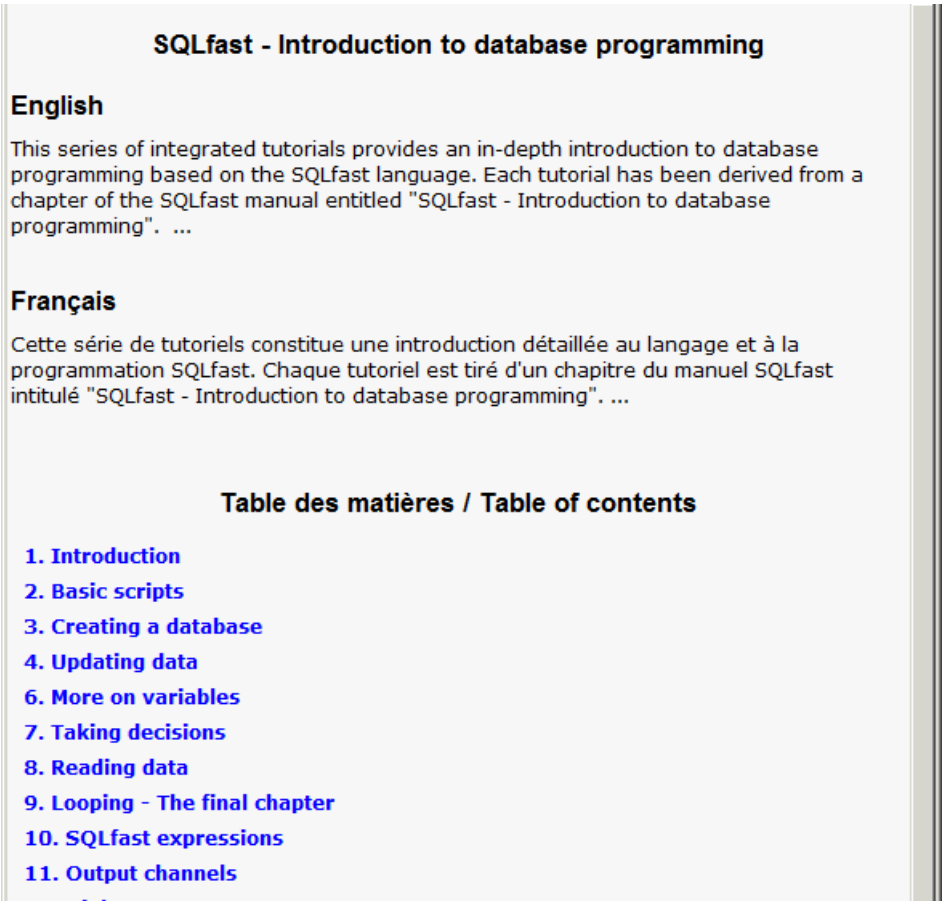
A tutorial is a document the primary role of which is to introduce, step by step, to the use of some resource (a text or image processor for example) or to solve some kind of problems (e.g., how to improve a low quality picture). They exist in various forms: paper or electronic illustrated documents, web pages with multimedia components, video clips, etc.

Most of the tutorials integrated in the SQLfast distribution are of that kind[5] in the scope of databases, SQL and database programming.

A simple tutorial will be implemented by a single SQLtuto document. For wider topics the tutorial generally is divided into separate but interconnected didactic units, forming a *hyper document*. By convention, its entry point comprises a short introduction, some preliminaries and a table of contents, the latter being made of a sequence of links (`.link:`) to more detailed didactic units (Figure 26.15). Each didactic unit may include

- a link to the next didactic unit
- a link to the table of contents
- if and where needed, links to more detailed documents.

5. Though this can change in future versions if the new features (of version 4) are exploited.



**SQLfast - Introduction to database programming**

**English**

This series of integrated tutorials provides an in-depth introduction to database programming based on the SQLfast language. Each tutorial has been derived from a chapter of the SQLfast manual entitled "SQLfast - Introduction to database programming". ...

**Français**

Cette série de tutoriels constitue une introduction détaillée au langage et à la programmation SQLfast. Chaque tutoriel est tiré d'un chapitre du manuel SQLfast intitulé "SQLfast - Introduction to database programming". ...

**Table des matières / Table of contents**

1. Introduction
2. Basic scripts
3. Creating a database
4. Updating data
6. More on variables
7. Taking decisions
8. Reading data
9. Looping - The final chapter
10. SQLfast expressions
11. Output channels

**Figure 26.15 -** Excerpt from tutorial *Lev1_Tutorials/0. SQLfast-tutorial.tuto*

A didactic unit can be given many different structures depending on the target audience, the nature of the concepts described, the objective of the tutorial, its underlying pedagogical principles or the prerequisite knowledge. In the domain of *SQL learning*, a typical structure could be the following:

- a title (+ last update date, author, etc.)
- a short introduction to the topic.
- a table of contents
- a short reminder of the concepts addressed in the tutorial (provided that these concepts have already been described elsewhere).
- a list of exercise statements that the reader is invited to solve. For a **`select`** query, each statement comprises the description, in natural language, of the data to extract from the database and a print of the resultset to produce. The reader is invited to write an SQL query that produces this resultset.

– a list of hints to help readers to find a solution. For complex exercises, a second level of hints could be needed.

– a list of solutions, possibly commented.

– a list of additional resources (such as inter-tutorial links).

Jumping to a definite component of the tutorial is easily performed by the *internal navigation* feature, notably:

– from the table of contents to a section.

– from an exercise title to the corresponding hint and to the suggested solution.

– from the solution of an exercise back to its statement.

### 26.12.2 The tutorial as a case study narrative

A series of case studies are available on the SQLfast web site.[6] Each of them describes a data oriented problem (actually a class of problems) and develops a solution in terms of database structures and SQL queries. The final result is a complete interactive application comprising a database and a set of scripts.

The main objective of these case studies is twofold. First, to show that many problems can be solved easily and in a concise way through database-oriented reasoning and technology. Second, to elaborate, most often informally and through examples, some guidelines (I wouldn't say *methodology*) useful to solve other problems of this kind.

To be convincing and therefore useful, these guidelines must not be asserted in an authoritarian manner (they are no *cookery recipes*), but rather derived from an intellectual journey, both rational and critic. So, many sections of these case studies include trial and error episodes. In some sense, such a study *tells a story.*

Now, what is the link between case studies and tutorials? This story is published on the web site as a *static pdf document*. This document is a formatted text material, interspersed with code fragments, full scripts and pictures, that can be immediately translated in SQLtuto. This tutorial could then by enhanced by integrating some dynamic features of SQLtuto:

– a table of contents for internal navigation

– some code snippets can be executed,

– some scripts can be embedded and therefore are executable

– links to other SQLtuto documents

– adding references to external pdf documents, web sites, audio and video messages.

6. https://projects.info.unamur.be/~dbm/mediawiki/index.php?title=LIBD:Outils#SQLfast

As an example, let us consider the first case study published on the SQLfast web site, entitled *Four hours to save the library*. It illustrates the so-called *Rapid Application Development* concept, RAD in short, by writing in less of 80 lines of code (and in about 4 hours[7]) an elementary but fully functional, library management application.

This static pdf document has been translated in SQLtuto. Figure 26.16 shows a page of this tutorial. It is important to note that this tutorial includes the narrative of the case study but also most of its scripts in an executable format. Therefore, it results from the **merging** into a single document of the static text of the case study with the contents of the Scripts\Case-Studies\Case_Small_Library_1 directory. The reader will find here the materials of this case study:

– the *static pdf text* of the case study at **https://staff.info.unamur.be/dbm/Documents/Tutorials/SQLfast/SQLfast-Case01-Small-library.pdf**,

– the *SQLfast scripts* of the case study in directory **SQLfast\Scripts\Case-Studies\Case_Small_Library_1** of the SQLfast distribution

– and the *narrative tutorial* we are talking about, named **The-small-library-(Case-Study).tuto**, in directory **SQLfast\SQLfastHelp_FR\Case_studies_Tutorials** of the SQLfast distribution

### 26.12.3 The tutorial as a technical documentation

Let us suppose that a case study leads to a full scale application, ready to use by a community of users.[8] It is not unreasonable to think that this application will include a sophisticated graphical interface, database management functions (such as backup and recovery) and a non trivial hierarchy of functions. We also can expect that users will soon ask for additional functions and that, here and there, a bug will be discovered. In short, this application will need to be *maintained* and this maintenance, to be reliable and inexpensive, will require a **precise, up-to-date documentation**.

Basically, most aspects of this documentation should be present in the case study narrative, while the latter would not be appropriate to support the maintenance task. Hence the idea of a (possibly somewhat extended) stripped down version of this case study narrative that only addresses the technical aspects of the application necessary to rapidly spot the places where the code modifications must be performed.

7. A professional programmer (what I'm not) would probably have written this code in much less time!

8. Let us be more specific. The final step of most case studies is the production of a *prototype*, not an end product sufficiently robust to be distributed. However, some prototypes may be considered mature enough to be adopted as a useful tool. The *Directory Management* program could be considered ready to use (as I do!)

## 5. The main window

The main window is the control panel of the application. It lets the user select a function (Figure 1). Button OK starts the selected function while button Cancel closes the application.

Each function is implemented as an independent procedure. These procedures are called from the body of a while-endwhile loop by a series of if statements (Script 2). This writing style is not particularly efficient from the point of view of programming science (all the conditions are evaluated in each execution of the body of the loop), but it provides a clear and concise code, which is essential in this experiment.

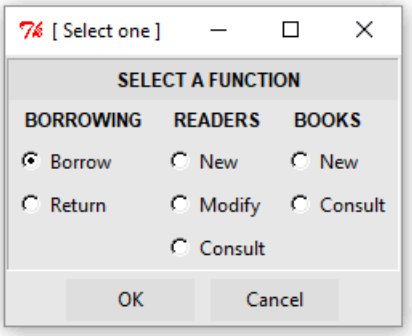



**Figure** 1 - The main window lets the user select a function

**LIBRARY-Main.sql**

```
openDB LIBRARY-small.db;
set oper = 1;
while (True);
  selectOne-u oper = [/b@S20SELECT A FUNCTION]
                     {BORROWING}|Borrow|Return
                   ||{READERS}|New|Modify|Consult
                   ||{BOOKS}|New|Consult;
  if ('$DIALOGbutton$' = 'Cancel') exit;
  if ($oper$ = 1) execSQL local._LIB-BORROWING-BORROW.sql;
  if ($oper$ = 2) execSQL local._LIB-BORROWING-RETURN.sql;
  if ($oper$ = 3) execSQL local._LIB-READER-NEW.sql;
  if ($oper$ = 4) execSQL local._LIB-READER-MODIFY.sql;
  if ($oper$ = 5) execSQL local._LIB-READER-CONSULT.sql;
  if ($oper$ = 6) execSQL local._LIB-BOOK-NEW.sql;
  if ($oper$ = 7) execSQL local._LIB-BOOK-CONSULT.sql;
endwhile;
closeDB;
```

**Script 2** - Code of the control panel

**Figure 26.16 -** Excerpt of a Case study converted into an SQLtuto document

### 26.12.4 The tutorial as a standalone application

This is the ultimate view of the application that has been built by the case study.

It consists of tutorial document that embeds but hides all the scripts of the application except those that implement user oriented functions, for which only the execution button is visible (Figure 26.17).

The tutorial can also includes the user's guide, either in-line (as in Figure 26.17) or as external tutorials. Note that it is not recommended to call such auxiliary tutorials through `.link:` commands since they will replace the application document, which will spoil the current work. Instead, one must open an independent help window as follows:

- Create an embedded script comprising only SQLfast statement `displayHelp`.
- Of this script, only the execution button is visible.
- Assign this button the name **Help**.

## 26.13 Color settings

The standard background colors of the help documents and of the embedded scripts can be rendered quite differently depending on the screen settings. They can be customized by two system variables:

- `colorHelp`: specifies the background color of the help documents (default grey96)
- `colorHelpScript`:s pecifies the background color of the embedded scripts (default light yellow)

Their standard values are set in initialization file SQLfast.ini:

```
colorHelp = grey96
colorHelpScript = light yellow
```

However, they can also be modified by a script:

```
set colorHelp = #FFFFFF;
set colorHelpScript = light green;
```

## 26.14 Summary

Short description of the commands of the SQLtuto language.

### Basic formatting commands

**.** Small blank line (courier 6).

**.text:<text>**

Displays `<text>` as a simple text. Small space before.

**<text>**
Same as **.text:**<text>.

**.text0:<text>**
Displays <text> as a simple text. No before space.

**.bold:<text>**
Same as **.text:** in bold.

**.italic:<text>**
Same as **.text:** in italic.

**.bolditalic:<text>**
Same as **.text:** in bold italic.

**.center:<text>**
Same as **.text:**, centered.

**.code:<text>**
Fixed width font. Indented.

**.charSize:n**
Set character scale (0 to 3)

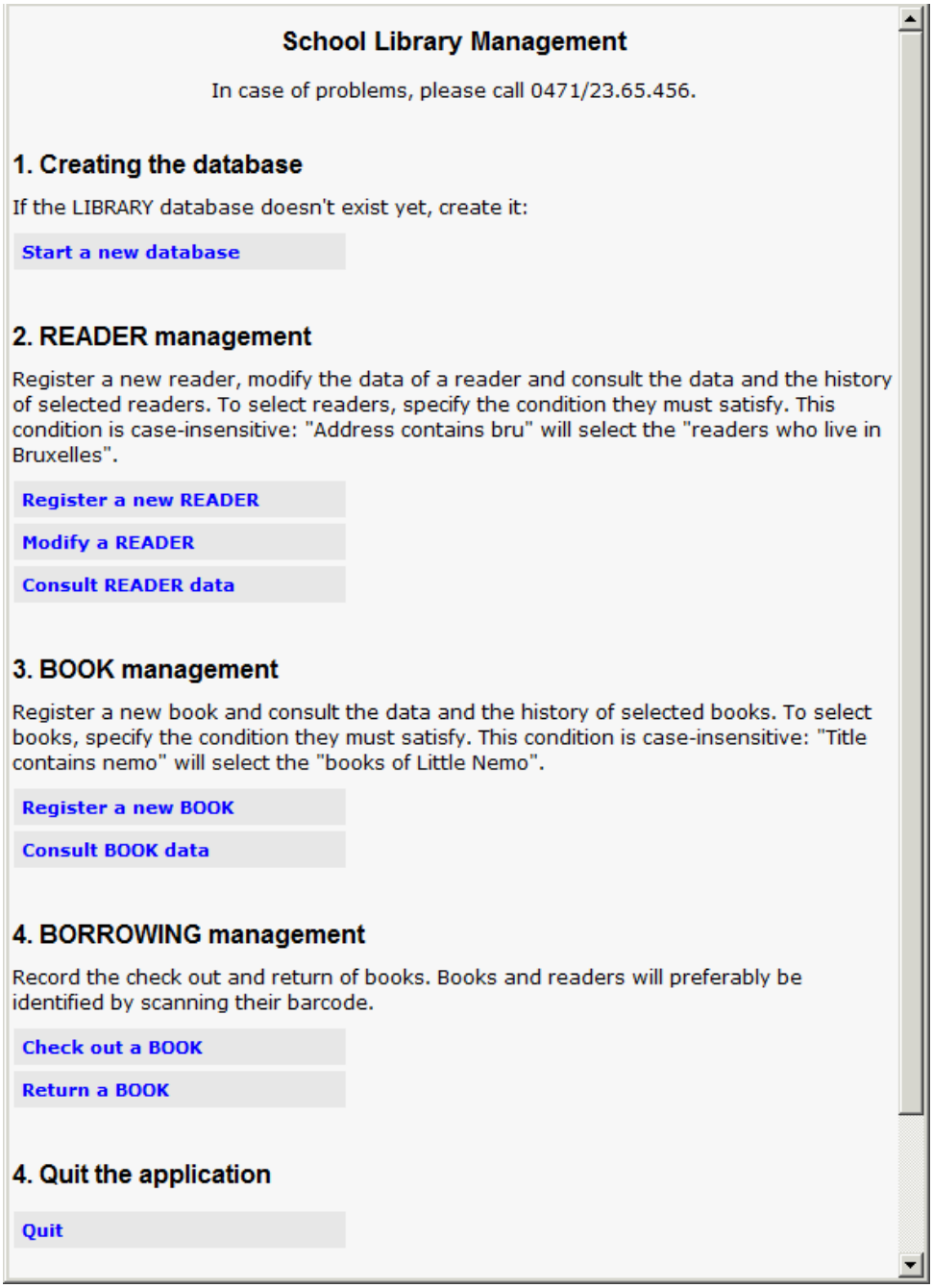



**Figure 26.17 -** The user interface of the case study final application

### Titles

**.titleN:<text>**
Displays <text> as a level-N title.

**.titlecode:<text>**
Fixed width font. Bold.

### Indented paragraphs

**.indentN:<text>**
Displays <text> with a left margin of Nx3 mm

### Itemized paragraphs

**.itemN:<text>**
Same as **.indentN:** preceded by a dash character.

**.itemNc:<header>:<text>**
Same as **.itemN:** The text comprises <header> in fixed width font, in bold, followed by <text>.

### Adding images

**.image:<address>**
Inserts a picture from a file.

**.tutoImageMaxSize:n**
Sets maximum size of images.

**.figure:<text>**
Displays a legend.

### Internal navigation

Double-clicking in a paragraph jumps to the next instance of this paragraph.

### External navigation

**.link:<address>:<text>**
Creates a link to another tutorial.

### Code snippets

Selecting a series of text lines then clicking on button **Transfer** adds these lines to those in the script area of the main SQLfast window. By clicking on button **Run** instead, these lines are executed.

## Embedded scripts

**.script:<scriptId>:<execLabel>**
Starts an embedded script. Generates the script Id and the execution button. See detail below.

**.endscript:<scriptRef>:<caption>**
Closes the embedded script and generates the legend. See detail below.

**.scriptMaxHeight:n**
Specifies the maximum of lines of the body window.

**.loadscripts:**
creates a script file for each embedded scripts the Id of which has prefix *.sql.

## Embedded scripts: the script header

Patterns of the script header.

| Commands | Result |
|---|---|
| **.script:** | default script id (not visible);<br>default execution button label = **Run** |
| **.script:**CREATE_DB.sql | script id = CREATE_DB.sql (visible);<br>default execution button label = **Run** |
| **.script:**:Execute | default script id (not visible);<br>custom execution button label = **Execute** |
| **.script:**CREATE_DB.sql:Execute | script id = CREATE_DB.sql (visible);<br>custom execution button label = **Execute** |
| **.script:/16**CREATE_DB.sql:Execute | script id = CREATE_DB.sql (spanning 16 characters);<br>custom execution button label = **Execute** |
| **.script:/0**CREATE_DB.sql:Execute | script id = CREATE_DB.sql (not visible);<br>custom execution button label = **Execute** |
| **.script:**CREATE_DB.sql:/8Go! | script id = CREATE_DB.sql (visible);<br>custom execution button label = **Go!** (spanning 8 characters) |
| **.script:**CREATE_DB.sql:/0 | script id = CREATE_DB.sql (visible);<br>no execution button created |
| **.script:/0**CREATE_DB.sql:/0 | script id = CREATE_DB.sql (not visible);<br>no execution button created |

### Embedded scripts: the script body

Patterns of the script body.

| Commands | Result |
|---|---|
| **.script:**<br><SQLfast statements><br>**.endscript:** | All the SQLfast statements are visible |
| **.scriptMaxHeight:**12<br>**.script:**<br><SQLfast statements><br>**.endscript:** | Only the first 12 SQLfast statements are visible |
| **.scriptMaxHeight:**0<br>**.script:**<br><SQLfast statements><br>**.endscript:** | The script body is not visible |

### Embedded scripts: the script legend

Patterns of the script legend creation.

| Commands | Result |
|---|---|
| **.endscript:** | |
| **.endscript:**12 | **Script 12** |
| **.endscript:**:Create the database | Create the database |
| **.endscript:**12:Create the database | **Script 12** - Create the database |

## 26.15 Technical complement: about *wrappers*

*<This section is more technical and can be skipped>*

The term *wrapper* is not a specialty of SQLfast tutorials but is a popular software structure in application development. A wrapper is a software layer, most often a library made up of data structures and functions, intended to convert those of a lower level resource. For this, it *wraps* or *encapsulates* the resource to hide it to the developer's view. Through this conversion, the wrapper is intended to provide services similar to those of the resource while adding some improvement. But it is best to illustrate this abstract definition with some representative examples in the database realm:

– ODBC (Open Database Connectivity) and its Java variant JDBC (Java Database Connectivity) each tries to offer software developers a unified view of the SQL language on top of most RDBMS.[9] Application programs access the database through the ODBC (or JDBC) driver (the *wrapper*) instead of the native interface of the underlying DBMS (the *resource*). The DB-API interface offers similar functionality for the Python language. This make application programs easier to port to another DBMS.

– ORM (Object-Relational Mapping) provides an object-oriented interface (through a *wrapper*) to relational databases (the *resource*). They provide a better integration of database programming protocols into object-oriented languages. Object-Relational Mappers are available for most programming languages.

– Access to legacy databases. One usually call *legacy database* a database developed several decades ago, implemented in obsolete technologies, but still in use and often supporting critical functions in their organization. They should be converted into modern tools (relational or NoSQL DBMS) but such a conversion appears to be both very expensive and risky.

  An alternative way to keep exploiting a legacy database is to maintain it as it is but adding on top of it a module simulating a modern interface, relational or object-oriented for example. This module is a *wrapper* through which modern applications access and update legacy data (the *resource*) through the new interface, while old applications (also qualified *legacy*) continue to access the same data through the native interface.

## What does the literature say?

The definition provided by *Wikipedia*[10] is so general that it really is useless: *A wrapper function is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine [...]. Wrapper functions can be used to make writing computer programs easier.*

The *TechTerms* web site (The Tech Terms Computer Dictionary) is a bit more informative[11]: *In computer science, a wrapper is any entity that encapsulates (wraps around) another item. Wrappers are used for two primary purposes: to convert data to a compatible format or to hide the complexity of the underlying entity using abstraction.*

The popular *stackoverflow* forum has been consulted on this topic. One of the answers describes the real things quite clearly[12]: *The vast majority of wrappers exist*

9. ODBC/JDBC wrappers also exist to access data in simple formatted text and CSV files, though with limited functionality.
10. https://en.wikipedia.org/wiki/Wrapper_function
11. https://techterms.com/definition/wrapper
12. https://stackoverflow.com/questions/3293752/where-and-how-is-the-term-used-wrapper-in-programming-what-does-it-help-to-do

*to hide some sort of complexity. [...] Wrappers may expose different features than the underlying object. [...] You'd write a wrapping class if you wanted to simplify things. You'd create an easy to use 'wrapper' and leave all the complicated bits inside the wrapper, ensuring that those complicated bits are not 'exposed'.*

## About tutorial wrappers

Now, does the code of Script 26.14 implement a real *wrappers*? Let us consider that the *resource* to wrap is the SQL native interface, with its powerful but complex features, and that the *user script* is what the user needs to see. What the SQLtuto wrapper does is,

- Managing the complexity of inserting insecure data into a table.
- Catching SQL errors and translating them into messages that can be understood by novice users.
- Executing the SQL commands that are valid but rejecting erroneous ones without rolling back the whole transaction.
- Therefore providing the user with a more natural and friendly behaviour than that of the native interface.
- While protecting the integrity of the data.

It is therefore not unreasonable to call Script 26.14 the code of a *wrapper* in the proper sense of the concept, though in a particular context!

# **A**ppendix **1**

# A basket of SQLfast examples

This appendix illustrates the use of SQL and SQLfast to solve a series of small problems.

While some examples are quite simple, others are more challenging.

This is a text in progress. The solution of some problems is only sketched and will be developed later. New sections are added on a regular basis.

# Table of contents

1. Introduction
2. Consulting customer data
3. Interactive data loading
4. Set and multiset operators
5. Displaying the content of two table in a single table
6. Dumping a database in XML format (version 1)
7. Dumping a database in XML format (version 2)
8. Data-driven multiple aggregations (table pivoting)
9. Paged reading of data
10. Raising temperature
11. Dynamic branching
12. Simulating arrays in SQLfast
13. Smart writing of data table
14. Checking functional dependencies in a table
15. Storing photos in a database
16. A photo album
17. Collecting web statistics
18. Computing missing data in incomplete statistical series
19. The Travelling Salesman problem
20. Numbering the rows of a table
21. Counting letters and words
22. Creating empty arrays
23. Computing the powerset of a set
24. The relational division
25. Checking set inclusion
26. Solving logic puzzles
27. Nest and unnest operators

## A1.1 Introduction

This appendix collects additional small applications illustrating the use of SQL and SQLfast to solve problems.

**Writing in progress**

## A1.2 Consulting customer data

Display the data of the customers who live in cities selected by the user.

```
openDB ORDERS.db;
  ask city;
  select * from CUSTOMER where City = '§city§';
closeDB;
```

**Script A1.1 -** Consulting the customers of a city - Simplest version

```
openDB ORDERS.db;
  ask city = [Enter the name of a city] City:;
  while ('$DIALOGbutton$' = 'OK');
    write-b Our customer(s) in $city$:;
    select CustID,Name from CUSTOMER where City = '§city§';
    ask city = [Enter the name of a city] City:;
  endwhile;
closeDB;
```

**Script A1.2 -** Consulting the customers of a city - Version 2

```
openDB ORDERS.db;
  while (True);
    ask city = [Enter the name of a city] City:;
        if ('$DIALOGbutton$' = 'Cancel') exit;
    write-b Our customer(s) in $city$:;
    select CustID,Name from CUSTOMER where CITY = '§city§';
  endwhile;
closeDB;
```

**Script A1.3 -** Consulting the customers of a city - Version 3

Note expression `'§city§'` instead of `'$city$'` to cope with city names including apostrophes (single quotes).

These scripts are available in file **Consulting-customer-data.sql**.

## A1.3 Interactive data loading

Let the user insert new rows in table CUSTOMER.

```
openDB ORDERS.db;
while (True);
  ask cid,nam,adr,cit,cat,acc = [Enter customer data]
      CustID|Name|Address|City|Cat|Account;
      if ('$DIALOGbutton$' = 'Cancel') exit;
  insert into CUSTOMER values
       ('$cid$','§nam§','§adr§','§cit§','$cat$',$acc$);
endwhile;
commitDB;
closeDB;
```

**Script A1.4 -** Filling the CUSTOMER table of the ORDERS database - Version 1

The version below controls SQL errors raised by the `insert` query:

```
openDB ORDERS.db;
set cid,nam,adr,cit,cat,acc = ;;;;;0;
while (True);
  ask-u cid,nam,adr,cit,cat,acc = [Enter customer data]
        CustID|Name|Address|City|Cat|Account;
        if ('$DIALOGbutton$' = 'Cancel') exit;
  insert into CUSTOMER values
        ('$cid$','§nam§','§adr§','§cit§','$cat$',$acc$);
        if ('$SQLdiag$' <> 'OK')
           showMessage Error $SQLdiag$: retry or exit;
endwhile;
commitDB;
closeDB;
```

**Script A1.5 -** Filling the CUSTOMER table of the ORDERS database - Version 2

These scripts are available in file **Interactive-data-loading.sql**.

## A1.4 Set and multiset operators

Considering two tables T1 and T2 as sets or multisets of elements (some elements are identical) how can we compute their union, difference and intersection? The answer provided by SQL is the use of union, except and intersect operators, as well as union all, except all and intersect all variants for multisets. However what if some of these operators are missing. So, the question is, how to simulate the set/multiset operators with standard queries.

### The SQL set operators

The queries of Script 1.6 illustrate the use of native union, except and intersect operator. We have also expressed the symmetric difference of the source tables. It must be noticed that these operators return pure sets, even when the arguments are multisets (that is, include duplicates).

```
create table T1(A integer);
create table T2(A integer);
insert into T1 values     (2),(3),(4),(5),   (7);
insert into T2 values (1),    (3),(4),(5),(6);
select A from T1 union select A from T2;
select A from T1 intersect select A from T2;
select A from T1 except select A from T2;
select A from (select A from T1 except select A from T2)
   union
select A from (select A from T2 except select A from T1);
```

**Script A1.6 -** Standard set operators

### Simulation of the set operators

The SQL set operator union, intersect and except can also be simulated with standard SQL constructs (Script 1.7).

In the simulation of the union operator, the full outer join operator creates couples with the identical values from both source tables, plus all the A values from each source table that do not match with the the other table. The coalesce function selects one of the *non null* values in each couple.

```
-- T1 union T2 (simulation) --
select coalesce(T1.A,T2.A) as A
from T1 full outer join T2 on (T1.A = T2.A);
-- T1 intersect T2 (simulation) --
select A from T1
where  A in (select A from T2);
-- T1 except T2 (simulation) --
select A from T1
where  A not in (select A from T2);
```

**Script A1.7 -** Simulation of standard set operators

## The SQL multiset operators

The queries of Script 1.8 create two multisets into tables T1 and T2, then computes their multiset *union*, *intersection* and *diffference*. We have also expressed the symmetric difference of the source tables. It must be noticed that these operators return pure sets, even when the arguments are multisets (that is, include duplicates).

```
insert into T1 values (1),(1),(2),(3),(3),(4),(4),(6),(7);
insert into T2 values (1),(2),(2),(3),(3),(5),(5),(6),(8);
select A from T1 union all select A from T2 order by A;
select A from T1 intersect all select A from T2 order by A;
select A from T1 except all select A from T2 order by A;
-- T1 symmetric difference T2 --
select A from (select A from T1 except all select A from T2)
   union all
select A from (select A from T2 except all select A from T1)
```

**Script A1.8 -** Native *union all*, *except all* and *intersect all*

## Simulation of the multiset operators

Let's suppose that our SQL engine does not provide the multiset operators. How can we simulate them with standard SQL queries?

For all the simulations, we first build a *synthetic view* of each argument, made up of the value and its multiplicity, as follows:

```
+-------+    +-------+
| V1    |    | V2    |
+---+---+    +---+---+
| A | N |    | A | N |
+---+---+    +---+---+
| 1 | 2 |    | 1 | 1 |
| 2 | 1 |    | 2 | 2 |
| 3 | 2 |    | 3 | 2 |
| 4 | 2 |    | 5 | 2 |
| 6 | 1 |    | 6 | 1 |
| 7 | 1 |    | 8 | 1 |
+---+---+    +---+---+
```

Then, we combine these views and we compute the resulting multiplicity:

- `union all`: we compute the *union* of these views and we add the N values for each A value (Script A1.9).
- `intersect all`: we compute the *left outer join* of the arguments and we subtract the second N value from the first one; if the second one is *null*, we choose the first one (Script A1.10).
- `except all` : same as `intersect all` but we compute the minimum of the N values or 0 if the second one is *null* (Script A1.11).

Finally, we *unfold* the resulting syntheric view through the UNFOLD CTE to produce a standard multiset.

```
create table T1(A integer);
create table T2(A integer);
insert into T1 values (1),(1),(2),(3),(3),(4),(4),(6),(7);
insert into T2 values (1),(2),(2),(3),(3),(5),(5),(6),(8);

with recursive
 TU(A,Multi) as
     (select A,sum(N)
      from (select A,count(*) as N,'T1' from T1 group by A
              union
            select A,count(*) as N,'T2' from T2 group by A)
      group by A),
 UNFOLD(A,N) as
     (select A,Multi from TU
        union
      select A,N-1 from UNFOLD where N > 1)
select group_concat(A) as "T1 union all T2"
from  (select A from UNFOLD order by A);
```

**Script A1.9 -** Simulating the *union all* operator

```
with recursive
 V1(A,N) as (select A,count(*) from T1 group by A),
 V2(A,N) as (select A,count(*) from T2 group by A),
 TE(A,Multi) as
   (select V1.A as "A", coalesce(V1.N - V2.N,V1.N) as SN
    from  (V1 left join V2 on V1.A = V2.A)
    where SN > 0),
 UNFOLD(A,N) as
   (select A,Multi from TE
      union
    select A,N-1 from UNFOLD where N > 1)
select group_concat(A) as "T1 except all T2"
from  (select A from UNFOLD order by A);
```

**Script A1.10 -** Simulating of the *except all* operator

```
with recursive
 V1(A,N) as (select A,count(*) from T1 group by A),
 V2(A,N) as (select A,count(*) from T2 group by A),
 TI(A,Multi) as
   (select V1.A, coalesce(min(V1.N,V2.N),0) as SN
    from   (V1 left join V2 on V1.A = V2.A)
    where  SN > 0),
 UNFOLD(A,N) as
   (select A,Multi from TI
      union all
    select A,N-1 from UNFOLD where  N > 1)
select group_concat(A) as "T1 intersect all T2"
from (select A from UNFOLD order by A);
```

**Script A1.11 -** Simulating of the *intersect all* operator

The results are shown below:

```
+-----------------------------------+
| T1 union all T2                   |
+-----------------------------------+
| 1,1,1,2,2,2,3,3,3,3,4,4,5,5,6,6,7,8 |
+-----------------------------------+


+------------------+
| T1 except all T2 |
+------------------+
| 1,4,4,7          |
+------------------+
```

```
+---------------------+
| T1 intersect all T2 |
+---------------------+
| 1,2,3,3,6           |
+---------------------+
```

These scripts are available in file **SQL-multiset-operators.sql**.

## A1.5 Displaying the content of two tables in a single table

It is easy to couple the data of tables CUSTORDER and DETAIL into a single table through a join on OrdID (Script 1.12).

```
select O.OrdID,CustID,DateOrd,ProdID,Qord
from   CUSTORDER O, DETAIL D
where  O.OrdID = D. OrdID
order by O.OrdID,ProdID;
```

**Script A1.12 -** A correct but awkward way to display order and detail data

However, the result is not particularly elegant since all the data of an order are repeated for each of its details:

```
+-------+--------+------------+--------+------+
| OrdID | CustID | DateOrd    | ProdID | Qord |
+-------+--------+------------+--------+------+
| 30178 | K111   | 2013-12-21 | CS464  | 25   |
| 30179 | C400   | 2013-12-22 | CS262  | 60   |
| 30179 | C400   | 2013-12-22 | PA60   | 20   |
| 30182 | S127   | 2013-12-23 | PA60   | 30   |
| 30184 | C400   | 2013-12-23 | CS464  | 120  |
| 30184 | C400   | 2013-12-23 | PA45   | 20   |
| 30185 | F011   | 2014-01-02 | CS464  | 260  |
| 30185 | F011   | 2014-01-02 | PA60   | 15   |
| 30185 | F011   | 2014-01-02 | PS222  | 600  |
| 30186 | C400   | 2014-01-02 | PA45   | 3    |
| 30188 | B512   | 2014-01-03 | CS464  | 180  |
| 30188 | B512   | 2014-01-03 | PA45   | 22   |
| 30188 | B512   | 2014-01-03 | PA60   | 70   |
| 30188 | B512   | 2014-01-03 | PH222  | 92   |
+-------+--------+------------+--------+------+
```

Let us try to display order data and detail data in a more natural layout. For example we would like to write the data of a detail just below the row of its order. Moreover, repeated data should be avoided.

Such a display requires two different formats, which suggests to extract the data through a **union** pattern (Script 1.13).

```
select OrdID,CustID,DateOrd,'' as ProdID,'' as Qord
from CUSTORDER
  union
select OrdID,'' as CustID,'' as DateOrd,ProdID,Qord
from DETAIL
order by OrdID,ProdID;
```

**Script A1.13 -** Order and detail data are displayed differently but ...

The result is much better:

```
+-------+--------+------------+--------+------+
| OrdID | CustID | DateOrd    | ProdID | Qord |
+-------+--------+------------+--------+------+
| 30178 | K111   | 2013-12-21 |        |      |
| 30178 |        |            | CS464  | 25   |
| 30179 | C400   | 2013-12-22 |        |      |
| 30179 |        |            | CS262  | 60   |
| 30179 |        |            | PA60   | 20   |
| 30182 | S127   | 2013-12-23 |        |      |
| 30182 |        |            | PA60   | 30   |
| 30184 | C400   | 2013-12-23 |        |      |
| 30184 |        |            | CS464  | 120  |
| 30184 |        |            | PA45   | 20   |
| 30185 | F011   | 2014-01-02 |        |      |
| 30185 |        |            | CS464  | 260  |
| 30185 |        |            | PA60   | 15   |
| 30185 |        |            | PS222  | 600  |
| 30186 | C400   | 2014-01-02 |        |      |
| 30186 |        |            | PA45   | 3    |
| 30188 | B512   | 2014-01-03 |        |      |
| 30188 |        |            | CS464  | 180  |
| 30188 |        |            | PA45   | 22   |
| 30188 |        |            | PA60   | 70   |
| 30188 |        |            | PH222  | 92   |
+-------+--------+------------+--------+------+
```

However, column OrdID should be valued for orders only. It must appear with its value in the CUSTORDER argument of the **`union`**, while it must be replaced by an empty string in the DETAIL argument. To make thinks more complicated, column OrdID serves as the major ordering key in the **`order by`** clause, so that each row (from CUSTORDER and DETAIL) must have the real value. Clearly, we need two different expressions of OrdID, one for sorting (column OrdSort) and the other one for display (column OrdDisp). In Script 1.14, the union operator builds an intermediate table with the two columns OrdSort and OrdDisp derived from source column OrdID. Only the second one is extracted to form the result set.

```
select OrdDisp as OrdID,CustID,DateOrd,ProdID,Qord
from (select OrdID as OrdSort,OrdID as OrdDisp,
             CustID,DateOrd,'' as ProdID,'' as Qord
      from CUSTORDER
        union
      select OrdID as OrdSort,'' as OrdDisp,
             '' as CustID,'' as DateOrd,ProdID,Qord
      from DETAIL)
order by OrdSort,ProdID;
```

**Script A1.14 -** Order and detail data are displayed differently. Final version.

Now the result is (near to) perfect!

| OrdID | CustID | DateOrd | ProdID | Qord |
|---|---|---|---|---|
| 30178 | K111 | 2013-12-21 | | |
| | | | CS464 | 25 |
| 30179 | C400 | 2013-12-22 | | |
| | | | CS262 | 60 |
| | | | PA60 | 20 |
| 30182 | S127 | 2013-12-23 | | |
| | | | PA60 | 30 |
| 30184 | C400 | 2013-12-23 | | |
| | | | CS464 | 120 |
| | | | PA45 | 20 |
| 30185 | F011 | 2014-01-02 | | |
| | | | CS464 | 260 |
| | | | PA60 | 15 |
| | | | PS222 | 600 |
| 30186 | C400 | 2014-01-02 | | |
| | | | PA45 | 3 |
| 30188 | B512 | 2014-01-03 | | |
| | | | CS464 | 180 |
| | | | PA45 | 22 |
| | | | PA60 | 70 |
| | | | PH222 | 92 |

These scripts are available in file **Displaying-the-content-of-2-tables.sql**.

## A1.6 Dumping a database in XML format (version 1)

**<to be revised>**

The objective of this application is to create an XML document that contains the data of the ORDERS.db database. The approach chosen is to generate a hierarchical document that describes all the customers, and within each of them, his/her orders with their details and the products ordered.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<database Name = "ORDERS.db">
  <CUSTOMER>
    <CustID> B112 </CustID>
    <Name> HANSENNE </Name>
    <Address> 23, r. Dumont </Address>
    <City> Poitiers </City>
    <Cat> C1 </Cat>
    <Account> 1250 </Account>
  </CUSTOMER>
  ...
  <CUSTOMER>
    <CustID> C400 </CustID>
    <Name> FERARD </Name>
    <Address> 65, r. du Tertre </Address>
    <City> Poitiers </City>
    <Cat> B2 </Cat>
    <Account> 350 </Account>
    <CUSTORDER>
      <OrdID> 30179 </OrdID>
      <DateOrd> 2013-12-22 </DateOrd>
      <DETAIL>
        <Qord> 60 </Qord>
        <PRODUCT>
          <ProdID> CS262 </ProdID>
          <Description> RAFT. PINE 200x6x2 </Description>
          <Price> 75 </Price>
          <QonHand> 45 </QonHand>
        </PRODUCT>
      </DETAIL>
      <DETAIL>
        <Qord> 20 </Qord>
        <PRODUCT>
          <ProdID> PA60 </ProdID>
          <Description> NAILS STEEL 60 (1K) </Description>
          <Price> 95 </Price>
          <QonHand> 134 </QonHand>
        </PRODUCT>
      </DETAIL>
    </CUSTORDER>
    ...
  </CUSTOMER>
...
</database>
```

*Warning*: the products that have not been ordered are not described. In addition, data on products may be strongly redundant since all the data of a product is copied as many times as this product is referenced by DETAIL rows.

Much improvement can be brought to this (skeleton) generator: generating the DTD of the document or its XML schema, specifying the identifiers (primary keys) and references (foreign keys) of the documents, producing an independent document for each entity (customer, order, product), etc.

```
outputOpen ORDERS.xml;
openDB ORDERS.db;
write <?xml version="1.0" encoding="iso-8859-1"?>;
write <database Name = "ORDERS.db">;
-- for each customer
for cid,nam,adr,cit,cat,acc = [select * from CUSTOMER];
  write @S2<CUSTOMER>;
    write @S4<CustID> $cid$ </CustID>;
    write @S4<Name> $nam$ </Name>;
    write @S4<Address> $adr$ </Address>;
    write @S4<City> $cit$ </City>;
    write @S4<Cat> $cat$ </Cat>;
    write @S4<Account> $acc$ </Account>;
  -- for each order of this customer
  for oid,dat = [select OrdID,DateOrd from CUSTORDER
                 where CustID = '$cid$'];
    write @S4<CUSTORDER>;
      write @S6<OrdID> $oid$ </OrdID>;
      write @S6<DateOrd> $dat$ </DateOrd>;
    -- for each detail of order
    for pid,qty = [select ProdID,Qord from DETAIL
                   where OrdID = '$oid$'];
      write @S6<DETAIL>;
        write @S8<Qord> $qty$ </Qord>;
        -- get product data of this detail
        extract pid,des,pri,qoh = select * from PRODUCT
                                  where ProdID = '$pid$';
        write @S8<PRODUCT>;
          write @S10<ProdID> $pid$ </ProdID>;
          write @S10<Description> $des$ </Description>;
          write @S10<Price> $pri$ </Price>;
          write @S10<QonHand> $qoh$ </QonHand>;
        write @S8</PRODUCT>;
      write @S6</DETAIL>;
    endfor;
    write @S4</CUSTORDER>;
  endfor;
  write @S2</CUSTOMER>;
endfor;
write </database>;
closeDB;
outputOpen console;
```

## A1.7 Dumping a database in XML format (version 2)

**<to be revised>**

In this version, we generate three subdocuments describing respectively the **customers** (THE_CUSTOMERS), the **orders** with their details (THE_CUSTOMER_ORDERS) and the **products** (THE_PRODUCTS). This structure includes no redundancies and is close to object-oriented data structures.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<database Name = "ORDERS.db">
<THE_CUSTOMERS>
  <CUSTOMER>
    <CustID> B112 </CustID>
    <Name> HANSENNE </Name>
    <Address> 23, r. Dumont </Address>
    <City> Poitiers </City>
    <Cat> C1 </Cat>
    <Account> 1250 </Account>
  </CUSTOMER>
  ...
  </CUSTOMER>
</THE_CUSTOMERS>

<THE_CUSTOMER_ORDERS>
  <CUSTORDER>
    <OrdID> 30185 </OrdID>
    <CustID> F011 </CustID>
    <DateOrd> 2014-01-02 </DateOrd>
    <DETAIL>
      <Qord> 260 </Qord>
      <ProdID> CS464 </ProdID>
    </DETAIL>
    <DETAIL>
      <Qord> 15 </Qord>
      <ProdID> PA60 </ProdID>
    </DETAIL>
    <DETAIL>
      <Qord> 600 </Qord>
      <ProdID> PS222 </ProdID>
    </DETAIL>
  </CUSTORDER>
  ...
</THE_CUSTOMER_ORDERS>

<THE_PRODUCTS>
  <PRODUCT>
    <ProdID> CS262 </ProdID>
    <Description> RAFT. PINE 200x6x2 </Description>
    <Price> 75 </Price>
    <QonHand> 45 </QonHand>
  </PRODUCT>
  ...
</THE_PRODUCTS>
</database>
```

Thanks to the use of the metadata of the SQLfast dictionary, it must be possible to write a *generator generator*, that is, a script that produces a generator for a definite

database. However, we must keep in mind that the structure of the document reflects a specific perception of the relations between the tables.

```
outputOpen ORDERS.xml;
openDB ORDERS.db;
write <?xml version="1.0" encoding="iso-8859-1"?>;
write <database Name = "ORDERS.db">;
write <THE_CUSTOMERS>;
for cid,nam,adr,cit,cat,acc = [select * from CUSTOMER];
  write @S2<CUSTOMER>;
    write @S4<CustID> $cid$ </CustID>;
    write @S4<Name> $nam$ </Name>;
    write @S4<Address> $adr$ </Address>;
    write @S4<City> $cit$ </City>;
    write @S4<Cat> $cat$ </Cat>;
    write @S4<Account> $acc$ </Account>;
  write @S2</CUSTOMER>;
endfor;
write </THE_CUSTOMERS>;
write <THE_CUSTOMER_ORDERS>;
for oid,cid,dat = [select OrdID,CustID,DateOrd
                   from CUSTORDER];
  write @S2<CUSTORDER>;
    write @S4<OrdID> $oid$ </OrdID>;
    write @S4<CustID> $cid$ </CustID>;
    write @S4<DateOrd> $dat$ </DateOrd>;
  for pid,qty = [select ProdID,Qord from DETAIL
                 where OrdID = '$oid$'];
    write @S4<DETAIL>;
      write @S6<Qord> $qty$ </Qord>;
      write @S6<ProdID> $pid$ </ProdID>;
    write @S4</DETAIL>;
  endfor;
  write @S2</CUSTORDER>;
endfor;
write </THE_CUSTOMER_ORDERS>;
write <THE_PRODUCTS>;
  for pid,des,pri,qoh = [select * from PRODUCT];
    write @S2<PRODUCT>;
      write @S4<ProdID> $pid$ </ProdID>;
      write @S4<Description> $des$ </Description>;
      write @S4<Price> $pri$ </Price>;
      write @S4<QonHand> $qoh$ </QonHand>;
    write @S2</PRODUCT>;
endfor;
write </THE_PRODUCTS>;
write </database>;
closeDB;
outputOpen console;
```

For example, we observe that DETAIL data are described as part of CUSTORDER data, while CUSTORDER data have not been considered to be part of CUSTOMER data. This means that we would have to consider that some foreign keys are to be interpreted as *part-of* composition while others just are plain relationships.

Going back to Chapter 20, we know that all the dictionary tables include a generic column named **`Ext`** (for **Ext**ension). This column could be used to indicate that a foreign key is the expression of a ***part-of*** relationship. However, since these tables are temporary, and therefore volatile, it is recommended to specify ***part-of*** foreign keys in an initialization SQLfast script to execute once the database has been opened.

## A1.8 Data-driven multiple aggregations (table pivoting)

Being able to use statistical functions *max*, *avg*, *count* and the like is one of the very basic competences of a novice SQL programmer. These functions can be used on a full subset of rows (in which case they return a single row result) or on a series of groups of rows through the `group by` clause.

Actually, they can be used in much more powerful queries that provide surprising results most of these novice programmers wouldn't think possible through a single (and simple) query. We will examine some of them.

Let us consider table EXPENSES in which we record ... expenses. For each of them, we indicate the amount and the day of the week it was spent. Clearly, this table has no unique key (Script 1.15).

```
create table EXPENSES (DAY integer, AMOUNT integer);
insert into EXPENSES values ('Monday',26);
insert into EXPENSES values ('Monday',56);
insert into EXPENSES values ('Monday',21);
insert into EXPENSES values ('Tuesday',15);
. . .
insert into EXPENSES values ('Sunday',32);
```

**Script A1.15 -** Recording expenses

We would like to extract from these raw data a table that provides, for each of the days of the week, the total amount spent. We could of course express a query that produces one line per day, but we add a special requirement: *each day must be represented by a column*, like in the following table:

```
+--------+---------+-----------+----------+--------+----------+--------+
| Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday |
+--------+---------+-----------+----------+--------+----------+--------+
|   103  |   196   |     84    |   261    |   251  |    0     |   32   |
+--------+---------+-----------+----------+--------+----------+--------+
```

What is unusual in this request, is that the number of columns depends on the data and not on the schema of the table, as is the case for almost all the queries we have developed until now: seven days, hence seven columns.

The query should act as if it would work on a source table that has been ***pivoted***, so that the column are data driven ('Monday', Tuesday', etc.) and the rows are the column of the source table (DAY, AMOUNT).

There are several ways to solve this kind of requests. The query of Script A1.16 produces the desired result in a particularly elegant and unexpensive way[1]. Each of

1. The source table is parsed only once contrary to other translations that rely on an expensive chain of self-joins.

the seven elements of the select list returns the sum of *something*, that is either the value of Amount if the line is of the appropriate day or zero for all the other days. For a given row, only one of these elements is augmented by a positive value, while all the other elements remain unchanged.

```
select sum(case when DAY = 'Monday'
                then AMOUNT else 0 end) as 'Monday',
       sum(case when DAY = 'Tuesday'
                then AMOUNT else 0 end) as 'Tuesday',
       . . .
       sum(case when DAY = 'Sunday'
                then AMOUNT else 0 end) as 'Sunday'
from EXPENSES;
```

**Script A1.16 -** Computing expenses by day of the week [Pivoting-1.sql ]

From this first example, we derive that of Script 1.17, which creates and populates a table in which we record, for each expense, the year and the quarter it was spent. Still no unique key since the same amount can be spent more than once the same quater.

```
create table EXPENSES (YEAR    integer not null,
                       QUARTER integer not null,
                       AMOUNT  integer not null);
insert into EXPENSES values (2011,4,220);
insert into EXPENSES values (2011,4,146);
insert into EXPENSES values (2011,4,128);
insert into EXPENSES values (2011,4,115);
insert into EXPENSES values (2012,1,126);
. . .
insert into EXPENSES values (2014,2,198);
insert into EXPENSES values (2014,2,235);
```

**Script A1.17 -** Recording expenses by year and by quarter [ Pivoting-2.sql ]

Now, we want to extract a more sophisticated table: for each year, the resulting data are totalled for each quarter. Each row is computed from a group of expenses of a given year as follows (observe that some years are incomplete, due to lack of data for some quarters):

| YEAR | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| 2011 | 0 | 0 | 0 | 609 |
| 2012 | 403 | 1016 | 1569 | 1261 |
| 2013 | 382 | 801 | 560 | 839 |
| 2014 | 379 | 1023 | 0 | 0 |

The corresponding query is given in Script 1.18.

```
 select YEAR,
       sum(case when QUARTER = 1
                then AMOUNT else 0 end) as 'Q1',
       sum(case when QUARTER = 2
                then AMOUNT else 0 end) as 'Q2',
       sum(case when QUARTER = 3
                then AMOUNT else 0 end) as 'Q3',
       sum(case when QUARTER = 4
                then AMOUNT else 0 end) as 'Q4'
from  EXPENSES
group by YEAR;
```

**Script A1.18 -** Computing expenses by year and by quarter [ Pivoting-2.sql ]

Adding a sixth column providing the total amount of the year is quite easy. The query is given in Script 1.19.

```
 select YEAR,
       sum(case when QUARTER = 1
                then AMOUNT else 0 end) as 'Q1',
       sum(case when QUARTER = 2
                then AMOUNT else 0 end) as 'Q2',
       sum(case when QUARTER = 3
                then AMOUNT else 0 end) as 'Q3',
       sum(case when QUARTER = 4
                then AMOUNT else 0 end) as 'Q4',
       sum(AMOUNT) as 'Total'
from  EXPENSES
group by YEAR;
```

**Script A1.19 -** Adding a column with the yearly expenses [ Pivoting-2.sql ]

```
+------+-----+------+------+------+-------+
| YEAR |  Q1 |  Q2  |  Q3  |  Q4  | Total |
+------+-----+------+------+------+-------+
| 2011 |   0 |    0 |    0 |  609 |   609 |
| 2012 | 403 | 1016 | 1569 | 1261 |  4249 |
| 2013 | 382 |  801 |  560 |  839 |  2582 |
| 2014 | 379 | 1023 |    0 |    0 |  1402 |
+------+-----+------+------+------+-------+
```

The total values can also be added as a new final line, through a **`union`** operator.

When counting only is needed, a simpler formula can be used[2]:

```
 select Year,
       count(QUARTER = 1 or null) as 'Q1',
       count(QUARTER = 2 or null) as 'Q2',
       count(QUARTER = 3 or null) as 'Q3',
       count(QUARTER = 4 or null) as 'Q4'
from  EXPENSES
group by YEAR;
```

**Script A1.20 -** Pivoting: simpler variant

These scripts are available in file **Pivoting.sql**.

2. Suggested in http://stackoverflow.com/questions/7674786/mysql-pivot-table/26968969 #26968969

## A1.9 Paged reading of data

There are several ways to examine the content of a small table (or of the result of a query). Writing it on the output channel certainly is the easiest technique. When the list of rows is fairly large, it is usual to display them in a *paged mode*. This technique consists in presenting the rows one page at a time. Each page comprises a fixed number N of rows. When a page has been displayed, the user asks for the next one by clicking on a button (for examply the **OK** button). The next N rows are then displayed, and so on until the end of the result set has been reached, or until the user decides to stop the presentation (by clicking on the **Cancel** button). Figure 1.1 shows a possible layout of the current data page.

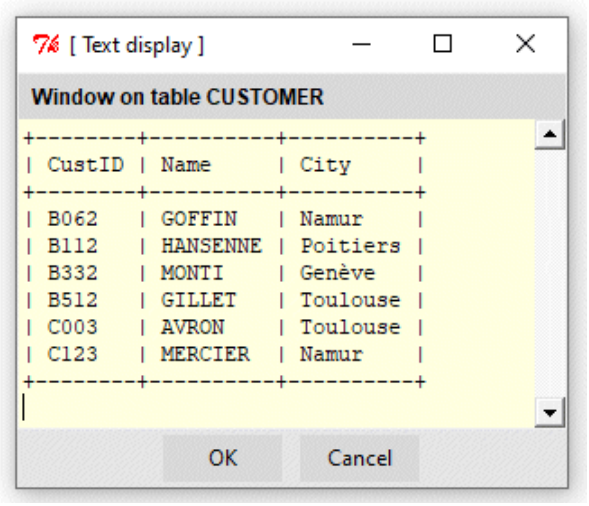



**Figure A1.1 -** Paged data browsing

This small application can be implemented by Script 1.21. It asks the user the presentation parameters: the rank of the first row, **`first`**, the rank of the last one, **`last`** (can be larger than the total number of rows) and the size of the page, **`step`**, (number of row). Each page is stored in variable **`result`** (**`outputOpen`** `result.var`). The indexed **`select`** query writes the next **`step`** rows in variable **`result`**, which are then displayed through the **`showText`** statement. If the user clicks on button **Cancel**, the display is stopped.

The script 1.22 provides several improvements on the latter:

– the user is invited to introduce the *data extraction query*,

– *default values* are provided for the page parameter and for the extraction query (Figure A1.2),

– the *size of the window* (**`ws`**) is automatically adapted to the number of rows of the page (**`step`**),

– the process is automatically stops when the last rows have been displayed (**if** ('$SQLdiag$' = 'NONE') **exit**).

```
ask start,end,step = [/b Parameters] Start|End|Step;
openDB ORDERS.db;
outputOpen result.var;
for I = [$start$,$end$,$step$];
   set result = ;
   compute J = $I$ + $step$;
   select CustID,Name,City from CUSTOMER
                            order by CustID #[$I$,$J$];
   showText result = [/b/x40/y10Window on table CUSTOMER];
   if ('$DIALOGbutton$' = 'Cancel') exit;
endfor;
outputappend window;
closeDB;
```

**Script A1.21 -** Presenting data one page at a time

```
set db = ORDERS.db;
set query = select CustID,Name,City\nfrom   CUSTOMER\norder by
CustID;
set start,end,step = 1,999,5;
set w = 40;
askCombo [askText-u query = [/b/x24/y5Data extraction query]]
        |[ask-u start,end,step = [/b/x12 Browsing parameters]
                                 Start||End||Step]
        |[ask-u w = Window width];
compute ws = $step$ + 6;
openDB $db$;
outputOpen result.var;
for I = [$start$,$end$,$step$];
   set result = ;
   compute J = $I$ + $step$;
   $query$ #[$I$,$J$];
   if ('$SQLdiag$' = 'NONE') exit;
   showText result = [/b/x$w$/y$ws$Window on table CUSTOMER];
   if ('$DIALOGbutton$' = 'Cancel') exit;
endfor;
outputAppend window;
closeDB;
```

**Script A1.22 -** Presenting data one page at a time - Improved script

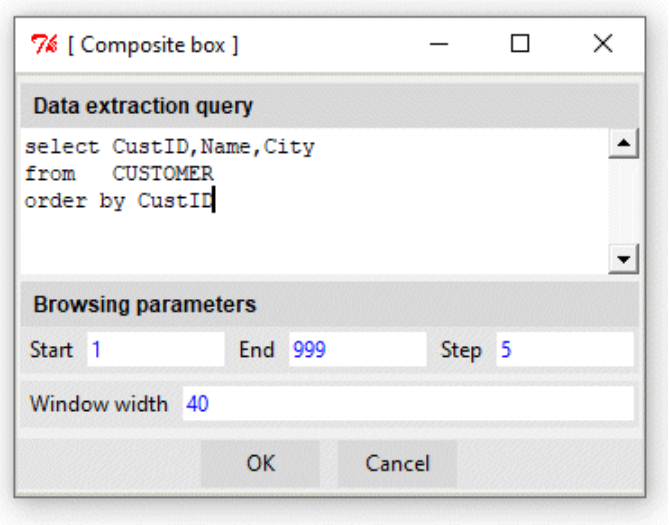



**Figure A1.2 -** Specifying the browsing parameters and the data extraction query

## Caveat

This way to extract the rows through an indexed query (`#[$I$,$J$]`) must be used with much caution. Indeed, it provides consistent results only if no concurrent process is allowed to update the data during the generation of the pages, in such a way that the numbering of the rows remain stable. If another user insert or delete rows, or modify the CustID column of some existing rows, the order of the rows may be disturbed. In addition, depending on the way the indexed accès is implemented, the extraction time can increase for large result sets[3]. In a real size application, a more efficient solution would be to read the rows sequentially and to build and display a page for each sequence of step rows.

The stability of the row positions can be achieved either through a shared locking of the table during its parsing, which may prevent concurent processes to update the data during a long time, or by storing a snapshot of the rows to display in a private temporary table.

The last script is available in file **Paged-reading-of-data.sql**.

3. The SQLfast engine implements indexed select queries through `offset` and `limit` clauses. Generally, the offset mechanism consists in retrieving the $I^{th}$ row by reading the I-1 preceding rows, which leads to an execution time of quadratic complexity.

## A1.10 Raising temperature

We are provided with a series of observations on the daily temperatures at different locations. These observations are stored in table OBS of database OBSERVATIONS.db. Table OBS is defined as follows:

```
create table OBS(Loc integer, Date date, Temp integer);
```

There is one observation per day at each location, so that the rows are unique on (Loc,Date). We consider the folllowing data:

```
insert into OBS values
(2,'2004-06-23',42),(2,'2006-12-28',39),(2,'2007-10-09',42),
(2,'2007-10-24',43),(2,'2009-06-17',45),(2,'2015-09-09',36),
(3,'2004-05-05',46),(3,'2005-05-24',43),(3,'2007-05-14',43),
(3,'2010-04-16',45),(3,'2010-11-22',43),(3,'2014-09-15',37),
(8,'2004-06-10',45),(10,'2005-06-10',46),(12,'2004-09-07',49),
(12,'2005-02-09',50),(12,'2006-09-04',46),(12,'2007-02-13',49),
(12,'2007-04-24',46),(12,'2007-05-15',47),(12,'2007-06-05',46),
(12,'2007-07-17',46),(12,'2007-10-23',49),(12,'2008-09-29',46),
(12,'2009-01-27',47),(12,'2009-04-23',45),(12,'2010-03-26',48),
(12,'2010-06-09',47),(12,'2011-06-23',43),(12,'2013-03-19',47),
(12,'2014-03-12',38),(12,'2014-12-19',40),(12,'2016-02-01',39),
(13,'2005-01-25',43),(13,'2005-09-15',45),(13,'2006-09-26',45),
(13,'2006-10-23',42),(13,'2008-11-03',44),(13,'2009-05-18',45),
(13,'2010-03-08',44),(13,'2010-10-12',45),(13,'2011-11-07',45),
(13,'2013-03-06',44),(13,'2013-11-20',36),(13,'2014-09-18',34),
(13,'2014-09-26',34),(13,'2015-06-16',36),(13,'2015-08-06',35),
(13,'2015-10-23',34),(13,'2016-08-08',35),(14,'2016-04-01',38),
(21,'2006-04-05',46),(21,'2008-08-07',46),(22,'2005-06-29',45),
(22,'2005-07-08',42),(22,'2005-07-15',42),(22,'2005-07-29',43),
(22,'2007-01-29',44),(22,'2007-04-25',44),(22,'2012-06-01',44),
(22,'2015-04-01',39),(24,'2010-05-28',49),(24,'2014-07-04',40),
(26,'2004-09-24',46),(26,'2004-10-15',46),(26,'2004-11-23',45),
(26,'2005-09-27',45),(26,'2006-12-04',46),(26,'2007-10-08',46),
(26,'2008-11-10',46),(26,'2009-11-13',46),(26,'2010-10-13',39),
(26,'2010-10-22',42),(26,'2010-11-03',43),(26,'2010-12-02',45),
(26,'2011-11-02',47),(26,'2012-10-22',46),(26,'2013-11-04',39),
(26,'2014-06-06',38);
```

The problem to solve is to *select the locations in which a temperature increase has been measured in at least three successive observations.*

To select them, we define all the consecutive triples of rows in each location, and we check whether their Temp values are increasing.

The critical part of the procedure is the construction of these triples: given an arbitrary row, how to select the next and the second next observations? The technique suggested here is to associate with each row a unique sequence number that follows the ascending order of values of (Loc,Date), then to execute a triple join based on the consecutive values of these sequence numbers.

```
create table OBS2(Seq integer not null primary key autoincrement,
                  Loc integer,Date date,Temp integer);
insert into OBS2(Loc,Date,Temp)
   select * from OBS order by Loc,Date;
select distinct O1.Loc,O1.Seq as "From",O3.Seq as "To"
from   OBS2 O1, OBS2 O2, OBS2 O3
where  O2.Seq = O1.Seq+1 and O3.Seq = O1.Seq+2
and    O1.Loc = O2.Loc   and O3.Loc = O1.Loc
and    O1.Temp < O2.Temp and O2.Temp < O3.Temp;
```

**Script A1.23 -**

Applied to the test data set, his query identifies seven triples of increasing temperature:

| Loc | From | To |
|---|---|---|
| 2 | 2 | 4 |
| 2 | 3 | 5 |
| 13 | 37 | 39 |
| 22 | 56 | 58 |
| 26 | 72 | 74 |
| 26 | 73 | 75 |
| 26 | 74 | 76 |

In a real size application, a more efficient solution would be to read the rows sequentially, ordered by (Loc,Date), and to identify the sequences of rows of increasing temperature.

The script is available in file **Temperature-rising.**sql.

## A1.11 Dynamic branching

Just a little programming trick: how to execute a specific section in a program based on the value of a variable. Probably not the most elegant way to proceed but nice illustration of dynamic branching. Reminiscence of the *computed goto* of Fortran!

```
label START;
   ask go = [Enter a number from 1 to 4, or 0 to exit] Number;
   if ('$go$' = '0' or '$DIALOGbutton$' = 'Cancel') stop;
   if ($go$ not between 1 and 5) goto START;
   goto display$go$;
label display1;
   pause -- 1 ---;
   goto START;
label display2;
   pause -- 2 ---;
   goto START;
label display3;
   pause -- 3 ---;
   goto START;
label display4;
   pause -- 4 ---;
   goto START;
```

```
set labels = display1,display2,display3,display4;
label START;
  ask go = [Select an operation] Operation:[!($labels$)];
  if ('$go$' = '0' or '$DIALOGbutton$' = 'Cancel') stop;
  goto $go$;
label display1;
   pause -- 1 ---;
   goto START;
label display2;
   pause -- 2 ---;
   goto START;
label display3;
   pause -- 3 ---;
   goto START;
label display4;
   pause -- 4 ---;
   goto START;
```

**Script A1.24 -** Playing with dynamic branching

## A1.12 Simulating arrays in SQLfast

In many programming languages, similar data items can be arranged in lists or in arrays, in which each item is denoted by its rang in the structure. If **CustList** is a list or array of customer numbers, then the 3th number generally is denoted by expression **CustList[3]**.

There is no such construct in SQLfast, and there are two reasons for this state of affair. First, SQLfast includes a much more powerful way to organize small or large collections of data items: *the SQL table*. In particular, creating a temporary table (a table that will disappear at database closing), loading any amount of data in it and manipulating its contents through the intuitive and powerful SQL language, make these pityful arrays and lists quite useless. Secondly, introducing arrays or lists would make the language more complex without real added value.

However, as a pure intellectual exercice and (also) as a nice application of loops, we will show how arrays can be simulated in SQLfast.

Script 1.25 is a small query system that displays the square of an input number. Instead of computing the squares in real time, it stores them into simulated array **SQUARE**. The script asks the user a number and returns her the square of this number.

The array is represented as a series of 50 variables named SQUARE[1], SQUARE[2], ..., SQUARE[50]. Actually, this number 50 is the value of a parameter defined by variable **size**. Element **I**, named **SQUARE[I]**, is created and filled by statement

```
compute SQUARE[$I$] = $I$ * $I$;
```

In the querying section, the script asks the user to type a number and stores it in variable **I** then checks the validity of the user input:[4]

```
if ('$I$' = '' or '$DIALOGbutton$' = 'Cancel') exit
if (not isNumeric('$I$')) next;
if ($I$ not between 1 and $size$) next
```

If the value if **I** is correct, it displays the value of the element at position **I** in **SQUARE**:

```
showMessage The square of $I$ is $SQUARE[$I$]$;
```

This statement deserves some explanation. Expression **$SQUARE[$I$]$** includes two embedded substitution operators. When it analyzes this expression, the SQLfast engine first identify **$I$**, a reference to variable **I**, and replaces **$I$** by the value of **I**, that we suppose to be **43**. So, after this substitution, the statement becomes

```
showMessage The square of 43 is $SQUARE[43]$;
```

4. Note that the first two comparisons consider that the value of **I** is a character string and not a number. This is prudent since we do not known the nature of the input data yet.

```
set size = 50;
-- creating and initializing the array
for I = [1,$size$] compute SQUARE[$I$] = $I$ * $I$;
-- Querying the array
while (True);
   ask I = [Number in [1,$size$] or Cancel to exit] Number;
      if ('$I$' = '' or '$DIALOGbutton$' = 'Cancel') exit;
      if (not isNumeric('$I$') or
          $I$ not between 1 and $size$) next;
   showMessage The square of $I$ is $SQUARE[$I$]$;
endwhile;
```

**Script 1.25 -** Simulating an array of squares as a series of variables

Further analysis identifies a second variable reference in this statement, namely **`$SQUARE[43]$`**, that denotes the value of variable **`SQUARE[43]`**, i.e., **1849**. The final state of the statement is then,

```
showMessage The square of 43 is 1849;
```

## Alternative technique: *SQLfast lists*

Arrays can also be simulated with SQLfast lists, that is, character strings built as the concatenation of character string elements separated by definite symbols. Function `group_concat` can be used to build lists and functions `item`, `itemLen`, `itemIndex` (and some others) can be used to consult lists.

- `group_concat2(v,d,o,d,sep)`: replacement for native `group_concat` aggregate function (functionally similar to the MySQL function). Values `v` are concatenated, separated by `sep`. Parameter `d` (0, 1) specifies the `distinct` modifier. Parameter `o` defines the order key and `d` specifies the order direction (1 = ascending, 2 = descending). Only available in SQL queries. Can be used to build lists from a database.
- `itemLen(s,sep)`: considering string `s` as an array of elements separated by string `sep`, returns the number of elements in `s`
- `item(s,i,sep)`: considering string `s` as an array of elements separated by string `sep`, returns the `i`th elements of `s`
- `itemIndex(s,e,sep)`: considering string `s` as an array of elements separated by string `sep`, returns the position of the first instance of `e` in `s`. Returns 0 if `e` is absent.
- `itemLenS(s)`, `itemS(s,i)`, `itemIndexS(s,e)`: same as functions above with the default separator as specified by parameter (e.g., `sep` = **`;`**).

The script 1.26 shows the application of SQLfast lists to implement the SQUARE query system.

```
set size = 50;
-- creating and initializing the array
set SQUARE = 1;
for I = [2,$size$]
    compute SQUARE = '$SQUARE$'||','||cast($I$*$I$ as char);
-- Querying the array
while (True);
   ask I = [Number in [1,$size$] or Cancel to exit] Number;
      if ('$I$' = '' or '$DIALOGbutton$' = 'Cancel') exit;
      if (not isNumeric('$I$') or
          $I$ not between 1 and $size$) next;
   compute sq = item('$SQUARE$',$I$,',');
   showMessage The square of $I$ is $sq$;
endwhile;
```

**Script A1.26 -** Simulating an array of squares as an SQLfast list

These scripts are available in file **Simulating-arrays.sql**.

## A1.13 Smart writing of a data table

The SQLfast **select** statement writes on the output channel the data extracted through a **select-from-where** SQL query in a tabular form (Script 1.27). This is particularly easy since we don't have to worry about the way output data are formatted. What if we want more specific format? We know that the format parameters of the **select** statement can be customized to a very large extent, ranging to pure text to LateX and RTF.

```
openDB ORDERS.db;
   select CustID,Name,City,Account from CUSTOMER
   where  Account <> 0;
closeDB;
```

**Script A1.27 -** The standard way to extract tabular data from CUSTOMER table

If we want more flexibility, then we can control the detail of the formatting process through a *dedicated script*, in which the data rows are *processed individually.* This is the goal of this exercise.

The more in-depth level of control is illustrated by the way column names and column values are aligned in the output format. For instance, we want all character values being left aligned, as in the standard format, but we would like the *column names being centered* and *numeric values being right aligned.* The result should look like the following:

```
+--------+-----------+-----------+---------+
| CustID |   Name    |   City    | Account |
+--------+-----------+-----------+---------+
| B112   | HANSENNE  | Poitiers  |    1250 |
| C123   | MERCIER   | Namur     |   -2300 |
| K111   | VANBIST   | Lille     |     720 |
| S127   | VANDERKA  | Namur     |   -4580 |
| B512   | GILLET    | Toulouse  |   -8700 |
| B062   | GOFFIN    | Namur     |   -3200 |
| C400   | FERARD    | Poitiers  |     350 |
| C003   | AVRON     | Toulouse  |   -1700 |
| D063   | MERCIER   | Toulouse  |   -2250 |
+--------+-----------+-----------+---------+
```

The script comprises three parts: computing the maximum length of each column, building the constant lines (bars and title line) then generating the data lines.

1. *Computing column width*. Script 1.28 is the fragment that computes maximum value lengths through an **extract** statement. The computing itself is carried out by an SQL statement (**select** `max(...),...`). The second step consists in adjusting these values to the length of column names (**compute** statements). These lengths are stored in variables **lcid**, **lnam**, **lcit** and **lacc**.

```
openDB ORDERS.db;
-- compute maximum column length
extract lcid,lnam,lcit,lacc
        = select max(length(CustID)),max(length(Name)),
                 max(length(City)),max(length(Account))
          from CUSTOMER;
-- ... including column name length!
compute lcid = max(length('CustID'),$lcid$);
compute lnam = max(length('Name'),$lnam$);
compute lcit = max(length('City'),$lcit$);
compute lacc = max(length('Account'),$lacc$);
```

**Script A1.28 -** Computing maximum column width

2. *Building constant lines*. This part is shown in Script 1.29. The three separation bars are identical. They are built through SQLfast function **substr**. The segments **d1** to **d4** separating names and values are generated as substrings of length lcid, lnam etc. of a dash string. The result is in variable **bar**. The title line (in variable **title**) is built from fragments obtained through the **Frame** function applied to column names. Each name is *centered* in its fragment (parameter **c**).

```
-- build bars
set dash = ------------------------------------------------;
compute d1 = substr('$dash$',1,$lcid$);
compute d2 = substr('$dash$',1,$lnam$);
compute d3 = substr('$dash$',1,$lcit$);
compute d4 = substr('$dash$',1,$lacc$);
set bar = +-$d1$-+-$d2$-+-$d3$-+-$d4$-+;
-- build title
compute t1 = Frame('CustID',$lcid$,'^',' ');
compute t2 = Frame('Name',$lnam$,'^',' ');
compute t3 = Frame('City',$lcit$,'^',' ');
compute t4 = Frame('Account',$lacc$,'^',' ');
set title = | $t1$ | $t2$ | $t3$ | $t4$ |;
```

**Script A1.29 -** Building bars and title line

3. *Generating data lines*. This part is shown in Script 1.30. The header, comprising 3 lines, is written from variables **bar** and **title**. A loop extracts the successive rows of table **CUSTOMER** and builds a data line for each of them. The values of the current row (stored in variables **cid** to **acc**) are framed in segments defined by the column lengths (variables **lcid** to **lacc**) and by their ty-

pe: *left* (**l**) for character values and right (**r**) for numbers. Finally they are stored in their source variables. The data line is built from these segments in variable **dataline** then written.

```
-- write header
write $bar$;
write $title$;
write $bar$
-- write data rows
for cid,nam,cit,acc = [select CustID,Name,City,Account
                       from   CUSTOMER where  Account <> 0];
   compute cid = Frame('$cid$',$lcid$,'<',' ');
   compute nam = Frame('$nam$',$lnam$,'<',' ');
   compute cit = Frame('$cit$',$lcit$,'<',' ');
   compute acc = Frame($acc$,$lacc$,'>',' ');
   set dataline = | $cid$ | $nam$ | $cit$ | $acc$ |;
   write $dataline$;
endfor;
-- write bottom bar
write $bar$;
closeDB;
```

**Script A1.30 -** Writing the data table

This script is available in file **Smart-writing-of-data-table.sql**.

## A1.14 Checking functional dependencies in a table

PURCHASE is a table in which we record purchases of products made by customers at a certain price. This table has three columns: CUST (customer name), PROD (product name) and PRICE (unit price of the purchase).

```
+--------+-------+-------+
| CUST   | PROD  | PRICE |
+--------+-------+-------+
| Andrew | sugar | 2.2   |
| John   | sugar | 2.2   |
| John   | salt  | 1.4   |
| Ann    | soap  | 1.4   |
| Ann    | salt  | 1.4   |
| John   | salt  | 1.2   |
| Andrew | sugar | 2.0   |
| John   | salt  | 1.6   |
| John   | soap  | 1.4   |
+--------+-------+-------+
```

We suspect functional dependency PROD $\longrightarrow$ PRICE to hold among the rows of this table. This property would mean that the value of PRICE depends on that of PROD only. In other words, for any two rows of the table, if they have the same value of PROD, they must have the same value of PRICE.

How could we confirm or discard this hypothesis? Let us find data that violate this supposed property. There are a lot of queries that can find such data. We examine four of them.

1. The first idea is to merely apply the definition. We seek pairs of rows with the same value of PROD and different values of PRICE (Script 1.31)

```
select distinct A1.PROD,A1.PRICE as PRICE
from   PURCHASE A1,PURCHASE A2
where  A1.PROD = A2.PROD
and    A1.PRICE <> A2.PRICE;
```

**Script A1.31 -** Checking violation of functional dependency (1)

The result is as follows, showing that *salt* has three different prices and *sugar* two:

```
+-------+-------+
| PROD  | PRICE |
+-------+-------+
| salt  | 1.2   |
| salt  | 1.4   |
| salt  | 1.6   |
| sugar | 2.0   |
| sugar | 2.2   |
+-------+-------+
```

2. The second approach considers the restriction[5] of PURCHASE to columns PROD and PRICE (`select distinct PROD,PRICE from PURCHASE`). In this 2-column table, the values of PROD should be unique, a property that is checked by identifying the groups of more than one row with the same values of PROD (Script A1.32).

```
select PROD,count(*) as Nb
from (select distinct PROD,PRICE from PURCHASE)
group by PROD
having count(*) > 1;
```

**Script A1.32 -** Checking violation of functional dependency (2)

The result provides the same diagnosis:

```
+-------+----+
| PROD  | Nb |
+-------+----+
| salt  | 3  |
| sugar | 2  |
+-------+----+
```

3. The third technique is similar to the previous one. It forms groups of rows with the same value of PROD then counts for each group the number of distinct values of PRICE (Script 1.33)

```
select PROD,count(distinct PRICE) as Nb
from   PURCHASE
group by PROD
having Nb > 1;
```

**Script A1.33 -** Checking violation of functional dependency (3)

4. The last query checks whether, for each row, there exists other rows with the same value of PROD but different values of PRICE (Scripts 1.34 and 1.35).

```
select distinct PROD
from   PURCHASE A
where exists(select 1 from PURCHASE
             where PROD = A.PROD
             and   PRICE < A.PRICE);
```

**Script A1.34 -** Checking violation of functional dependency (4)

5. The technical term for this restriction is *projection*.

```
select PROD,count(*) as Nb
from PURCHASE A
where exists(select 1 from PURCHASE
             where PROD = A.PROD
             and PRICE < A.PRICE)
group by PROD;
```

**Script A1.35 -** Checking violation of functional dependency (4bis)

**Conclusion**: PRICE does not depend functionally on PROD.

These scripts are available in file **Checking-FD-with-SQL.sql**.

## A1.15 Storing photos in a database

Most illustrations of the use of databases rely on simple data structures made up of elementary values, such as names, dates or numbers. A database may also be a convenient repository for pictures, photos, document, program binaries or videos, generally known as *large objects*. If they comprise characters only, they are called *Character Large Objects*, or *CLOB* for short. If they are made up of a (seemingly) arbitrary bit string, they are called *Binary Large Objects*, or *BLOB*. Through this small exercise, we have the opportunity to experiment with the loading and extraction of a collection of photos that we store in a database, then we extract them to produce a copy of this collection.

The database consists of one table with four columns: **ID** that identifies the photo, **FullName**, the full name of the file of the photo, **Name**, the local name of this file, **Size**, its size and **Picture**, the photo itself, defined of **BLOB** type (Script 1.36).

```
create table PHOTO(ID       integer not null primary key,
                   FullName varchar(128),
                   Name     char(64),
                   Size     integer,
                   Picture  BLOB);
```

**Script A1.36 -** Defining the photo database

### Selecting the photos to load

The next operation consists in letting the user select a directory that stores the photos we want to load then in writing in a text file (**dir.txt**) the list of the file names of these photos (Script 1.37). The command **selectDirectory** allows the user to select the source directory (returned in variable `dir`). A candidate directory is suggested (`Images`, a subdirectory of SQLfast). The command **fileNames** specifies the name of the source directory (`$dir$`) and the name of the target text file (`dir.txt`) , which is to be stored in the `File` directory.

```
selectDirectory dir = Images;
fileNames $fileDirectory$/dir.txt = $dir$/*.jpg;
```

**Script A1.37 -** Extraction of the list of photos to load

### Loading the photos in the database

The file **dir.txt** is read, one line at a time. For each file name, the size of the file is extracted through the function **infoFile** of library **LFile** (only the first result is interesting) and the content of the file (a photo) is loaded in binary variable **varLOB** through the command **BfileToVariable**.

This photo is then stored in the database in two steps. First, a row describing the photo is inserted in table PHOTO, then, the column **Picture** of this row is updated (**insertLOB**) to assign it the current photo (Script 1.38).

```
set id = 1;
for name = [file $fileDirectory$/dir.txt];
   function siz,x,x,x = LFile:infoFile {$name$};
   BfileToVariable varLOB = $name$;
   insert into PHOTO(ID,FullName,Name,Size)
          values ($id$,'$name$',lastPathNode('$name$'),$siz$);
   insertLOB varLOB = Picture from PHOTO where ID = $id$;
   compute id = $id$ + 1;
endfor;
commitDB;
```

**Script A1.38 -** Loading the photos in the database

Some data extracted from the database:

```
+----+------------------------+--------------+---------+
| ID | FullName               | Name         | Length  |
+----+------------------------+--------------+---------+
| 1  | D:\Photos\DSC02280.jpg | DSC02280.jpg | 1365745 |
| 2  | D:\Photos\DSC02282.jpg | DSC02282.jpg | 1417951 |
| 3  | D:\Photos\DSC02283.jpg | DSC02283.jpg | 1492799 |
| 4  | D:\Photos\DSC02284.jpg | DSC02284.jpg | 1448536 |
| 5  | D:\Photos\DSC02285.jpg | DSC02285.jpg | 1497381 |
 . . . .
| 250| D:\Photos\DSC02349.jpg | DSC02349.jpg | 1293530 |
+----+------------------------+--------------+---------+
```

## Extracting the photos from the database

The last operation extracts the photos from the database and stores each of them in the source file (Script 1.39), their name being prefixed with `'X_'`.

```
for name,photo = [select Name,Picture from PHOTO]
   VariableToFile photo = $dir$/X-$name$;
```

**Script A1.39 -** Extracting photos from the database

## Performance

Loading 250 photos (1.2 MB each) takes about 5 s. The extraction of these photos and their storage as individual files costs 8 s.

The SQL code developed above is available in script file **Photos-in-Database.sql**.

## A1.16 A photo album

Three small exercises of picture display.

### Introduction: a 4-line picture viewer

Surely, this picture viewer cannot compete with IrfanView, but it is **just 4 lines long**! The first step is the extraction of the names of the pictures. Contrary to the application in Section 1.15, we only store the *description* of the picture files in the database (Script 1.40). Its content is shown in Figure 1.3. The UDF function lastPathNode returns the last node of a file or directory path.

```
create table PICTURE(
       ID       integer not null primary key autoincrement,
       FullName varchar(128),
       Name     varchar(64));
fileNames Files/dir.txt = Images/Planes;
for fullName = [file Files/dir.txt]
    insert into PICTURE(FullName,Name)
    values ('$fullName$',lastPathNode('$fullName$'));
```

**Script A1.40 -** Storing the picture descriptions in the database

| ID | FullName | Name |
|---|---|---|
| 1 | Images/Planes/Plane-01.jpg | Plane-01.jpg |
| 2 | Images/Planes/Plane-02.jpg | Plane-02.jpg |
| 3 | Images/Planes/Plane-03.jpg | Plane-03.jpg |
| 4 | Images/Planes/Plane-04.jpg | Plane-04.jpg |
| 5 | Images/Planes/Plane-05.jpg | Plane-05.jpg |
| .. | ... | ... |
| 14 | Images/Planes/Plane-14.jpg | Plane-14.jpg |
| 15 | Images/Planes/Plane-15.jpg | Plane-15.jpg |

**Figure A1.3 -** Content of the PICTURE table

The slideshow is implemented as a simple loop that reads the table PICTURE and that displays, through a command **showPicture**, the content of a **jpg** file (or of a binary variable) (Script 1.41). Button **OK** displays the next picture while button **Cancel** stops the slideshow. The script 1.42 suggests a variant based on an intermediate binary variable. Figure 1.4 shows an example of the viewer.

```
for fullName,name = [select FullName,Name from PICTURE];
   showPicture $fullName$ = [/b Picture slideshow ($name$)];
   if ('$DIALOGbutton$' = 'Cancel') exit;
endfor;
```

**Script A1.41 -** A minimal picture viewer

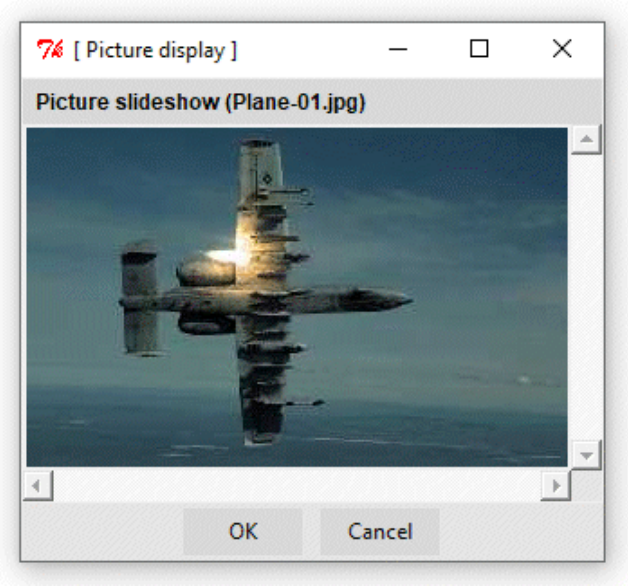



**Figure A1.4 -** The picture viewer

```
for fullName,name = [select FullName,Name from PICTURE];
   BfileToVariable photo = $FullName$;
   showPicture photo = [/b Picture slideshow ($name$)];
   if ('$DIALOGbutton$' = 'Cancel') exit;
endfor;
```

**Script A1.42 -** A variant of the picture viewer using an intermediate binary variable

## Selecting pictures - Preliminary version

The objective is to allow the user to select pictures among those displayed in a 2x2 grid, as suggested in Figure 1.5. In this preliminary version, the application displays only the first four pictures.

The command of Script 1.41 creates this selection box.

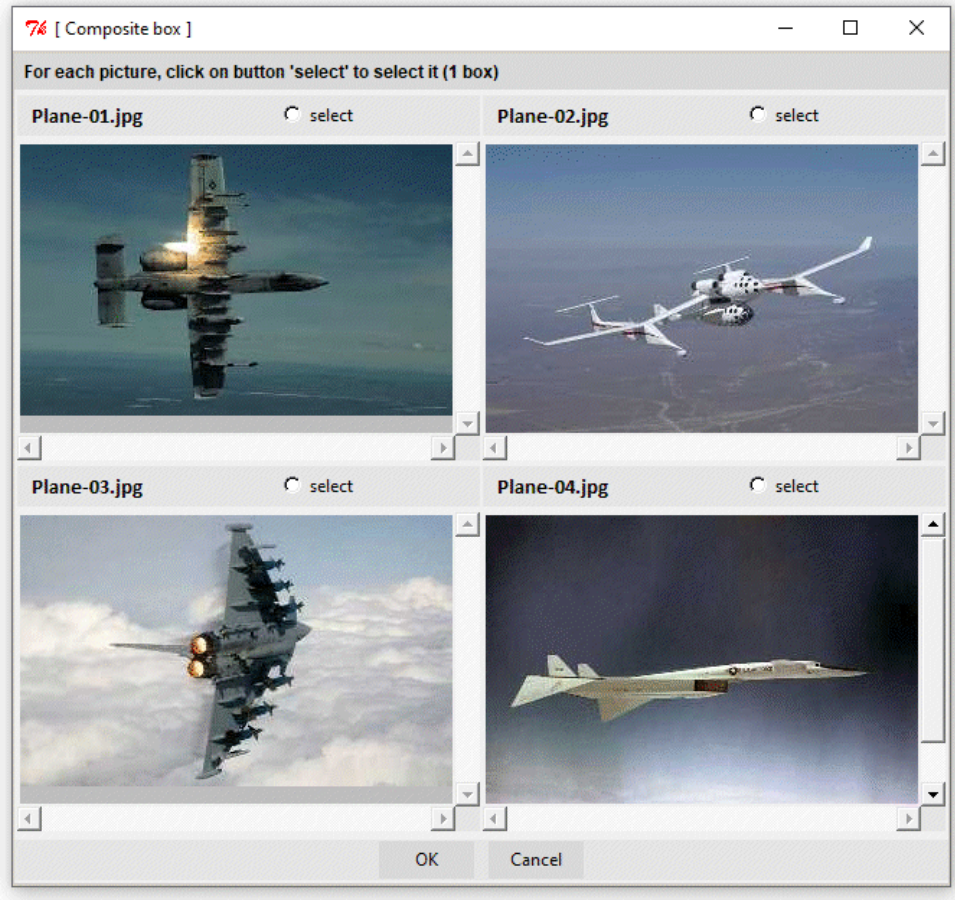



**Figure A1.5 -** Four pictures in a 2x2 grid

It is structured as a composite box made up of 8 elementary boxes. To understand how this command produces this layout, let us recall the role of the three box seperator:

| : the next box is placed under the last one

| | . the next box is placed in a column at the right of the last one

| | | : the next box is placed under the last one, but moved at the far left.

## Selecting pictures - Full version

Unfortunately, this prototype suffers from a severe drawback: the names and the paths of the pictures are hardwired in the code. Should we want to display other pictures, we would have to modify the code of the command.

The script 1.44 solves the problem. The first section is a loop that creates four variables (name1 to name4) for the names of the pictures and four variables (path1 to path4) for the paths of the picture file. In addition, these variables receive the appropriate values extracted from the database. In the `askCombo` command, the constants of the prototype are replaced by these variables.

```
askCombo [/bFor each picture, click on button 'select' ...]
        [selectOne p1 = {Plane-01.jpg}||select]
       |[showPicture Images/Planes/Plane-01.jpg = [/x300/y200]]
      ||[selectOne p2 = {Plane-02.jpg}||select]
       |[showPicture Images/Planes/Plane-02.jpg = [/x300/y200]]
     |||[selectOne p3 = {Plane-03.jpg}||select]
       |[showPicture Images/Planes/Plane-03.jpg = [/x300/y200]]
      ||[selectOne p4 = {Plane-04.jpg}||select]
       |[showPicture Images/Planes/Plane-04.jpg = [/x300/y200]]
```

**Script A1.43 -** The model of command for the 2x2 grid

The last command of the script lists the name of the pictures selected by the user. It deserves some explanation:

- for each value of `i`, the expression `p$i$` denotes the target variable of the `i`th `selectOne` command; same for the expression `nam$i$`, that will contain the name of the `i`th picture,
- actually, we need the *content* of these variables; so, `$p$i$$` is the state of the `i`th radio button and `$nam$i$$` is the name of the `i`th picture.

```
for id,name,path = [select ID,Name,FullName
                    from   PICTURE limit 4];
   set nam$id$  = $name$;
   set path$id$ = $path$;
endfor;

askCombo [/bFor each picture, click on button 'select' ...]
         [selectOne p1 = {$nam1$}||select]
        |[showPicture $path1$ = [/x300/y200]]
       ||[selectOne p2 = {$nam2$}||select]
        |[showPicture $path2$ = [/x300/y200]]
      |||[selectOne p3 = {$nam3$}||select]
        |[showPicture $path3$ = [/x300/y200]]
       ||[selectOne p4 = {$nam4$}||select]
        |[showPicture $path4$ = [/x300/y200]];

for i = [1,4] if ($p$i$$) write Picture $nam$i$$ selected;
```

**Script A1.44 -** A more general model of script for the 2x2 grid

The next exercise is a bit more challenging. Let's develop a script that presents in a series of 2x2 grids **all the images** in the database. Once the user has made his selection in the first box of four images (as in Figure 1.5), a new box opens with the next four images, and so on, until all the images have been submitted.

The structure of the `askCombo` command of Script 1.44 seems appropriate, provided it is fed with the successive sequences of 4 pictures.

When we examine this `askCombo` command, we observe that it is made up of four `selectOne-showPicture` blocks of commands of three different types, depending on the position of the block in the grid:

1. block of type 1 (first block)

```
[/bFor each picture, click on button 'select' ...]
 [selectOne p1 = {$nam1$}||select]
|[showPicture $path1$ = [/x300/y200]]
```

2. blocks of type 2 (second and fourth blocks)

```
||[selectOne p2 = {$nam2$}||select]
 |[showPicture $path2$ = [/x300/y200]]
```

3. block of type 3 (third block)

```
|||[selectOne p3 = {$nam3$}||select]
  |[showPicture $path3$ = [/x300/y200]]
```

A problem arises when the number of pictures is not a multiple of 4. In such a case, the last box comprises 1 to 3 blocks and has no longer a fixed size. Our sample picture directory contains 15 files, which leaves a last incomplete box of 3 pictures (Figure 1.6).

To address the greater complexity of the box structures, we suggest a quite different approach: instead of executing the `askCombo` commands in the script, we generate them in an external script file, then we execute the latter.

The external loop creates the boxes by generating the id of the first picture of each box (variable `first`), that is, 1, 5, 9, 13 in our example. For each value of `first`, the inner loop visits the four pictures of the current box. This loop provides the ID, Name and FullName of each picture *plus* its position in the box (variable `pos`, ranging from 1 to 4).

We deduce the block type of a picture from its position of the picture. The algorithm of the box generation is shown in Script 1.45.

The commands are written in file _Pictures.sql. To do so, the default output channel (the Output window) is changed to this file through command `outputOpen`.

Writing the command "`if ('$DIALOGbutton$' = 'Cancel') return;`" is a bit tricky. Indeed, the expression "`'$DIALOGbutton$'`" must be written as such, without the substitution operator `$$` being applied. So, we disable this substitution (command `substitution off`) before writing then enable it just after.

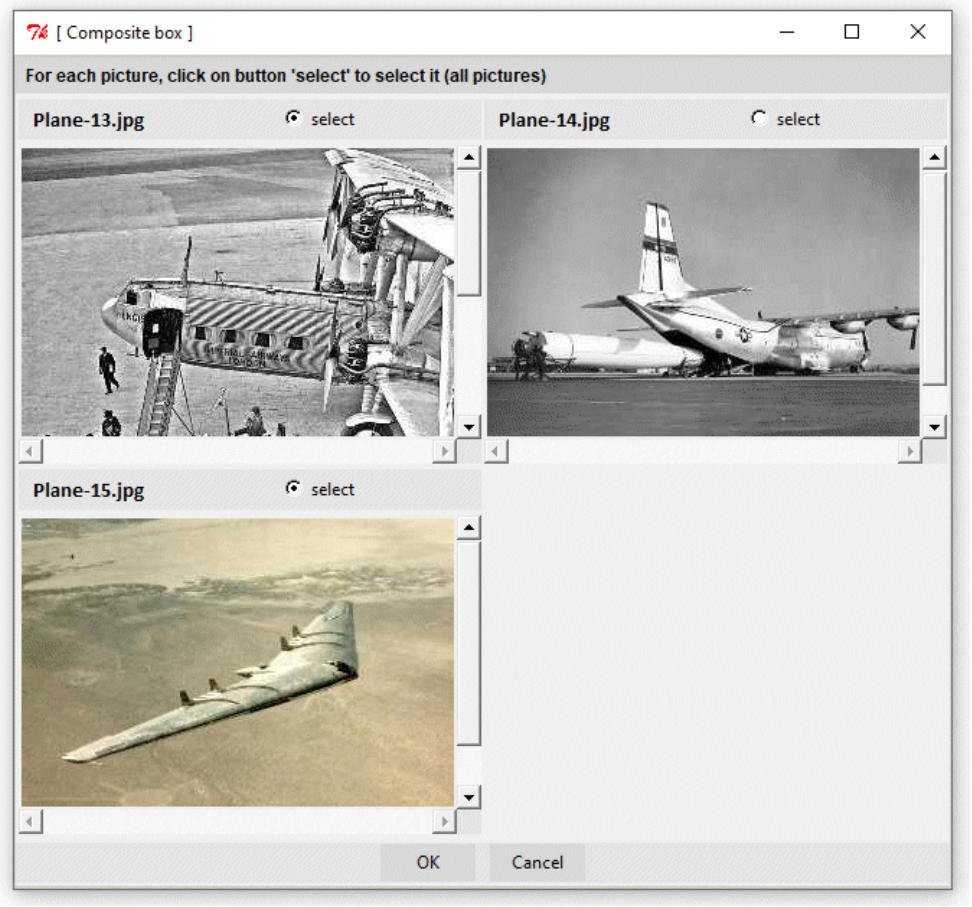



**Figure A1.6 -** The last box of a series of 15 pictures is incomplete

```
extract N = select count(*) from PICTURE;
outputOpen $scriptDirectory$/_Pictures.sql;
for first = [1,$N$,4];
   for id,nam,path,pos = [select ID,Name,FullName,
                          row_number() over ()
                          from PICTURE
                          where  ID >= $first$ limit 4];
      if ($pos$ = 1);
         -- first picture
         write askCombo [/bFor each picture, click ...];
         write @S9[selectOne p$id$ = {$nam$}||select];
         write @S8|[showPicture $path$ = [/x300/y200]];
      else;
         -- pictures 2, 3, 4
         if ($pos$ in (2,4);
            -- go to right
            write @S7||[selectOne p$id$ = {$nam$}||select];
            write @S8|[showPicture $path$ = [/x300/y200]];
         else;
            -- go to left
            write @S6|||[selectOne p$id$ = {$nam$}||select];
            write @S8|[showPicture $path$ = [/x300/y200]];
         endif;
      endif;
   endfor;
   write ;;
   substitution off;
   write if ('$DIALOGbutton$' = 'Cancel') return;;
   substitution on;
endfor;
outputAppend window;
execSQL $scriptDirectory$/_Pictures.sql;
```

**Script A1.45 -** Picture selection application - Full version

The SQL code of these three exercise is available in script file Photo-Albums.sql.

## A1.17 Collecting web statistics

Let us suppose that we have built a new web site to advertise our products. As usual, the home page includes a view counter that we eagerly check every evening to analyze the traffic. Instead of manually copying this counter in, say, a spreadsheet, we plan to automate this process.

The idea is to download the HTML code of the home page, to extract from it the value of the view count, then to store it in a database.

### Web page visual inspection

The first step of the project is to locate the counter in the HTML code. Once this code has been downloaded, we search it for a text fragment that appears near the count value in the browser window, e.g., "This page has been visited" that precedes this value. The following extract shows that the view count (**20723**) is unambiguously identified between two character strings, namely

```
'<li id="viewcount"> This page has been visited'
```

and

```
'times.</li>'.
```

```
<!DOCTYPE html ...>
<html xmlns="http://www.w3.org/1999/xhtml" ...>
  <head>
  ...
  </head>
  <body ...>
    ...
    <div id="footer">
      <ul id="f-list">
        <li id="lastmod"> ...</li>
        <li id="viewcount">This page has been visited 20723 times.</li>
        <li id="privacy">...</li>
        <li id="about">...</li>
        <li id="disclaimer">...</li>
      </ul>
    </div>
    ...
  </body>
</html>
```

### Extracting the view count

The address of the home page is symbolically represented by url `http://www.mywebsite/home.html`. Through SQLfast statement **`webpageToVariable`**, we download its HTML code and store it into text variable **page**:

```
webpageToVariable page = http://mywebsite/home.html;
```

To extract the view count from this text, we call the function **ExtractTagged-String** from the **LStr** library, which returns the first character string enclosed between two string arguments. By executing the following fragment, we store in variable **str** the value of the view count, then, by removing the space characters that may remain in this string, we obtain, in variable **count**, the current view count:

```
set tag1 = <li id="viewcount">This page has been visited;
set tag2 = times.</li>:
function str = LStr:ExtractTaggedString {$page$},$tag1$,$tag2$;
compute count = replace('$str$',' ','');
```

We note that the first argument, that can be quite large and that is likely to include commas, is protected by curly brackets { and }.

We can then store this value in a database, together with the current date (through system variable **date**). The database includes table STAT, comprising columns StatID (a technical id), DateStat , the date on which the count were extracted and Visits, the value of the view count (Script 1.46).

```
createOrReplaceDB WEB.db;
create table STAT(
      StatID   integer not null primary key autoincrement,
      DateStat date    not null,
      Visits   integer);
```

**Script A1.46 -** The database for statistics recording

Script 1.46 extracts the value of the view counter and stores it in the database.

```
openDB WEB.db;
  webpageToVariable page = http://www.mywebsite/home.html;
  set tag1 = <li id="viewcount">This page has been visited;
  set tag2 = times.</li>:
  function str = LStr:ExtractTaggedString {$page$},$tag1$,$tag2$;
  compute count = replace('$str$',' ','');
   insert into STAT(DateStat,Visits) values('$date$',$count$);
   commitDB;
closeDB;
```

**Script A1.47 -** Recording the view count of a web page - Final version

## A1.18 Computing missing data in incomplete statistical series

Computing statistics from a time series is fairly easy when the series is complete, that is, when it includes a valid value for each possible time point within its range. Unfortunately, in many series collected in the real world, some data are missing. Valid data have been collected for time points ..., t3, t4, t8, t9, ... , but those at t5, t6 and t7 are missing.

To make the discussion more concrete, let us consider the traffic statistics of a web page we have collected in section 1.17. Let us imagine that we plan to execute the extraction script manually every evening, in order to get a complete series from date t0 to yesterday. Unfortunately, from time to time, for all kinds of reasons, we forget, or we are unable, to capture the data of the day. Consequently, all we get is an *incomplete series* that could look like that depicted in Figure A1.7 (left). Should we show all time points, have they assigned a value or not, we would obtain the table at the right side.

| DateStat | Visits |
|---|---|
| ... | ... |
| 2016-02-26 | 63852 |
| 2016-02-27 | 63967 |
| 2016-03-02 | 64530 |
| 2016-03-03 | 64603 |
| 2016-03-04 | 64699 |
| 2016-03-06 | 64901 |
| ... | ... |

| DateStat | Visits |
|---|---|
| ... | ... |
| 2016-02-26 | 63852 |
| 2016-02-27 | 63967 |
| 2016-02-28 | ? |
| 2016-02-29 | ? |
| 2016-03-01 | ? |
| 2016-03-02 | 64530 |
| 2016-03-03 | 64603 |
| 2016-03-04 | 64699 |
| 2016-03-05 | ? |
| 2016-03-06 | 64901 |
| ... | ... |

**Figure A1.7 -** An incomplete time series and its explicit *unfolding*

Several algorithms have been designed to compute statistics from incomplete series. In this exercise we will follow another approach consisting in deriving a complete series by computing *plausible values* for missing data, so that we can apply standard function to compute statistics.

This raises two problems: *identifying missing dates* and *computing missing values*. To solve them, we first need to study how to deal with dates.

### Date arithmetic

Date manipulation is a fairly complex matter. Subtracting a date from another date or adding days to a definite date cannot be performed on the usual *human* representations shown in Figure 1.7, following ISO format YYYY-MM-DD. To this aim, dates (as well as time within a day) are converted into so-called *julian days*. This represen-

tation system assigns a real number to any date and time combination according to the following rules:

- Value **0.0** is assigned to an arbitrary reference date, typically *November 4th, 4712 BC, at noon* (this is the convention for SQLfast, inherited from the SQLite rules). Negative values represent previous dates and times, and positive ones later dates and times.
- The whole part of the value denotes a day; more precisely, it represents the second half of a day and the first half of the next day;
- The decimal part represents a fraction of a day, starting from noon: **.0** means noon of the current day, **.5** represents midnight of the current day and **.99999...** represents (nearly) noon of the next day.

Though these rules may seem needlessly complicated (why *at noon*?), the *julian day* representation of dates is fairly simple to use, at least for our needs. SQLfast offers two date conversion functions that hide the intricacy of human representation of dates:

- `julianday(D)` returns the *julian day* code of date `D` expressed as a character string in the ISO format . This code is a floating point value.
- `date(J)` returns the ISO date representation of *julian day* code `J` expressed as a real number.

For example, the expression

```
date(julianday('2016-02-28') + 2)
```

returns `2016-03-01`, since 2016 is a leap year. Similarly, the expression

```
int(julianday('2016-03-01') - julianday('2016-02-28'))
```

returns `2` (days).

Actually, SQLfast also provides some useful shorthands for these calculations:

- `jd(D)`: same as `juliandate(D)`, except that it returns 0.0 in case of error;
- `addToDate(D,N)`: returns the date `N` days later (or earlier if negative) than `D`; returns the empty string in case of error;
- `nextDate(D)`: returns the date that follows `D`; returns the empty string in case of error;
- `prevDate(D)`: returns the date that precedes `D`; returns the empty string in case of error.

## Identifying missing dates

Now, we have the necessary knowledge and tools to play with dates.

We consider the source table STAT in which the statistics collected are stored (Script 1.48).

```
create table STAT(
       StatID   integer not null primary key autoincrement,
       DateStat date    not null,
       Visits   integer);
insert into STAT(DateStat,Visits)
       values('2016-02-26',63852),'2016-02-27',63967),
             ('2016-03-02',64530),('2016-03-03',64603),
             ('2016-03-04',64699),('2016-03-06',64901), etc.
```

**Script A1.48 -** The source table and some sample data

Our concrete goal is to extract the *missing intervals* from table STAT. A missing interval is a maximum sequence of consecutive dates that are not present in STAT. It is identified as follows.

– We consider dates **D1** and **D2** from STAT such that
  • the row of **D2** immediately follows that of **D1** in STAT
  • **D2** is more than one day later than **D1**.
– All the dates between **D1** and **D2** (those excluded) form a *missing interval*.

The definitions of missing intervals are stored in table INTERVAL. For each of them, we specify dates D1 and D2 (Script 1.49) The result is shown in Figure 1.8.

```
create table INTERVAL as
   select D1.DateStat as D1, D2.DateStat as D2
   from   STAT D1, STAT.D2
   where  D2.DateID = D1.DateID + 1
   and    D2.DateStat > nextDate(D1.DateStat)
```

**Script A1.49 -** Computing the missing intervals

| D1 | D2 |
|---|---|
| ... | ... |
| 2016-02-27 | 2016-03-02 |
| 2016-03-04 | 2016-03-06 |
| ... | ... |

**Figure A1.8 -** Description of two missing intervals of STAT - First version

## Computing missing values

Now, we have to decide which values we assign to the missing dates. This process generally is called *interpolation*. "*Interpolation is a method of constructing new data points within the range of a discrete set of known data points. In engineering and science, one often has a number of data points, obtained by sampling or experimentation, which represent the values of a function for a limited number of values of the independent variable. It is often required to interpolate (i.e. estimate) the value of that function for an intermediate value of the independent variable. This may be achieved by curve fitting or regression analysis*." [en.wikipedia.org/wiki/Interpolation]

The function we have to deal with is Visits = V(DateStat), which appears to be very complicated mathematically, to say the least! The usual way to reconstruct missing values is through *polynomial interpolation*, that considers that the values of a missing interval should be close to those of a polynomial function, the coefficients of which are computed from the known values before and after the interval.

Its simplest form, *linear interpolation*, distributes the missing values *linearly* between the two surrounding values. If a missing interval is between D1 (with value V1) and D2 (with value V2), the value **V** of date D of this interval is computed as follows:

```
V = V1 + (V2 - V1)*(D - D1) / (D2 - D1)
```

Of course, date difference must be computed on their *julian day* codes and not on their human representations. We modify and extend the structure of table INTERVAL to better meet the requirements of this computation (Figure 1.9). Now the four parameters D1, D2, V1 and V2 are available for each interval.

| D1 | D2 | V1 | V2 |
|---|---|---|---|
| ... | ... | ... | ... |
| 2457445.5 | 2457449.5 | 63967 | 64465 |
| 2457451.5 | 2457453.5 | 64699 | 64901 |
| ... | ... | ... | ... |

**Figure A1.9 -** Missing intervals of STAT - Extended version

This new version is obtain through Script A1.50.

```
create table INTERVAL(
       D1 real, D2 real, V1 integer, V2 integer);
insert into INTERVAL
   select jd(S1.DateStat) as D1, jd(S2.DateStat) as D2,
          S1.Visits, S2.Visits
   from   STAT S1, STAT S2
   where  S2.StatID = S1.StatID + 1
   and    D2 > D1 + 1;
```

**Script A1.50 -** Computing the missing intervals - Extended version

The query of Script computes the intervals by comparing two successive rows, defined by window function `lead`.

```
insert into INTERVAL
   select D1,D2,V1,V2
   from (select jd(DateStat) as D1,
                lead(jd(DateStat),1) over (order by DateStat)
                       as D2,
                Visits as V1,
                lead(Visits,1)  over (order by DateStat)
                       as V2
         from   STAT
         order by DateStat)
   where  D2 > D1 + 1;
```

**Script A1.51 -** Computing the missing intervals through *window* functions

SQL provides several ways to compute the intermediate missing values, notably through a recursive query. However, the simplest and most intuitive algorithm is through a loop that computes the successive values of each interval.

For each interval, we first compute its *length* (the number of missing values):

```
len = D2 - D1 - 1
```

then the *step*, that is, the increment from a missing value to the next one:

```
step = (V2 - V1)/(D2 - D1)
```

Script A1.52 translates this algorithm. By combining these fragments, we get the complete solution of Script 1.53. The result of its execution on the sample data of Figure 1.7 is shown in Figure 1.10.

```
for D1,D2,V1,V2 = [select * from INTERVAL order by D1];
  compute len = $D2$ - $D1$ - 1;
  compute step = ($V2$ - $V1$)/($len$ + 1);
  for I = [1,$len$]
      insert into STAT(DateStat,Visits)
             values(date($D1$ + $I$), int($V1$ + $I$*$step$));
endfor;
```

**Script A1.52 -** Inserting the missing values for all the missing intervals

```
create table INTERVAL(
       D1 real, D2 real, V1 integer, V2 integer);
insert into INTERVAL
   select jd(D1.DateStat), jd(D2.DateStat), D1.Visits, D2.Visits
   from   STAT D1, STAT D2
   where  D2.StatID = D1.StatID + 1
   and    jd(D2.DateStat) > jd(D1.DateStat) + 1;
for D1,D2,V1,V2 = [select * from INTERVAL order by D1];
  compute len = $D2$ - $D1$ - 1;
  compute step = ($V2$ - $V1$)/($len$ + 1);
  for I = [1,$len$]
      insert into STAT(DateStat,Visits)
             values(date($D1$ + $I$), int($V1$ + $I$*$step$));
 endfor;
```

**Script A1.53 -** Inserting missing values - The complete script [Completing-Time-Series.sql]

| DateStat | Visits |
|---|---|
| ... | ... |
| 2016-02-26 | 63852 |
| 2016-02-27 | 63967 |
| **2016-02-28** | **64108** |
| **2016-02-29** | **64249** |
| **2016-03-01** | **64389** |
| 2016-03-02 | 64530 |
| 2016-03-03 | 64603 |
| 2016-03-04 | 64699 |
| **2016-03-05** | **64800** |
| 2016-03-06 | 64901 |
| ... | ... |

**Figure A1.10 -** Final state of table STAT completed

The SQL code developed above is available in script file Completing-Time-Series.sql.

## A1.19 The *Travelling Salesman* problem

The *Travelling Salesman* is a very popular optimization problem relying on graph processing. It can be described as follows: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?* [http://en.wikipedia.org/wiki/Travelling_salesman_problem]

Actually, this definition describes a special case of a more abstract problem: *Given a weighted graph (directed or undirected) and a starting node, find the least weighted loop(s) that visits all the other notes exactly once*.

We will develop the simplest of the exact solving techniques. It consists in generating all the circuits of the map starting from a definite city, then selecting the best ones. This brute-force method is easy to implement and is well fitted to small maps. We will develop it on the small map of Figure 1.11, comprising 4 cities and 12 roads. One of the characteristics of this case is that it is modelled by a *complete graph*, in which each city has a direct road to each of the other cities.

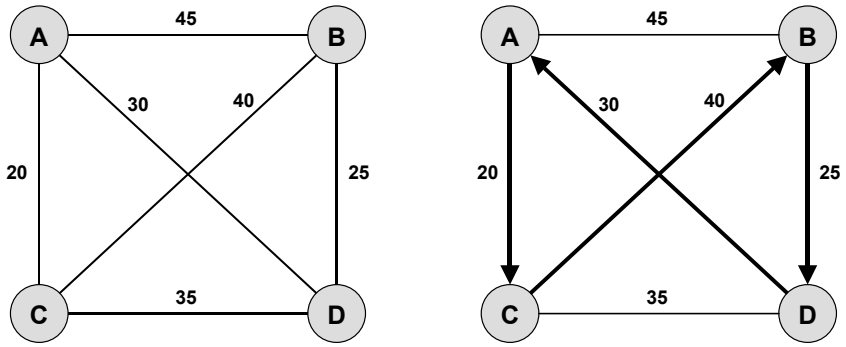



**Figure A1.11 -** The full 4-city problem and one of its optimal solutions

The SQL code of Script 1.54 creates the map in a temporary database (synthetic view in Figure 1.12).

The table PATHS created in Script 1.55 comprises all the paths starting from city A. Some comments about this code:

- Valid loops must consist of N roads, N being the number of cities connected by roads (we ignore the isolated cities).
- The content of PATHS is defined by a recursive CTE, named ... CTE!
- Its initialization query creates a 0-length loop comprising city A.

```
createDB InMemory;
create table CITY(Name char(10));
create table ROAD(Source char(10),Destin char(10),
                  Distance integer);
insert into CITY values ('A'),'B'),('C'),('D');
insert into ROAD values ('A','B',45),('A','C',20),
                        ('A','D',30),('B','C',40),
                        ('B','D',25),('C','D',35);
-- complete the graph:
insert into ROAD select Destin,Source,Distance from ROAD;
```

**Script A1.54 -** Create the DB for the 4-city problem

```
+--------+-----------+
| Source | Neighbors |
+--------+-----------+
| A      | B,C,D     |
| B      | C,D,A     |
| C      | D,A,B     |
| D      | A,B,C     |
+--------+-----------+
```

**Figure A1.12 -** Synthetic view of the 4-city map

– Its recursive query adds one *leg* to each of the paths last created, provided one of these conditions is met:
  - either the new city will close a valid loop (the path being extended comprises N-1 roads and the new city is A).
  - or the path being extended comprises less than N-1 roads and the new city has not yet been visited.

The global query inserts into PATHS all the paths created by CTE but the initial 0-length path.

The table PATHS contains 21 paths, shown in Figure 1.13. Among them 6 valid loops, with a total length ranging from 115 to 150. The solution is the loop A.C.B.D.A (and its inverse, since the graph is undirected), computed by Script 1.56.

```
extract N = select count(distinct Source) from ROAD;
create table PATHS as
 with recursive CTE(Len,Start,End,Path,Dist) as
 (select 0,'A','A','.A.',0
     union all
  select P.Len + 1, P.Start, R.Destin, P.Path||R.Destin||'.',
         P.Dist + R.Distance
  from   CTE P,ROAD R
  where  P.End = R.Source
  and    P.Len < $N$
  and    ((P.Len = $N$-1 and R.Destin = 'A')
           or
          (P.Len < $N$-1
           and P.Path not like '%.'||R.Destin||'.%'))
 )
 select Len,Start,End,Path,Dist
 from   CTE P
 where  P.Len > 0;
```

**Script A1.55 -** Computing all the paths starting from city A

| Len | Start | End | Path | Dist |
|---|---|---|---|---|
| 1 | A | B | .A.B. | 45 |
| 2 | A | C | .A.B.C. | 85 |
| 3 | A | D | .A.B.C.D. | 120 |
| 4 | A | A | .A.B.C.D.A. | 150 |
| 2 | A | D | .A.B.D. | 70 |
| 3 | A | C | .A.B.D.C. | 105 |
| 4 | A | A | .A.B.D.C.A. | 125 |
| 1 | A | C | .A.C. | 20 |
| 2 | A | B | .A.C.B. | 60 |
| 3 | A | D | .A.C.B.D. | 85 |
| 4 | A | A | **.A.C.B.D.A.** | 115 |
| 2 | A | D | .A.C.D. | 55 |
| 3 | A | B | .A.C.D.B. | 80 |
| 4 | A | A | .A.C.D.B.A. | 125 |
| 1 | A | D | .A.D. | 30 |
| 2 | A | B | .A.D.B. | 55 |
| 3 | A | C | .A.D.B.C. | 95 |
| 4 | A | A | **.A.D.B.C.A.** | 115 |
| 2 | A | C | .A.D.C. | 65 |
| 3 | A | B | .A.D.C.B. | 105 |
| 4 | A | A | .A.D.C.B.A. | 150 |

**Figure A1.13 -** The content of table PATHS

```
select Path as "Best paths",Dist as Distance
from   PATHS
where  Len = $N$
and    Dist = (select min(Dist) from PATHS where Len = $N$);
```

**Script A1.56 -** Selecting the best paths

```
+-------------+----------+
| Best paths  | Distance |
+-------------+----------+
| .A.C.B.D.A. | 115      |
| .A.D.B.C.A. | 115      |
+-------------+----------+
```

**Figure A1.14 -** The best path and its inverse

The map of Figure 1.15 may be more representative of small *travelling salesman* problems. It comprises 15 cities and 70 roads, each city being connected to 4.7 neighbors on the average.

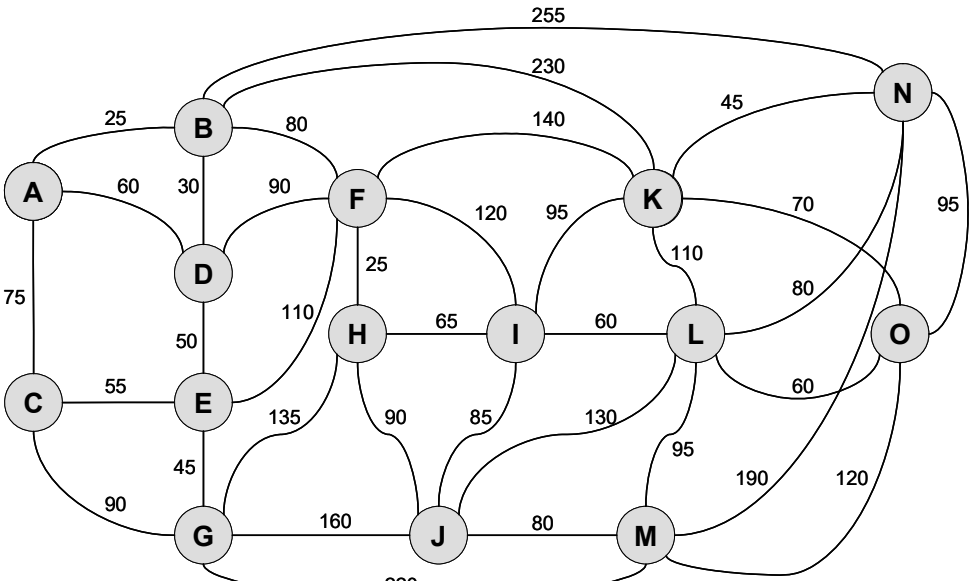



**Figure A1.15 -** A realistic 13-city problem

The algorithm generates 210,810 paths, including 2,226 full paths of length 16. The best solution is A.B.D.F.H.I.K.N.O.L.M.J.G.E.C.A (or its inverse) and costs 1,040 (Figure 1.16) The most expensive full path costs 1,890, that is, 82% more. The computing time is 0.9 s.

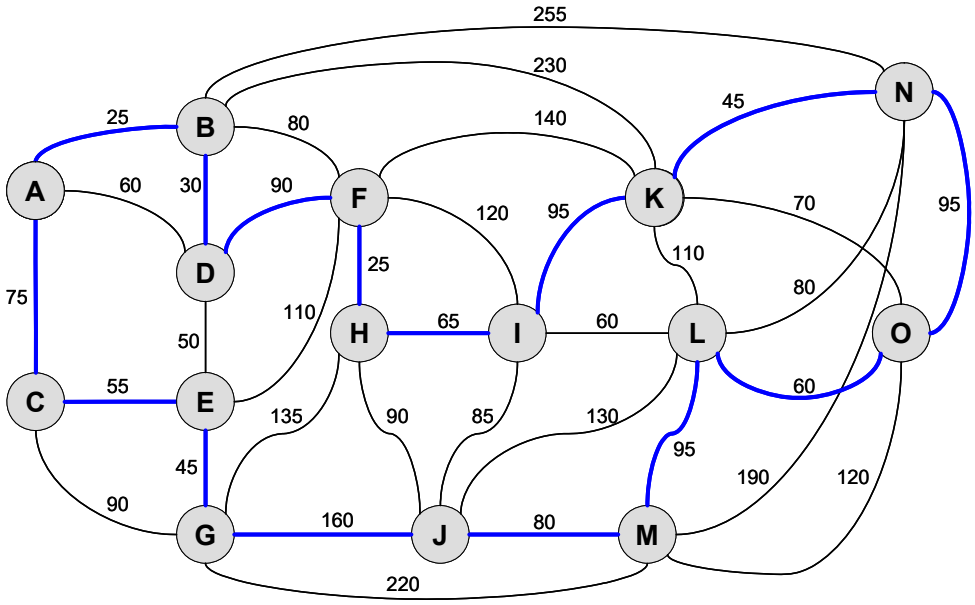



**Figure A1.16 -** The best solution(s) of the problem of Figure 1.15

The worst case of the algorithm is based on a complete graph, in which each node is connected to all the others, it is known to be O(N!). This means that if a problem with 10 cities is solved in 0.1 seconds, adding one city may cost 0.1x11 = 1.1 seconds.

For larger problems more sophisticated techniques have been designed. All of them have a complexity between $O(2^N)$ and $O(N^2x2^N)$. Some techniques have solved large problems (several thousands of cities), but their execution required several years of computation, fortunately on a large number of parallel processsors (16 CPU-years for 33,810 cities in 2005 and 136 CPU-years for 86.000 cities in 2006.

To compute solutions of even larger problems or to reduce drastically the computation time, *approximate algorithms* have been developed, based on heuristics or artificial intelligence. They generally provide *good enough* solutions.

For more information, http://en.wikipedia.org/wiki/Travelling_salesman_problem is a good starting point.

The SQL code developed above is available in script file Travelling-Salesman.sql.

## A1.20 Numbering the rows of a table

In some circumstances, it could be useful to associate to the rows of a table *unique sequence numbers*, like those that are shown in Figure 1.17, in which each row of table CUSTOMER (reduced to the first two columns) is assigned a sequence number.

These numbers define a specific order among the rows. In most cases, this order is also that of an expression computed for each row, in our case, the values of CustID. If this expression provides unique values, then the order is *strict total*. In this section, we examine row numbering techniques based on the values of CustID, that define a *strict total order*.

| Seq | CustID | Name |
|---|---|---|
| 1 | B062 | GOFFIN |
| 2 | B112 | HANSENNE |
| 3 | B332 | MONTI |
| 4 | B512 | GILLET |
| 5 | C003 | AVRON |
| 6 | C123 | MERCIER |
| 7 | C400 | FERARD |
| 8 | D063 | MERCIER |
| 9 | F010 | TOUSSAINT |
| 10 | F011 | PONCELET |
| 11 | F400 | JACOB |
| 12 | K111 | VANBIST |
| 13 | K729 | NEUMAN |
| 14 | L422 | FRANCK |
| 15 | S127 | VANDERKA |
| 16 | S712 | GUILLAUME |

**Figure A1.17 -** Numbering the rows of table CUSTOMER according to the order on CustID values

### A1.20.1 First technique: applying a mathematical definition

We assign to each row the number of CustID values that precede it in the desired ordering, plus 1. This is what Script 1.57 translates. The rows of the source table are stored in target table CUST_ORD1 augmented by these numbers.

```
create table CUST_ORD1 as
select (select count(*)+1 as Seq
        from CUSTOMER C
        where C.CustID < CUSTOMER.CustID) as Seq,
        CustID,Name
from CUSTOMER order by CustID;
```

**Script A1.57 -** Technique 1: applying the mathematical definition

### A1.20.2Second technique: adding the source rowids

The rowids of the rows of a table uniquely identify these rows in this table. Most often, the rowids are integer numbers sequentially assigned to the rows when they are inserted. These rowids are therefore good candidates for numbering rows.

In this technique, we store each row of the source table, augmented by its rowid. This algorithm is shown in Script 1.58.

However, though it creates a strict total order, this approach provides an order that is meaningless, just depending on the way the SQL engine have stored the rows in the source table. This technique provides an ordering on Seq that is random wrt to that on CustID, unless the row initially have been stored by increasing value of CustID (which is not the case). It could be fine when we content ourself with any arbitrary ordering.

```
create table CUST_ORD2(Seq integer,
                       CustID char(12),Name char(32));
insert into CUST_ORD2
select rowid,CustID,Name from CUSTOMER order by CustID;
```

**Script A1.58 -** Technique 2: adding the source rowids

### A1.20.3Third technique: adding the target rowids

We store the rows of table CUSTOMER ordered on CustID. Then, we update them by adding their rowids in the target table. If the SQL engine assigns rowids in the order the rows are inserted, then these rowids define the same order as the CustID values. This technique is coded in Script 1.59.

```
create table CUST_ORD3 as
  select 0 as Seq,CustID,Name from CUSTOMER order by CustID;
update CUST_ORD3 set Seq = rowid;
```

**Script A1.59 -** Technique 3: adding the target rowids

### A1.20.4Fourth technique: adding an *auto incremente* column

When the SQL engine provides (as most do) a variant of *auto increment* column type, we can use such a column to number the rows of the source table. This technique is illustrated in Script 1.60. The target table contains a primary key that comprises *auto increment* column Seq. When the rows of CUSTOMER are inserted in this table, in ascending order of column CustID, then column Seq automatically is assigned the desired row numbering.

```
create table CUST_ORD4(Seq integer not null primary key
                       autoincrement,
                       CustID char(12),Name char(32));
insert into CUST_ORD4(CustID,Name) select CustID,Name
                                   from   CUSTOMER
                                   order by CustID;
```

**Script A1.60 -** Technique 4: adding a auto increment column

### A1.20.5Fifth technique: recursive CTE

The CTE technique is based on a recursive definition of the desired numbering: the rows are numbered in ascending order of CustID values if

1. the first row is assigned sequence number 1,
2. the sequence number of the **n**th row of the sequence is that of the preceding row + 1.

This definition is translated into a recursive CTE in Script 1.61.

The initialization query (`select 1,CustID,Name ...`) assigns number **1** to the row with the lowest value of CustID.

The recursive query (`select S.Seq+1,C.CustID,C.Name ...`) selects the row in CUSTOMER that has the lowest value of CustID (`min(CustID)`) among all those that have a value of CustID higher than that of the last new row inserted (`S.CustID`).

```
create table CUST_ORD5 as
with SEQUENCE(Seq,CustID,Name) as
     (select 1,CustID,Name
      from (select CustID,Name
            from   CUSTOMER order by CustID limit 1)
         union
      select S.Seq+1,C1.CustID,C1.Name
      from   CUSTOMER C1, SEQUENCE S
      where  C1.CustID = (select min(C2.CustID)
                          from   CUSTOMER C2
                          where  C2.CustID > S.CustID)
      )
select * from SEQUENCE;
```

**Script A1.61 -** Technique 5: recursive CTE

### A1.20.6Sixth technique: Window function

The SQL `window` function can be used to dynamically associate a sequence number to the sorted result set of a select query. This is just what we want to do in this application (Script 1.62)!

```
create table CUST_ORD6 as
select row_number() over (order by CustID) as CustID, Name
from   CUSTOMER;
```

**Script A1.62 -** Technique 6: SQLfast list

### A1.20.7Seventh technique: SQLfast list

We still use a CTE to create the sequence numbers (SEQUENCE) and a comma-separated list of the CustID values (single row CTE LIST). Then, by joining these CTE, for each value of SEQUENCE, we extract the next item of LIST. The resulting table can be used to extract the other data from CUSTOMER.

```
create table CUST_ORD7 as
with SEQUENCE(Seq) as
     (select 1
         union all
      select Seq+1 from SEQUENCE
      where Seq < (select count(*) from CUSTOMER)),
     LIST(Id) as
     (select group_concat2(CustID,1,CustID,0,',')
      from   CUSTOMER)
select Seq,item(Id,Seq,',') as CustID
from SEQUENCE,LIST;
```

**Script A1.63 -** Technique 6: SQLfast list

### A1.20.8Comparing the numbering techniques

These scripts have been executed to number the 32,768 rows of a source table. The execution times of the first five techniques on a typical machine are shown in Figure 1.18.

| Numbering technique | Time (in ms.) |
|---|---|
| 1: applying a mathematical definition | 59,500 |
| 2: adding the source rowids | 10 |
| 3: adding the target rowids | 18 |
| 4: adding a auto increment column | 11 |
| 5: recursive CTE | 70,800 |

| | |
|---|---|
| 6: Window function | 30 |
| 7: SQLfast lists[a] | 57,800 |

a.Result with the fast version of SQLfast list operators

**Figure A1.18 -** Comparison of the five techniques

Techniques 1, 5 and 7 clearly must be avoided but for small tables. Though technique 2 provides the best figure, it should also be discarded since it does not define a meaningful ordering. Among the five other techniques, which produce the desired result, the fastest is based on a *auto increment* column. If such columns are not available, technique 3 will be used preferably. If the latter technique is not appropriate, then the window technique will do the job.

The seven scripts are available in file **Numbering-rows.sql**. They can be executed on source tables of various size.

### A1.20.9Pairing two tables

The operation consists in joining two tables, say T1 and T2, that have no *common columns*. The source tables, that have the same size (same number of rows), are joined on the rank of their rows: first row of T1 with the first row of T2, second row of T1 with the second row of T2, and so on.

This operation is quite common (and quite easy) in standard programming languages but a bit more tricky in set oriented languages such as SQL. The techniques we have developed in this section give us a straightforward solution: we add to T1 and T2 a numbering column, then we perform a simple join on these columns. The latter can then be discarded from the result.

The SQL code developed above is available in script file **Numbering-rows.sql**.

## A1.21 Counting letters and words

Though a generalist programming language, SQL includes powerful functions to analyze textual values. In this short application, we decompose the values of a character strings into individual characters. Then, we try a more ambitious goal, identifying and counting words in a text.

### First aplication: counting characters

Script 1.64 takes as input a column of a table (City of CUSTOMER) that comprises character strings. It generates a table showing, for each value of the column, each of its character and its position in column value.

The core of the process is the definition of view LETTERS. It is built from the cross join of two tables:

- SEQ that provides the sequence of the first 30 integer numbers starting from 1; these numbers represents the positions of characters in character strings; note that 30 is the declared length of column City.
- the set of distinct values of City from CUSTOMER.

Through the cross join, each value N of SEQ is associated with each value of City. The values of SEQ are limited by the length of the current value of City (otherwise, function `substr` would fail). For any couple of values (N,City), function `substr` extract the $N^{th}$ character of City. This character appears in column Letter of view LETTERS.

However, not all characters are letters. First, we want to remove accents and to convert upper case letters into lower case. So, (Genève ≡ geneve). This is the objective of UDF function `simplify`. Secondly, we want to discard characters that are not pure letters: spaces, symbols, digits. For this, we will use the powerful SQL function `translate(S,P,R)`. This function converts each character of string S according to equal-length strings P and R, that act as a dictionary. Each character of S that appears in P is replaced by the corresponding character in R. In view LETTERS, each character of each City value is translated as follows:

- P is formed by the characters to reject: `' .,''0123456789'`
- R replaces each of them by symbol `'$'` (we can chose any symbol absent from all city names): `'$$$$$$$$$$$$$$$'`. The other characters (the letters) pass through this filter without change.

So, when translated in this way, any character from City either remains intact or is converted into symbol `'$'`. So a letter is a character that has not been converted into `'$'`. Hence the last condition of the view definition. An excerpt of the result is shown in Figure 1.19

```
openDB ORDERS.db;
create temp view LETTERS(City,Rank,Letter) as
  with SEQ(N) as
     (select 1 union all select N+1 from SEQ where N < 30)
  select CUS.City,N,simplify(substr(CUS.City,N,1)) as Letter
  from   SEQ,(select distinct City
              from   CUSTOMER) as CUS
  where  N <= length(CUS.City)
  and    translate(Letter,
                   ' .,''0123456789',
                   '$$$$$$$$$$$$$$') <> '$';
select * from LETTERS order by City,Rank;
```

**Script A1.64 -** Decomposing character strings into individual characters

| City | Rank | Letter |
|---|---|---|
| Bruxelles | 1 | b |
| Bruxelles | 2 | r |
| Bruxelles | 3 | u |
| Bruxelles | 4 | x |
| Bruxelles | 5 | e |
| Bruxelles | 6 | l |
| Bruxelles | 7 | l |
| Bruxelles | 8 | e |
| Bruxelles | 9 | s |
| Genève | 1 | g |
| Genève | 2 | e |
| Genève | 3 | n |
| Genève | 4 | e |
| Genève | 5 | v |
| Genève | 6 | e |
| Lille | 1 | l |
| Lille | 2 | i |
| Lille | 3 | l |
| Lille | 4 | l |
| Lille | 5 | e |
| ... | . | . |

**Figure A1.19 -** Extracting the characters from column City

Just curious: what are the most frequent letters in City names? The answer is given by Script 1.65. It is generally agreed that the frequency of letters is a characteristics of each language. These seemingly innocent queries could perhaps be a good starting point for a tool that guesses the language of texts of unknown origin!

```
select Letter,count(*) as Frequency
from   LETTERS
group by Letter
order by Frequency desc,Letter;
```

**Script A1.65 -** Computing letter frequencies

| Symbol | Frequency |
|---|---|
| e | 8 |
| l | 6 |
| u | 4 |
| s | 4 |
| r | 4 |
| i | 4 |
| o | 3 |
| t | 2 |
| p | 2 |
| n | 2 |
| a | 2 |
| x | 1 |
| v | 1 |
| m | 1 |
| g | 1 |
| b | 1 |

**Figure A1.20 -** Frequency of the letters of column City of table CUSTOMER

## Second application: analyzing the words of a text

Now let's take on a slightly more demanding challenge: identifying and counting **words**. We consider two text units, paragraphs and words. The source of texts is a series of tweets (the paragraphs) stored in table TWEET. We just record the identifier and the body of each message (coded as a *character large object*, or CLOB).

```
createDB InMemory;
create table TWEET(TweetID char(24),Body CLOB);
insert into TWEET values
  ('T01','How will cities recover from covid-19'),
  ('T02','More cases in small cities! OK?'),
  ('T03','Still no drug?'),
  ('T04','OK but how to protect poor cities from Covid 19?'),
  ('T05','Why not reuse the drug that cured Covid 18?'),
  ('T06','We should investigate the role of Illuminati!');
```

**Script A1.66 -** The source text as a series of tweets

Once again, to make words more visible (and easier to extract), we eliminate non alphabetical characters, such as symbols and digits, through functions simplify and translate (Script 1.67). This produces the canonical version of the text (1.21).

```
select TweetID,
       replace(translate(simplify(Body),
                         '.,''-!?"0123456789',
                         '   $$$$$$$$$$$$$'),'$','') as Body
from TWEET;
```

**Script A1.67 -** Deriving the canonical form of the messages

| TweetID | Body |
|---|---|
| T01 | how will cities recover from covid |
| T02 | more cases in small cities ok |
| T03 | still no drug |
| T04 | ok but how to protect poor cities from covid |
| T05 | why not reuse the drug that cured covid |
| T06 | we should investigate the role of illuminati |

**Figure A1.21 -** Canonical version of the messages

Now, we extract the individual words from each message by joining a list of integers (CTE SEQ) with the list of canonical messages based on the following condition: a integer **N** matches message **C** if **N** is less than or equal to the length (in words) of **C**. For each match, the **N**$^{th}$ word of **C** is output. (Script 1.68). The result is shown in Figure 1.22.

```
create view WORDS(TweetID,Rank,Word) as
with recursive
   SEQ(N) as
      (select 1 union all select N+1 from SEQ where N < 1000),
   CAN(TweetID,Body) as
      (select TweetID,
              replace(translate(simplify(Body),
                                '.,''-!?"0123456789',
                                '   $$$$$$$$$$$$$'),'$','')
       from TWEET),
   WRD(TweetID,N,Word) as
       (select C.TweetID,N,trim(item(C.Body,N,' ')) as Word
        from SEQ,(select TweetID,Body from CAN) C
        where N <= itemLen(C.Body,' ') and Word <> '')
select * from WRD;
```

**Script A1.68 -** Building the list of words of a series of messages

| TweetID | Rank | Word |
|---|---|---|
| T01 | 1 | how |
| T02 | 1 | more |
| T03 | 1 | still |
| T04 | 1 | ok |
| T05 | 1 | why |
| T06 | 1 | we |
| T01 | 2 | will |
| T02 | 2 | cases |
| T03 | 2 | no |
| T04 | 2 | but |
| T05 | 2 | not |
| T06 | 2 | should |
| T01 | 3 | cities |
| T02 | 3 | in |
| T03 | 3 | drug |
| T04 | 3 | how |
| T05 | 3 | reuse |
| T06 | 3 | investigate |
| T01 | 4 | recover |
| T02 | 4 | small |
| T04 | 4 | to |
| T05 | 4 | the |
| T06 | 4 | the |
| T01 | 5 | from |
| T02 | 5 | cities |
| T04 | 5 | protect |
| T05 | 5 | drug |
| T06 | 5 | role |
| T01 | 6 | covid |
| T02 | 6 | ok |
| T04 | 6 | poor |
| T05 | 6 | that |
| T06 | 6 | of |
| T04 | 7 | cities |
| T05 | 7 | cured |
| T06 | 7 | illuminati |
| T04 | 8 | from |
| T05 | 8 | covid |
| T04 | 9 | covid |

**Figure A1.22 -** List of the words extracted from the Imessage together with their source (TweetID) and position (Rank) in the message

Table-view WORDS is a fine basis for many interesting analytical algorithms. We briefly describe three of them.

The script 1.69 rebuilds the source messages (at least it canonical form). It assumes that the order of the word in the concatenation is that of column Rank. Otherwise, we must use the SQLfast version group_concat2.

Two nice ways to process the list of words: rebuilding the source tweets and computing word frequencies

```
select TweetID,group_concat(Word,' ') as Tweet
from   WORDS
group by TweetID;
```

**Script A1.69 -** Rebuilding the source tweets

When examining the words of list 1.22, we observe that many words seems to be *unimportant*, bearing little meaning, such as *how*, *we*, *no*, *but*, *in* and the like. In text analysis, these words are qualify stop words. It is a good practice to remove them from the list of words before processing the latter.

In Script 1.70, we create the new table STOP that contain all the stop words we want to eliminate (list of common stop words for different languages are available on the web). Then, we derive table SWORDS from WORDS, that contains only the significant words. Finally, we compute the frequency of each word in the whole text (Figure 1.23).

```
create table STOP(Word varchar(16));
insert into STOP values ('a'),('are'),('as'),('but'),('from'),
            ('how'),('in'),('is'),('it'),('no'),('not'),('of'),
            ('ok'), ('that'),('the'),('to'),('we'),('will');

create view SWORDS as
   select TweetID,Rank,Word from WORDS
   where  Word not in (select Word from STOP);

select Word,count(*) as Frequency
from   SWORDS
group by Word
order by Frequency desc,Word;
```

**Script A1.70 -** Computing the significant words and their frequencies

| Word | Frequency |
|---|---|
| cities | 3 |
| covid | 3 |
| drug | 2 |
| cases | 1 |
| cured | 1 |
| illuminati | 1 |
| investigate | 1 |
| more | 1 |
| poor | 1 |
| protect | 1 |
| recover | 1 |
| reuse | 1 |

```
| role         | 1          |
| should       | 1          |
| small        | 1          |
| still        | 1          |
| why          | 1          |
+--------------+------------+
```

**Figure A1.23 -** Frequencies of the significant words

Script 1.71 associates with each word, the list of messages in which it appears (1.24).

```
select Word,group_concat(TweetID) as Source
from   SWORDS
group by Word
order by itemLen(Source,',') desc,Word;
```

**Script A1.71 -** Computing the source of the significant words

```
+--------------+--------------+
| Word         | Source       |
+--------------+--------------+
| cities       | T01,T02,T04  |
| covid        | T01,T05,T04  |
| drug         | T03,T05      |
| cases        | T02          |
| cured        | T05          |
| illuminati   | T06          |
| investigate  | T06          |
| more         | T02          |
| poor         | T04          |
| protect      | T04          |
| recover      | T01          |
| reuse        | T05          |
| role         | T06          |
| should       | T06          |
| small        | T02          |
| still        | T03          |
| why          | T05          |
+--------------+--------------+
```

**Figure A1.24 -** Where can I find significant words?

The third application computes the pairs of successive words that appear in the messages. We consider the self-join of SWORDS, that will produce all the pairs of words (W1,W2) in each message. A pair in a definite message is valid if W2 is the closest word to W1. Due to the elimination of stop words, the significant words are no longer numbered continuously, so that checking their *closeness* will prove a bit tricky. We apply the following logic: W2 can be associated to W1 if its rank is the lowest among the words that follow W1 in the message. Script 1.72 translates this principle and Figure 1.25 shows the result.

```
select W1.Word||' '||W2.Word as "Word pairs",
       W1.TweetID as "Found in"
from   SWORDS W1, SWORDS W2
where  W1.TweetID = W2.TweetID
and    W2.Rank = (select min(Rank) from SWORDS
                  where  TweetID = W1.TweetID
                  and    Rank > W1.Rank)
order by W1.TweetID;
```

**Script A1.72 -** Compute the pairs of significant words

```
+--------------------+----------+
| Word pairs         | Found in |
+--------------------+----------+
| cities recover     | T01      |
| recover covid      | T01      |
| more cases         | T02      |
| cases small        | T02      |
| small cities       | T02      |
| still drug         | T03      |
| protect poor       | T04      |
| poor cities        | T04      |
| cities covid       | T04      |
| why reuse          | T05      |
| reuse drug         | T05      |
| drug cured         | T05      |
| cured covid        | T05      |
| should investigate | T06      |
| investigate role   | T06      |
| role illuminati    | T06      |
+--------------------+----------+
```

**Figure A1.25 -** Pairs of significant words and their sources

## Final note

Though these algorithms are nice applications of SQL, they cannot realistically be used to analyze very large textual materials. In particular, extracting words from large texts will be better performed through standard or ad hoc parsing procedures written in standard programming languages.

The SQL code developed above is available in script file **Counting_words_in_texts.sql**.

## A1.22 Creating empty arrays

**Sparse arrays**, in which a subset only of the elements contain values, are easy to create and to fill. They are implemented as tables in which a subset of one or several columns represents the array indices + a column (or, more generally a set of columns) in which the values of the elements are stored. The index columns form the primary key of the table. An array of floating point values with two dimensions could be implemented by table **A**:

```
create table A(
       X integer not null,
       Y integer not null,
       V float   not null,
       primary key (X,Y));
```

Array elements are created by insertion:

```
insert into A values (8,12,29.5);
```

The elements that have not been assigned a value are not represented in the table.

### A1.22.1Full arrays

In some cases, we would like to create arrays in which all the elements are represented from the start, be they valued or not. The definition of the table is the same as above, except that column **V** is nullable (or `not null` with a default value):

```
create table A(
       X integer not null,
       Y integer not null,
       V float,
       primary key (X,Y));
```

Elements are assigned values through update queries:

```
update A set V = 29.5 where X = 8 and Y = 12;
```

The problem is to generate all the index values when we create the table implementing the array.

#### Technique 1: using a cross join to simulate decimal numeration

Script 1.73 suggests a first technique, applied to the creation of a unidimensional array, named (appropriately!) VECTOR, with 110 elements. It is based on table U that contains the 10 decimal digits, from 0 to 9. This table is then used to create index values from 1 to 110 according to the standard decimal numeration: the value of the number represented by '**abc**' is `100*a + 10*b + 1*c`. To create the index values,

table U is joined twice with itself, each of these three instances creating the 10 values of an order of magnitude. The generation is limited to 110 values of the index.

```
create temp table U(U integer);
  insert into U values(0),(1),(2),(3),(4),(5),(6),(7),(8),(9);
 create table VECTOR(X integer not null,V float);
 insert into VECTOR(X)
   select 100*U100.U + 10*U10.U + 1*U1.U + 1 as Ndx
   from   U U100, U U10, U U1 where Ndx <= 110;
```

**Script A1.73 -** Creation of an empty unidimensional array with 110 elements

| X | V |
|---|---|
| 1 | -- |
| 2 | -- |
| 3 | -- |
| 4 | -- |
| 5 | -- |
| 6 | -- |
| 7 | -- |
| 8 | -- |
| 9 | -- |
| 10 | -- |
| 11 | -- |
| ... | .. |
| 108 | -- |
| 109 | -- |
| 110 | -- |

**Figure A1.26 -** The unidimensional array created by Script A1.73

## Technique 2: using a recursive CTE

In Script 1.61, we have used a recursive CTE to add sequence numbers to the rows of a table. The problem we are tackling is fairly similar. Script 1.74 defines a CTE that creates the list of numbers from 1 to 110. This list is then stored in table VECTOR.

```
with recursive
  INIT(I)
  as (select 1 union select I+1 from INIT where I < 110)
insert into VECTOR select I from INIT;
```

**Script A1.74 -** Creation of an empty unidimensional array through a recursive CTE

This technique can be generalized to more than one dimension. Script A1.75 creates the table of an 8X6 array through the recursive CTE technique. CTE INIT1 creates a list of row numbers, from 1 to 8. From each number of this list, CTE INIT2 creates the couples of coordinates of the colomns of this row. The result is shown in Figure 1.27.

```
create table MATRIX(
       X integer not null,
       Y integer not null,
       V float,
       primary key (X,Y));
with recursive
  INIT1(I)
  as (select 1 union select I+1 from INIT1 where I < 8),
  INIT2(I,J)
  as (select I,1 from INIT1
        union
      select I,J+1 from INIT2 where J < 6)
insert into MATRIX(X,Y) select I,J from INIT2;
```

**Script A1.75 -** Creation of an empty bidimensional array through recursive CTE

| X | Y | V |
|---|---|---|
| 1 | 1 | -- |
| 1 | 2 | -- |
| 1 | 3 | -- |
| 1 | 4 | -- |
| 1 | 5 | -- |
| 1 | 6 | -- |
| 2 | 1 | -- |
| 2 | 2 | -- |
| . | . | .. |
| 7 | 6 | -- |
| 8 | 1 | -- |
| 8 | 2 | -- |
| 8 | 3 | -- |
| 8 | 4 | -- |
| 8 | 5 | -- |
| 8 | 6 | -- |

**Figure A1.27 -** A bidimensional array

Generalizing this technique to three or more dimensions is straightforward.

The SQL code developed above is available in script file **Creating-empty-arrays.sql**.

## A1.23 Computing the powerset of a set

The powerset of a set **S** is the set of all the subsets of **S**. This set is noted **P**(S) or $2^S$. For instance, the powerset of **S** = {A,B,C} is:

**P**(**S**) = {{},{A},{B},{C},{A,B},{A,C},{C,B},{A,B,C}}

Quite often, we are not interested in the empty set, so that, if considered useless, it will be discarded from the powerset.

To proceed, we first build a small example in which **S** is a set of names stored in the table NAMES:

```
create table NAMES(Name varchar(32) primary key);
insert into NAMES values ('Peter'),('Paul'),('Mary'),('Ann');
```

**Script A1.76 -** Example database

```
+-------+
| Name  |
+-------+
| Peter |
| Paul  |
| Mary  |
| Ann   |
+-------+
```

**Figure A1.28 -** The set of names

Our goal is to build the list of all the teams we can define with these persons.

### A1.23.1First technique

We will first generate all the *permutations* of each non empty subset of NAMES. Considering an arbitrary set, each sequence of its elements is a permutation. On S = {A,B,C}, we can derive six permutations: (ABC), (ACB), (BAC), (BCA), (CAB), (CBA).

Generating the permutations of a set is quite easy. To derive sets of elements from them, we must reduce the duplicates, that is, the permutations that comprise the same elements. This reduction itself is easy, provided the elements of the permutations are sorted.

In query 1.77, view POWERSET1 creates the powerset of the contents of table NAMES. It is defined as recursive CTE POSET that creates all the permutations of all the non empty subsets of NAMES. The `where` clause of the recursive query prevents names to appear more than once in a permutation. The permutations are formed as SQLfast lists of coma-separated names.

The permutations with the same contents are reduced by the main query as follows:

– each permutation is sorted through UDF function **itemSort**, so that all the permutations with the same contents are made identical
– the `distinct` modifier eliminates the duplicates.

```
create temp view POWERSET1(NameList) as
with recursive
  POSET(Member) as
  (select Name from NAMES
   union all
   select P.Member||','||N.Name
   from   POSET P, NAMES N
   where  P.Member not like '%'||N.Name||'%'
  )
select distinct itemSort(Member,0,0,',') from POSET;
```

**Script A1.77 -** Generating the permutations of each subset of NAMES then eliminating duplicates

Displaying the content of view POWERSET1 provides the expected powerset (figure 1.29).

```
+---------------------+
| NameList            |
+---------------------+
| Ann                 |
| Paul                |
| Mary                |
| Peter               |
| Ann,Paul            |
| Ann,Mary            |
| Ann,Peter           |
| Paul,Mary           |
| Paul,Peter          |
| Mary,Peter          |
| Ann,Paul,Mary       |
| Ann,Paul,Peter      |
| Ann,Mary,Peter      |
| Paul,Mary,Peter     |
| Ann,Paul,Mary,Peter |
+---------------------+
```

**Figure A1.29 -** The powerset of NAMES

### A1.23.2Second technique

With this technique, we skip the permutation step by enforcing the alphanumeric order in subset lists. A Name value is added to the current list if it is higher than the last one appended. In this way, the Name values are unique within each list and the Member lists also are unique within the view.

```
create temp view POWERSET2(NameList) as
with recursive
  POSET(Member) as
  (select Name from NAMES
     union all
   select P.Member||','||N.Name
   from   POSET P, NAMES N
   where  N.Name > item(P.Member,-1,',')
  )
select Member from POSET;
```

**Script A1.78 -** Generating the powerset of NAMES (technique 2)

**Notes**

`item(L,x,sep)` is a UDF function that returns the item in position **x** within list **L**. The list was built with separator **sep**. If **x** is negative, the position is specified backward, starting from the last element of **L**.

`itemSort(L,D,U,sep)` is a UDF function that returns list **L** the elements of which are sorted in ascending (**D** = 0) or descending (**D** = 1) alphanumeric order. If **U** = 0, all the source elements are preserved while if **U** = 1 duplicates are eliminated. The values of **L** are separated by symbol **sep**.

### A1.23.3Performances

Not surprisingly, the first technique, that generates a large number of permutations, proves slower than the second one. On a standard machine, the first technique generates the powerset of sets of size 7 in one second, while, in the same time, the second technique can process sets of size 12. Generating the powerset of a set of 10 elements costs 500 s. with technique 1 and 0.5 s. with technique 2.[6]

### A1.23.4Storing the powerset members in a table

From the list-based representation of the powerset of NAMES, we would like to transform these lists into a normalized table, in which each subset receives an Id (setID) and in which each member of each subset is represented by a row, as shown in figure A1.30.

```
+-------+--------+
| SetID | Member |
+-------+--------+
| 1     | Ann    |
| 2     | John   |
```

6. Technique 1 behaves like **N!** while technique 2 behaves like $k^N$, where N is the size of the source set and k some small integer constant.

```
| 3      | Mary   |
| 4      | Peter  |
| 5      | Ann    |
| 5      | John   |
| 6      | Ann    |
| 6      | Mary   |
| 7      | Ann    |
| 7      | Peter  |
| ...    | ...    |
| 14     | John   |
| 14     | Mary   |
| 14     | Peter  |
| 15     | Ann    |
| 15     | John   |
| 15     | Mary   |
| 15     | Peter  |
+--------+--------+
```

**Figure A1.30 -** Normalized representation of a powerset

In Script 1.79, the subset Id is created by counter setID while the inner loop decomposes lists to insert subset members into table POWERSET.

```
create temp table POWERSET(SetID integer, Member varchar(32));
set setId = 1;
for subset = [select NameList from POWERSET2];
   compute subsetLen = itemLen('$subset$',',');
   for idx = [1,$subsetLen$];
      compute name = item('$subset$',$idx$,',');
      insert into POWERSET values ($setId$,'$name$');
   endfor;
   compute setId = $setId$ + 1;
endfor;
```

**Script A1.79 -** Normalizing powerset representation

Rebuilding lists from this normalized table is straightforward (Script A1.80).

```
select SetID, group_concat(Member,',') as Member from POWERSET
group by SetId;
```

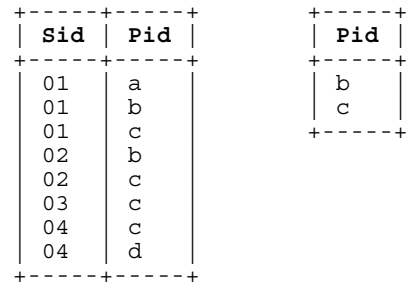
**Script A1.80 -** Rebuilding list-based representations of powerset members

These scripts are available in files **Compute-Powerset-of-a-set.sql** and **Compute-Powerset-of-a-set-(times).sql** .

## A1.24 The relational division

The relational division introduces in the relational algebra (one of the theoretical basis of SQL) an operator that implements the universal quantifier **for all**. It works as follows.

Let us consider the table S(Sid,Pid), recording data about which suppliers (Sid) can deliver which products (Pid) and the table P(Pid) recording the id's of a set of reference products (Pid). Typical contents of S and P are shown in Figure A1.31.

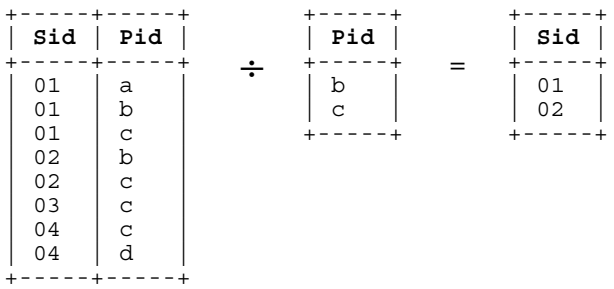


| Sid | Pid |
|---|---|
| 01 | a |
| 01 | b |
| 01 | c |
| 02 | b |
| 02 | c |
| 03 | c |
| 04 | c |
| 04 | d |

| Pid |
|---|
| b |
| c |

**Figure A1.31 -** Current contents of tables S (left) and P (right)

The division of S by P, noted S÷P, is the set of Sid values, each of them being associated in S with **all** the Pid values of P. This division returns *the suppliers who can supply all the reference products*. Since only suppliers 01 and 02 can deliver the products b and c, S÷P = {(01),(02)} (Figure 1.32).



| Sid | Pid |
|---|---|
| 01 | a |
| 01 | b |
| 01 | c |
| 02 | b |
| 02 | c |
| 03 | c |
| 04 | c |
| 04 | d |

÷

| Pid |
|---|
| b |
| c |

=

| Sid |
|---|
| 01 |
| 02 |

**Figure A1.32 -** Result of the division operator

If **s** is an **Sid** value from S such that there exists in P a value **p** such as (**s**,**p**) is not in S, then **s** does not belong to S÷P.

The relational division solves numerous problems the description of which includes expressions like *all* or *for each* or *for all*. Some examples based on our reference database ORDERS.db:

– which orders reference **all** the products?

– which customers have ordered **all** the products in 2017?

– in which cities did customers order **all** the models of nail?

– which products have been ordered in **all** cities?

The bad news is that SQL does not include the *division* operator, nor any equivalent predicate that could implement it. Mathematically speaking, SQL allows the expression of the *existence quantifier* (∃ = predicate `exists`) but not the *universal quantifier* (∀). The good news is that each of these quantifiers can be expressed with the other one. Considering predicate `p(x)`, we have:

$$\forall x \in X, p(x) \equiv \neg(\exists x \in X, \neg p(x))$$

This expression is a bit more complicated than desired but it can be translated in SQL in a (rather) straightforward way.

Let us go back to our S, P example. The natural description of the result of division S÷P can be written:

*the set of Sid values **s** in S such that:*

*for all Pid value **p** in P, (**s**,**p**) is a row of S*

Converted according to the SQL limitation, it comes:

*the set of Sid values **s** in S such that:*

*it is not true that (there exists a Pid value **p** in P such that (**s**,**p**) is not in S)*

Now, we are able to express the division of S by P in SQL (Script 1.81).

```
select distinct S1.Sid
from   S S1
where  not exists(select *
                  from   P
                  where  not exist(select *
                                   from   S
                                   where  S.Sid = S1.Sid
                                   and    S.Pid = P.Pid
                                   )
                  );
```

**Script A1.81 -** The SQL expression of S÷P

## Alternative technique

The problem can be seen as a *set inclusion* pattern. Let us consider this other view of S and P, that is easily produced through the group_concat aggregate function:

```
+-------------+     +------+
| SS          |     | PP   |
+-----+-------+     +------+
| Sid | Pids  |     | Refs |
+-----+-------+     +------+
| 01  | a,b,c |     | b,c  |
| 02  | b,c   |     +------+
| 03  | c     |
| 04  | c,d   |
+-----+-------+
```

**Figure A1.33 -** Synthetic view of S and P

A supplier is selected if its Pids value *includes* the Refs value of the single row of PP (Script 1.82).

```
select Sid
from  (select Sid,group_concat(Pid) as Pids
       from   S group by Sid) SS,
      (select group_concat(Pid) as Refs
       from   P) PP
where itemInclude(Pids,Refs,',');
```

**Script A1.82 -** Computing S÷P, an alternative technique

These scripts are available in file **Relational-division.sql**.

## A1.25 Checking set inclusion

Evaluating the relation between two sets (*are they identical*, *is one set a part of the other*) is a frequent problem in data analysis. SQL does not offer straightforward functions or predicates to evaluate such relations. They must be computed by (somewhat) convoluted queries that exploit the properties of the set inclusion relation. This section is devoted to the development of such computations.

### A1.25.1 The inclusion problem

Let us consider table T(A,B), populated as shown in Script 1.83. It contains abstract data but, for the sake of illustration, it could be interpreted as *supplier A can provide product B*.

```
createDB InMemory;
create table T(A char(10) not null, B char(10) not null);
insert into T values ('01','a'),('01','b'),('01','c'),
                     ('02','b'),('02','c'),
                     ('03','c'),
                     ('04','c'),('04','d');
```

**Script A1.83 -** Content of table T

One can read the data of table T as: *each A value has an associated* ***set of B values***, which we will call its ***B-set***. This interpretation is made more concrete by the data of Figure A1.34, which shows the content of table ABSET computed by Script 1.84.

```
create table ABSET as
       select A, group_concat(distinct B) as Bset
       from   T group by A;
```

**Script A1.84 -** Computing the B-set of each A value

```
+----+-------+
| A  | Bset  |
+----+-------+
| 01 | a,b,c |
| 02 | b,c   |
| 03 | c     |
| 04 | c,d   |
+----+-------+
```

**Figure A1.34 -** Table ABSET: a set of B values is associated to each A value

Now, let us consider the problem of finding all the couples (a1,a2) of A values such that the B-set associated to a2 is a subset of the B-set of a1. In set theory, the B-set of

a1 is said to *include* the B-set of a2. The table of Figure 1.35 shows these couples for the current content of table T.

| A1 | Bset1 | includes | A2 | Bset2 |
|---|---|---|---|---|
| 01 | a,b,c | includes | 02 | b,c |
| 01 | a,b,c | includes | 03 | c |
| 02 | b,c | includes | 03 | c |
| 04 | c,d | includes | 03 | c |

**Figure A1.35 -** The inclusion relationships among the A values of table T

The inclusion problem is quite close to the *relational division*, the famous operator of the relational algebra that is missing in SQL but can be simulated as shown in Section 1.24. We will describe and discuss four algorithms that computes the inclusion relations within table T as shown in Figure 1.35.

## A1.25.2The algorithms

The first algorithm is a direct application of the relational division. Each couple (a1,a2) of A values from table T is examined to check whether the B-set of a2 is a *divisor* of the B-set of a1 (Script 1.85).

```
select distinct T1.A,'includes',T2.A
from   T T1,T T2
where  T1.A <> T2.A
and    not exists (select *
                   from   T T3
                   where  T3.A = T2.A
                   and    not exists (select *
                                      from   T T4
                                      where  T4.A = T1.A
                                      and    T4.B = T3.B
                                      )
                   );
```

**Script A1.85 -** Algorithm 1: through division operator (1)

The second algorithm is another expression of the previous one where the inner `not exist` predicate is rewritten as an equivalent `not in` predicate (Script 1.86).

```
select distinct T1.A,'includes',T2.A
from   T T1,T T2
where  T1.A <> T2.A
and    not exists (select *
                   from   T
                   where  A = T2.A
                   and    B not in (select B from T
                                    where  A = T1.A)
                   );
```

**Script A1.86 -** Algorithm 2: through division operator (2)

The third algorithm (Script 1.87) is based on the following property:

> set S2 is a part of set S1 if the difference S2 - S1 is empty, that is, there is no element of S2 that is not in S1.

```
select distinct T1.A, 'includes',T2.A
from   T T1,T T2
where  T1.A <> T2.A
and    not exists (select B from T where A = T2.A
                     except
                   select B from T where A = T1.A);
```

**Script A1.87 -** Algorithm 3: through set difference

The last algorithm follows a quite different approach. It is based on the query of intermediate table ABSET (computed by Script 1.84) that computes the list of B values for each A value. For each couple (a1,a2) of distinct A values from table ABSET, the query verifies that the first one *includes* the second one. The inclusion relation is evaluated by SQLfast function **itemInclude** (Script 1.88).[7]

```
select S1.A,'includes',C2.A
from   ABSET C1, ABSET C2
where  C1.A <> C2.A
and    itemInclude(C1.Bset,C2.Bset,',');
```

**Script A1.88 -** Algorithm 4: through SQLfast lists inclusion

7. The arguments of function itemInclude are the *including list*, the *included list* and the *value separator* used to build these lists.

### A1.25.3Performance evaluation

We check our algorithms on a table T that contains 512 rows. The algorithms generate 4.520 inclusion relationships. The computing times of each algorithm for various combinations of indexes are reported in table 1.36.

| | Algorithm 1 | Algorithm 2 | Algorithm 3 | Algorithm 4 |
|---|---|---|---|---|
| 1. no index | 27.78 | 38.69 | 25.34 | **0.62** |
| 2. primary key (A,B) | **0.46** | **3.33** | **1.73** | **0.63** |
| 3. primary key (B,A) | 8.31 | 38.38 | 25.83 | **0.63** |
| 4. primary key (A,B) + index on B | 0.46 | 3.34 | 1.72 | **0.62** |
| 5. index on A | 1.24 | 4.03 | 2.02 | **0.63** |
| 6. index on B | 13.94 | 39.04 | 25.53 | **0.63** |

**Figure A1.36 -** Comparison of the execution time (in seconds) of the four algorithms

**Comments**

- Without index, the fourth algorithm is, by far, the faster.
- The fourth algorithm doesn't make use of any index when they are available.
- For algorithms 2 and 3, an index on A, and, even better, an index on (A,B), gives the best performance.
- An index on B is useless except for the first algorithm.

**Conclusions**

The best performance is obtained by the first algorithm with an index on (A,B). Interestingly, this algorithm is the direct derivation of the mathematical definition of the relational division.

When such an index cannot be built (for instance in an intermediate computation or in a view), the fourth algorithm is the best.

Keep in mind, however, that these recommendations are valid for the test case on which the algorithms are applied and that they may vary for larger or smaller problems.

### A1.25.4The set equality problem

Let us first observe that all these algorithms discard the trivial relationships telling that *set E is included into itself*, thanks to conditions `T1.A <> T2.A` or equivalent.

Two sets E1 and E2 are equal if E1 includes E2 and E2 includes E1. There are several techniques to evaluate this property. The more obvious way is to extend one of the algorithm developed above to identify the couples of A values such that the B-set of each (and not just one) is included in the B-set of the other. Another usual technique consists in checking whether E1 includes E2, and E1 and E2 have the

same size. The fourth algorithm is the easiest one to adapt: we just use **itemEqual** function instead of **itemInclude** (Script 1.89).

```
select S1.A,'equal',C2.A
from   ABSET C1, ABSET C2
where  C1.A <> C2.A
and    itemEqual(C1.Bset,C2.Bset,',');
```

**Script A1.89 -** Checking set equality

These techniques will generate duplicates (if E1 is equal to E2 then E2 is equal to E1) that are easy to reduce.

### A1.25.5Transitive reduction

The set inclusion relation is transitive, which means that if set E1 includes E2 and E2 includes E3, then E1 includes E3. Considering an arbitrary set inclusion relation R, if we add to R all the couples obtained recursively by transitivity, we obtain the **transitive closure** (generally noted $R^+$) of R. If, on the contrary, we remove the couples that can be derived by transitivity, we obtain a **transitive reduction** of R,[8] that is, a minimal set of inclusion relationships from which R and $R^+$ can be rebuilt. In the result set of Figure 1.35, we observe that relationship *01 includes 03* can be obtained by the composition of *01 includes 02* and *02 includes 03*. Therefore, it can be discarded, to produce the transitive reduction of Figure 1.37.

| A1 | Bset1 | includes | A2 | Bset2 |
|---|---|---|---|---|
| 01 | a,b,c | includes | 02 | b,c |
| ~~01~~ | ~~a,b,c~~ | ~~includes~~ | ~~03~~ | ~~c~~ |
| 02 | b,c | includes | 03 | c |
| 04 | c,d | includes | 03 | c |

**Figure A1.37 -** Transitive reduction of the inclusion relation of table T

The result set produced by the algorithms of Section A1.25.2 provides all the inclusion relationships that hold in T, including transitive ones, that is, their *transitive closure*. Hence the question: how could we compute its transitive reductions, or at least one of them?

The algorithm is quite simple, at least for the kind of problems we have considered so far.

Let us consider that the result set shown in Figure 1.35 has been stored in table INCLUSION(A1,A2), telling that the B-set of A1 includes the B-set of A2 (Script 1.90).

8. There is only one transitive closure for an inclusion relation but there may be several transitive reductions, as we will observe in the following.

```
create table INCLUSION as
       select S1.A as A1,C2.A as A2
       from   ABSET C1, ABSET C2
       where  C1.A <> C2.A
       and    itemInclude(C1.Bset,C2.Bset,',');
```

**Script A1.90 -** Storing the closure of the inclusion relation in table INCLUSION

We then select in INCLUSION each row I1 for which *there is no two rows* (say, I2 and I3) *the composition of which is I1*. This is what Script 1.91 translates in SQL. The result set for table T is shown in Figure 1.38.

```
select A1,A2
from   INCLUSION I1
where  not exists (select *
                   from   INCLUSION I2, INCLUSION I3
                   where  I2.A1 = I1.A1
                   and    I2.A2 = I3.A1
                   and    I3.A2 = I1.A2)
```

**Script A1.91 -** Computing the transitive reduction of table INCLUSION

| A1 | A2 |
|---|---|
| 01 | 02 |
| 02 | 03 |
| 04 | 03 |

**Figure A1.38 -** Transitive reduction of table INCLUSION

### A1.25.6Transitive reduction with equality relationships

Unfortunately, this algorithm works fine for *acyclic* graphs only. Circuits will appear as soon as (at least) two B-sets are equal. Indeed, if the B-sets of a1 and a2 are equal, then table INCLUSION contains rows (a1,a2) and (a2,a1), that, together, form a circuit. Let us illustrate the problem with the contents of INCLUSION shown in Figure 1.39 (both in tabular and graphical representation), in which 01 and 02 are equal (i.e., have the same B-set) and form a circuit.

| A1 | A2 |
|---|---|
| 01 | 02 |
| 02 | 01 |
| 01 | 03 |
| 02 | 03 |

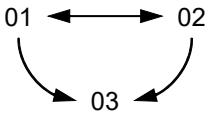



**Figure A1.39 -** INCLUSION table including circuit (01,02)

The analysis of these rows shows that:

– (01,03) is the composition of (01,02) and (02,03)

– (02,03) is the composition of (02,01) and (01,03)

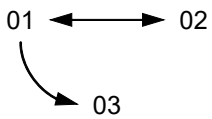
and therefore can be removed, leading to the following transitive reduction:

| A1 | A2 |
|---|---|
| 01 | 02 |
| 02 | 01 |

which obviously is incorrect! The flaw of this algorithm is that a row that has been proved to be transitive (e.g., (01,03)) is further used to prove that another row (e.g., (02,03)) also is transitive. Therefore, both (01,03) and (02,03) are discarded.

There are two correct transitive reductions, one which removes (01,03) and the other which removes (02,01) (Figure 1.40).

| A1 | A2 |
|---|---|
| 01 | 02 |
| 02 | 01 |
| 01 | 03 |




| A1 | A2 |
|---|---|
| 01 | 02 |
| 02 | 01 |
| 02 | 03 |

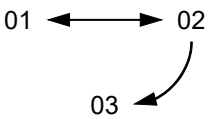



**Figure A1.40 -** The two valid transitive reductions

To study this problem, we will rely on the larger example of Figure 1.41. It shows that A values 01, 02, 03 have the same B-set and form a circuit. Similarly, a second circuit is formed with A values 04 and 05.

| A | BSet |
|---|---|
| 01 | a,b,c,d |
| 02 | a,b,c,d |
| 03 | a,b,c,d |
| 04 | a,b,c,d,e |
| 05 | a,b,c,d,e |
| 06 | a,b,c |
| 07 | a,b |

**Figure A1.41 -** Agregated view of the contents of T table

The transitive closure of the inclusion relation is represented in Figure 1.42 as an aggregated view of table INCLUSION.

| A1 | A2 |
|---|---|
| 01 | 02,03,06,07 |
| 02 | 01,03,06,07 |
| 03 | 01,02,06,07 |
| 04 | 01,02,03,05,06,07 |
| 05 | 01,02,03,04,06,07 |
| 06 | 07 |

**Figure A1.42 -** Agregated view of the contents of INCLUSION table

The graphical representation of a transitive reduction of the inclusion relation is shown in Figure 1.43. The two sets of equal A values are represented on a grey background. Equivalent reductions can be derived by replacing (04,01) by (04,02), (04,03), (05,01), (05,02) or (05,03) and replacing (01,06) by (02,06) or (03,06). So, there are eighteen reductions of the closure of Figure 1.42.

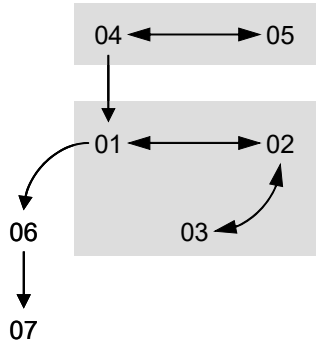



**Figure A1.43 -** One of the eighteen valid transitive reductions of the inclusion relation

Now, the question is, how can we compute these transitive reductions, or at least one of them? We will examine the idea illustrated by Figure 1.44, in which all the A values with the same B-set have been merged to form *super* A values 01-02-03 and 04-05. These super A values can be considered *circuit denotations*.[9] In this way, all the members of each circuit are collectively replaced by the denotation of their circuit. The circuits being removed (actually hidden), the graph becomes acyclic and is unique.

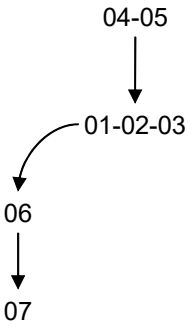



**Figure A1.44 -** Transitive reduction in which the members of each circuit have been reduced to one *super* A value.

To implement this solution, we build the table DICTIONARY that maps the members of each circuit to the denotation of their circuit (Figure 1.45). If an A value does not belong to a circuit, it is left unchanged. Applying the substitutions of this dictionary, we derive the new version of the transitive closure (Figure 1.47), that is now acyclic.

| A | Substitute |
|---|---|
| 01 | 01-02-03 |
| 02 | 01-02-03 |
| 03 | 01-02-03 |
| 04 | 04-05 |
| 05 | 04-05 |
| 06 | 06 |
| 07 | 07 |

**Figure A1.45 -** Table DICTIONARY mapping all A values to the denotation of their circuit (if any)

To compute the dictionary, we first define view BSET_ASET, which creates the list of A values with which the same B-set is associated (Figure 1.46). Then, we join table ABSET with this view on column Bset (Script 1.92).

9. Concatenating A values of a circuit is not the only way to denote it. Since circuits are disjoint, any one of their members identifies them, for instance the lowest of their values.

| BSet | Size | ASet |
|---|---|---|
| a,b | 1 | 07 |
| a,b,c | 1 | 06 |
| a,b,c,d | 3 | 01-02-03 |
| a,b,c,d,e | 2 | 04-05 |

**Figure A1.46 -** Intermediate view BSET_ASET showing to which A value(s) each B-Set is associated

```
create view BSET_ASET(BSet,Size,ASet) as
       select BSet,count(*),group_concat(A,'-')
       from   ABSET group by BSet;

create table DICTIONARY as
       select ABSET.A as A, BA.ASet as Substitute
       from   BSET_ASET BA, ABSET
       where  BA.BSet = ABSET.BSet;
```

**Script A1.92 -** Creating table DICTIONARY

Now, we create the view GR_INCLUSION, the variant of INCLUSION where each A1 and A2 value is replaced by its substitute from table DICTIONARY (Script 1.93). This view provides the acyclic transitive closure of the inclusion relation that holds in T.

```
create view GR_INCLUSION(A1,A2) as
       select distinct D1.Substitute, D2.Substitute
       from   INCLUSION I, DICTIONARY D1, DICTIONARY D2
       where  D1.A = I.A1 and D2.A = I.A2
       and    D1.Substitute <> D2.Substitute;
```

**Script A1.93 -** Converting table INCLUSION into GR_INCLUSION

| A1 | A2 |
|---|---|
| 01-02-03 | 06 |
| 01-02-03 | 07 |
| 04-05 | 01-02-03 |
| 04-05 | 06 |
| 04-05 | 07 |
| 06 | 07 |

**Figure A1.47 -** Translation of the content of INCLUSION table where all the members of each circuit have been replaced by its denotation

Its reduction through the algorithm of Script 1.91 is straightforward. It is represented in Figure 1.48.

| A1 | A2 |
|---|---|
| 01-02-03<br>04-05<br>06 | 06<br>01-02-03<br>07 |

**Figure A1.48 -** Transitive reduction of the inclusion relation of Figure 1.42

Finally, we have to decide what to do with the members of the circuits and their inclusion relationships. We suggest two techniques:

– The circuits are described in a separate table, which just happens to be a part of our dictionary!

– We complete the reduction table of Figure 1.48 with the inclusion relationships internal to each circuit.

These scripts are available in files **Expressing_set_inclusion.sql** and **Computing_transitive_reduction.sql**.

## A1.26 Solving logic puzzles

Applying database principles to solve logic puzzles is one of the most exciting of SQL applications. Many of them rely on a set of conditions that must simultaneously be satisfied by a set of entities. Since SQL is based on first-order logic (+ some non orthodox but convenient features, such as recursive CTE, multiset algebra and meta-data) solving problems of this kind shouldn't be too complicated. Or would it be?

### A1.26.1The dwellers

Finding the order of a series of objects from some hints on local properties of this order is particular popular. This one is borrowed from Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs* - 2nd Edition, (p. 568) 1996, MIT Press

> *Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?*

#### The data structures

This statement tells us that five persons must be placed into five floors. They are denoted by their name and the floors by their position (1 is the bottom floor and 5 the top one). We store in table LIVES information on *who* (column Name) lives *where* (column Floor). Initially, this table contains all the possible (Name, Floor) combinations (Script 1.94). The solution is somewhere among the 25 rows of this table!

Now, let us extract the constraints expressed in the problem statement and let us translate them in (pseudo) SQL.

```
create table LIVES as
with
  NAMES(Name) as
    (values ('Baker'),('Cooper'),('Fletcher'),('Miller'),('Smith')),
  FLOORS(Floor) as (
    (values (1),(2),(3),(4),(5))
select Name,Floor from NAMES,FLOORS;
```

**Script A1.94 -** Generating all the *(person,floor)* combinations

#### The constraints and their SQL translation

In the analysis below, we note "X.Floor" the floor of person X.

- *Baker does not live on the top floor*

```
Baker.Floor < 5
```

- *Cooper does not live on the bottom floor*

```
Cooper.Floor > 1
```

- *Miller lives on a higher floor than does Cooper*

```
Cooper.Floor < Miller.Floor
```

- *Smith does not live on a floor adjacent to Fletcher's*

```
abs(Cooper.Floor - Fletcher.Floor) > 1
```

- *Fletcher does not live on a floor adjacent to Cooper's*

```
abs(Smith.Floor - Fletcher.Floor) > 1
```

- *Fletcher does not live on either the top or the bottom floor*

```
Fletcher.Floor between 2 and 4
```

- *Baker, Cooper, Fletcher, Miller, and Smith live on different floors*

  We chose the simplest translation, if not the most elegant (what we generally call a *brute force method*.

```
not (   Baker.Floor  = Cooper.Floor
     or Baker.Floor  = Fletcher.Floor
     or Baker.Floor  = Miller.Floor
     or Baker.Floor  = Smith.Floor
     or Cooper.Floor = Fletcher.Floor
     or Cooper.Floor = Miller.Floor
     or Cooper.Floor = Smith.Floor
     or Fletcher.Floor = Miller.Floor
     or Fletcher.Floor = Smith.Floor
     or Miller.Floor   = Smith.Floor)
```

  This constraint can also be integrated into the cross join that builds the set of all possible floor assignments (Script 1.94).

### Looking for the floor assignment solution

We now build all the possible 5-uples (i.e., sequences of five floors) such that each element is a tentative place of a definite person. Each row represents a candidate floor assignment of the five persons. Without constraints, this set, generated by the cross join of five instances of LIVES (one for each person), includes 9,765,625 rows, of which only one is valid![10]

The code of Script 1.95 completes the query thats builds this set with the constraints translated from the problem statement.

10. Actually, this set is fictitious and will not be computed as such by the SQL engine.

```
select distinct
       Baker.Floor    as FBaker,
       Cooper.Floor   as FCooper,
       Fletcher.Floor as FFletcher,
       Miller.Floor   as FMiller,
       Smith.Floor    as FSmith
from LIVES Baker, LIVES Cooper, LIVES Fletcher,
     LIVES Miller, LIVES Smith
where Baker.Floor < 5
and   Cooper.Floor > 1
and   Cooper.Floor < Miller.Floor
and   abs(Cooper.Floor - Fletcher.Floor) > 1
and   abs(Smith.Floor - Fletcher.Floor) > 1
and   Fletcher.Floor between 2 and 4
and   not (   Baker.Floor  = Cooper.Floor
           or Baker.Floor  = Fletcher.Floor
           or Baker.Floor  = Miller.Floor
           or Baker.Floor  = Smith.Floor
           or Cooper.Floor = Fletcher.Floor
           or Cooper.Floor = Miller.Floor
           or Cooper.Floor = Smith.Floor
           or Fletcher.Floor = Miller.Floor
           or Fletcher.Floor = Smith.Floor
           or Miller.Floor   = Smith.Floor);
```

**Script A1.95 -** Assigning a floor to each person

Its execution produces this (unique) solution in 0.075 s:

```
+--------+---------+-----------+---------+--------+
| FBaker | FCooper | FFletcher | FMiller | FSmith |
+--------+---------+-----------+---------+--------+
| 3      | 2       | 4         | 5       | 1      |
+--------+---------+-----------+---------+--------+
```

## A1.26.2The eight queens problem

*The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.*[11]

The queen can move anywhere in its row, in its column and in its diagonals; thus threatening quite many other pieces on the chessboard. This makes the puzzle far from easy! Figure 1.50 shows one of the solutions of the eight queens puzzle.

11. From https://en.wikipedia.org/wiki/Eight_queens_puzzle

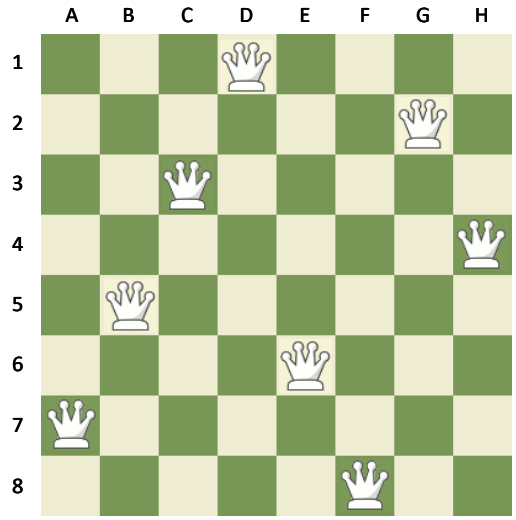



**Figure A1.49 -** One of the solutions ( "75316824") of the eight queen problem

So, when we place a queen in a definite square, we must check that

1. its row is free,
2. its column is free
3. and its two diagonals are free as well.

These constraints are illustrated in Figure 1.50, in which the queen in position D2 condemns 23 positions, therefore preventing us from placing the next queens there.

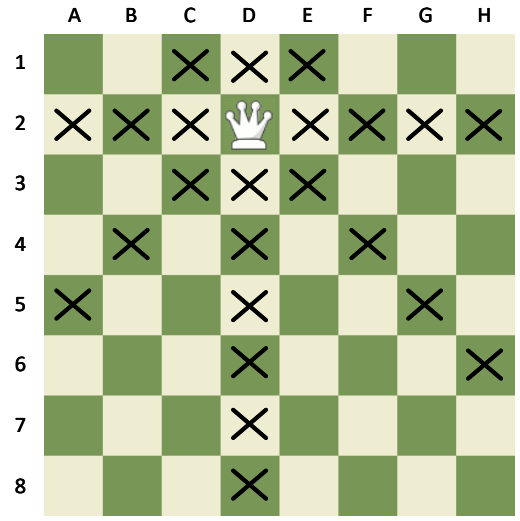



**Figure A1.50 -** Placing a queen in D2 condemns 23 positions

To denote a solution, or more generally a layout of the eight queens (whether it is a valid arrangement or not), we use a 8-digit character string, where the digit in position C designates the row in which the queen is placed in column C. "7**5**316824" is a valid layout (that of Figure 1.50) in which a queen is placed in row **5** of column **2** (second digit).

Let us first create a base table, called BASE, comprising digits 1 to 8, that will be used to denote rows and columns. The Script 1.96 suggests two queries to create this table.

```
create table BASE as
   with recursive NUM(N) as (values (1)
                                union all
                             select N+1 from NUM where N < 8)
   select cast(N as char) N from NUM;

create table BASE as
   with NUM(N) as values (1),(2),(3),(4),(5),(6),(7),(8))
   select N from NUM;
```

**Script A1.96 -** Two ways to create a set of eight digits

The most obvious *brute force method* we can imagine consists in generating all the sequences of 8 digits (i.,e., arrangements with repetitions), then eliminating those which violate the three constraints.

In Script 1.97, these sequences are created by a simple cross join of eight instances of BASE. Considering each row as a (naive) tentative layout of our eight queens, it (trivially) satisfies constraint n° 1 (no two queens in the same column). However, the result is impressive: $8^8$ = 16,777,216 rows![12] We should be able to do better.

```
create table LAYOUT as
  select A.N||B.N||C.N||D.N||E.N||F.N||G.N||H.N as Solutions
  from  BASE A,BASE B,BASE C,BASE D,BASE E,BASE F,BASE G,BASE H
```

**Script A1.97 -** Creating all the arrangements of 8 digits with repetitions

Now, consider the creation of arrangements in which all digits are distinct, that is, they are *permutations*. This set satisfies the first two constraints: no two queens in the same column nor in the same row.

The query of Script 1.98 creates the desired permutations. It extends the query 1.97 by checking that the digit in position k is distinct from digits 1..k-1.

12. Needless to recall that this set will not be evaluated by the SQL engine.

```
create table LAYOUT as
  select A.N||B.N||C.N||D.N||E.N||F.N||G.N||H.N as Solutions
  from  BASE A,BASE B,BASE C,BASE D,BASE E,BASE F,BASE G,BASE H
  where B.N not in (A.N)
  and   C.N not in (A.N,B.N)
  and   D.N not in (A.N,B.N,C.N)
  and   E.N not in (A.N,B.N,C.N,D.N)
  and   F.N not in (A.N,B.N,C.N,D.N,E.N)
  and   G.N not in (A.N,B.N,C.N,D.N,E.N,F.N)
  and   H.N not in (A.N,B.N,C.N,D.N,E.N,F.N,G.N)
```

**Script A1.98 -** Creating all the permutations of 8 digits

This leaves us with the validation of 8! = 40,320 permutations against the third constraints: no two queens in any diagonal.

The rule checking this property is fairly simple: considering queens Q1 and Q2, with coordinates (c1,r1) and (c2,r2) respectively, they are in the same diagonal *if their vertical and horizontal distances are equal*. That is, if[13]

```
|r1 - r2| = |c1 - c2|
```

Therefore, these queens are in safe positions if

```
|r1 - r2| <> |c1 - c2|
```

Going back to query 1.98, we translate this rule as follows:

> *considered from left to right, each digit is successively compared to each of its predecessor; their absolute différence must be different from the difference of their column positions.*

For example, the current rows of columns A and D (horizontal distance = 3) are compatible if their difference (in absolute values) is different from 3, which translates in SQL:

```
abs(D.N - A.N) <> 3
```

Checking the diagonal constraints requires 1+2+3+...+7 = 28 SQL conditions. The Script 1.99 is the final version of the query solving the eight queens puzzle. It generates all the 92 valid layouts[14] in 0.01 second. Some of them are shown in Figure 1.51.

13. Numeric operator |e| returns the absolute value of expression e.
14. Actually, only 12 solutions (the *fundamental* solutions) need to be computed. All the others can be retrieved by rotating and flipping them.

```
create table LAYOUT as
  select A.N||B.N||C.N||D.N||E.N||F.N||G.N||H.N as Solutions
  from  BASE A,BASE B,BASE C,BASE D,BASE E,BASE F,BASE G,BASE H
  where B.N not in (A.N)
  and   C.N not in (A.N,B.N)
  and   D.N not in (A.N,B.N,C.N)
  ...
  and   H.N not in (A.N,B.N,C.N,D.N,E.N,F.N,G.N)
  --
  and  abs(B.N - A.N) <> 1
  --
  and  abs(C.N - B.N) <> 1
  and  abs(C.N - A.N) <> 2
  --
  and  abs(D.N - C.N) <> 1
  and  abs(D.N - B.N) <> 2
  and  abs(D.N - A.N) <> 3
  ...
  --
  and  abs(H.N - G.N) <> 1
  and  abs(H.N - F.N) <> 2
  and  abs(H.N - E.N) <> 3
  and  abs(H.N - D.N) <> 4
  and  abs(H.N - C.N) <> 5
  and  abs(H.N - B.N) <> 6
  and  abs(H.N - A.N) <> 7;
```

**Script A1.99 -** Finally, we add the filter on the diagonal constraint

```
+-----------+
| Solutions |
+-----------+
| 15863724  |
| 16837425  |
| 17468253  |
| 17582463  |
| 24683175  |
| 25713864  |
| 25741863  |
| ...       |
| 73168524  |
| 73825164  |
| 74258136  |
| 74286135  |
| 75316824  |
| 82417536  |
| 82531746  |
| 83162574  |
| 84136275  |
+-----------+
```

**Figure A1.51 -** The solutions of the eight queens puzzle (excerpts

Now we discuss another way to generate the solutions of the eight queen puzzle. First, we consider that the most elegant way to compute permutations naturally relies on a recursive CTE. To compute the permutations of eight digits:

1. we start from the 8 permutations of length 1
2. we add to each of these permutations each digit that is not included yet
3. and so on until the permutations of length 8 have been generated.

The SQL query that implements this algorithm is shown in Script 1.100, an alternative to Script 1.98.

```
with recursive
PERM(Digits) as
 (select N from BASE
   union all
  select PERM.Digits||B1.N
  from  PERM,BASE B1
  where PERM.Digits not like '%'||B1.N||'%'
  and   length(PERM.Digits) < 8)
select Digits as Permutations from PERM
where  length(PERM.Digits) = 8
```

**Script A1.100 -** An alternative way to generate permutations of length 8

Now we add the SQL condition that translates the diagonal constraint. To this aim, we compute and compare the vertical and horizontal distances between the current element to insert and each element already inserted. This is a bit more tricky than in the first version because we have to apply arithmetic operators on digits that are represented by characters. Here are the elements of the SQL expression of the diagonal constraint.

– the candidate digit under examination will be inserted in position (i.e., *column*)

```
length(PERM.Digits) + 1
```

– the candidate digit must be compared with the existing digits, that is, those in columns `1` to `length(PERM.Digits)` of `PERM.Digits`

– the column numbers of the existing digits are provided by the rows of table BASE (a new instance different from that used to compute permutations; let's call it B2) such that `B2.N <= length(PERM.Digits)`

– the horizontal distance is computed by

```
length(PERM.Digits) - B2.N + 1
```

– the row of an existing column in the permutation is known as a numeric character that must be converted into an integer value:[15]

```
cast(substr(PERM.Digits,B2.N,1) as int)
```

– the vertical distance is computed between `B1.N`, the row of the candidate column, and the row of each existing column in the permutation:

```
abs(B1.N - cast(substr(PERM.Digits,B2.N,1) as int))
```

– finally, we exclude the candidate digit if its two distances are equal
`not exists (... and` *horizontal distance* `=` *vertical distance*`)`

Note that the instances of BASE in computing the permutations (alias B1) and that in the expression of the diagonal constraint (alias B2) denote different objects. In the first case, B1 designates the *row* of the column to insert, while, in the second case, B2 designates the *rank* of an existing column.

The diagonal constraint is printed in blue in Script 1.101. To make this query fully independent, we have integrated the definition of BASE as a preliminary CTE.

The execution time of this query is a bit higher but still close to that of the first form.

```
create table LAYOUT as
  with recursive
  BASE(N) as (values (1)
                union all
              select N+1 from BASE where N < 8),
  PERM(Digits) as
   (select N from BASE
     union all
    select PERM.Digits||B1.N
    from  PERM,BASE B1
    where PERM.Digits not like '%'||B1.N||'%'
    and   length(PERM.Digits) < 8
    and   not exists
           (select *
            from  BASE B2
            where B2.N <= length(PERM.Digits)
            and length(PERM.Digits) - B2.N + 1
                =
                abs(B1.N - cast(substr(PERM.Digits,B2.N,1)
                              as int)
                    )
           )
   )
  select Digits as Permutations from PERM
  where  length(PERM.Digits) = 8;
```

**Script A1.101 -** Solving the eight queens puzzle through a CTE

15. Function `substr(S,P,L)` extracts from character string `S`, the substring of length `L` starting from position `P`. In SQL, the first position is 1.

The CTE form may seem less intuitive than the former one, but it enjoys a particularly interesting property: *scalability* with respect to the problem dimension, that is, the number of queens to place.

Extending the first form (Script 1.99) to, say, 9 queens to place on a 9x9 chessboard requires (1) adding an instance of BASE in the join, (2) adding a new condition on the horizontal constraint and (3) 8 new conditions on the diagonal constraint, that is, a deep modification of the *structure of the query*.

On the other hand, the CTE form just requires changing the constant 8 to 9, that is, changing *the value of a parameter*. Since this parameter appears in three different places in the query, we replace it with an SQLfast variable as shown in Script 1.102.

```
set dim = 9;
create table LAYOUT as
  with recursive
  BASE(N) as (...
              select N+1 from BASE where N < $dim$),
  PERM(Digits) as
   (...
    and   length(PERM.Digits) < $dim$
    ...)
  select Digits as Permutations from PERM
  where  length(PERM.Digits) = $dim$;
```

**Script A1.102 -** Parametrization of the CTE query

## Warning

These algorithms work fine for puzzles of dimension 1 to 9. For greater values, we cannot model the state of a column with a single digit any more. Coding column ranks with single letters would be more appropriate, the computations and comparisons on them being performed on their ASCII codes.

These scripts are available in file **Solving-logic-puzzles.sql**.

## A1.27 Nest and unnest operators

(Very) short introduction to tables in *Non First Normal Form* and their *nest* and *unnest* relational operators. This topic is important considering the advanced data structures allowed in most current SQL versions, such as arrays, embedded tables, object-relational databases and JSON column type.

### A1.27.11NF and N1NF tables

Let us consider a table **T** through which, to each person, we associate a *set of cities*, possibly empty (Figure 1.52). It has been created by Script 1.103. As usual, the absence of cities is represented by a *null* value.

Since the column *Cities* may comprise several individual values, it is referred to as *multivalued*. A table all the columns of which are single-valued is said to be in *First Normal Form* (*1NF* or *flat structure*). Each row of this table is made up of a single individual value per column. Conversely, and logically, a table some of the columns of which are multivalued is said to be in *Non First Normal Form* (*N1NF* for short) according to the relational theory. Though N1NF structures can be considered natural from the user's point of view, they raise complex issues that make them undesirable. One of them is the complexity of data manipulation.

| Id | Name | Cities |
|---|---|---|
| 1 | Adam | London,Paris,Madrid |
| 2 | Brian | Paris,Genève |
| 3 | Christie | -- |
| 4 | David | London |

**Figure A1.52 -** Table T is in N1NF

```
createDB InMemory;
create table T(Id integer,Name varchar(32),Cities varchar(32));
insert into T values (1,'Adam',    'London,Paris,Madrid'),
                     (2,'Brian',   'Paris,Genève'),
                     (3,'Christie', null),
                     (4,'David',   'London');
```

**Script A1.103 -** Creating N1NF table T

### A1.27.2The *unnest* operator

The *unnest* operator converts a N1NF table into a (hopefully) equivalent 1NF table. In a sense, it *flattens* N1NF tables. It is coded by the query of Script 1.104 and, applied to table T, produces the table T_unnest of Figure 1.53. This query creates and populates table T-unnest through recursive CTE S that extracts the successive values of column Cities. More specifically, it works as follows.

– The values of Cities are processed as SQLfast lists. Each value of this column is a comma-separated list of city names.
– CTE TT defines a view of T in which the *null* values of Cities are replaced by empty strings, much easier to manipulate.
– Recursive CTE S is the core of the transformation. For each row of T, it produces a sequence of rows comprising the identifier of a customer (Id), one of the values from its Cities list (City) and the index of this value in the Cities value (Idx). For each row of T:
  • the starting query creates a fictitious row that initializes index Idx to 0,
  • the recursive query increments index Idx and extracts the value of the Cities list at position Idx,
  • the iteration stops when the value of Idx is equal to the length of Cities,
  • we note that when Cities is empty, the CTE generates a row with Idx = 1 and City = ''.
– The final query collects the rows of S, eliminates the fictitious ones, converts empty values into *null* and joins the remaining rows with T to complete them with the name of the customers.

```
create table T_unnest as
with recursive
   TT(Id,Name,Cities) as
      (select Id,Name,case when Cities is null
                           then ''
                           else Cities
                      end
       from   T),
   S(Id,Idx,City) as
      (select id,0,'' from TT
        union all
       select S.Id,S.Idx+1,item(TT.Cities,Idx+1,',')
       from   S,TT
       where  TT.Id = S.Id and S.Idx < itemLen(TT.Cities,',')
       )
select S.Id,T.Name,case when length(S.City) > 0
                        then S.City
                        else null
                   end as City
from   S,T
where  S.Id = T.Id and S.Idx > 0;
```

**Script A1.104 -** An SQL query implementing the unnest operator for table T

The result is shown in Figure 1.53. To compare the 1NF and the N1NF versions of the same data in terms of ease of processing, just try to compute the *number of customers in each city*!

| Id | Name | City |
|---|---|---|
| 1 | Adam | London |
| 1 | Adam | Paris |
| 1 | Adam | Madrid |
| 2 | Brian | Paris |
| 2 | Brian | Genève |
| 3 | Christie | -- |
| 4 | David | London |

**Figure A1.53 -** Table T_unnest, the 1NF equivalent of N1NF table T

### A1.27.3The *nest* operator

The inverse operator is called *nest*. Considering 1NF table T1, partitioned into two subsets of columns, C1 and C2, the *nest* operator produce N1NF table T2 comprising the columns of C1 and a new multivalued column, say C2s, that contains the set of values of C2 associated in T1 to a definite value of C1. *Nesting* a column, or a set of columns is straightforward thanks to the SQL `group_concat` function. The query of Script 1.105 nests the column City of the normalized table T_unnest to rebuild the N1NF version.

```
select id,Name,group_concat(City) as Cities
from   T_unnest
group by Id,Name;
```

**Script A1.105 -** Nesting the column City of 1NF table T_unnest to rebuild the N1NF table of Figure 1.52

### A1.27.4Are the *nest* and *unnest* really reversible?

In other words, does nesting the unnested version of T always regenerate T? The theory says *yes*, as long as the source N1NF table T enjoys the following property[16]:

*T has a primary (or at least a unique) key, and this key comprises single-valued columns only.*

16. A N1NF table that satisfies this property is said to be in *Partitioned Normal Form*. The complete definition of this form is a bit more detailed but is sufficient for tables such as T in this study.

These scripts are available in file **Unnest-Nest-Operators.sql**. It also includes a parametric version of the unnest operator.

# Index

## Symbols

- (arithmetic operator) 150
- (unary arithmetic operator) 150
Symbols
! 214, 216
# (index clause) 121
$ $ (substitution operator) 39, 98
% (binary arithmetic operator) 150
& (binary operator) 155
(block separator) 223
* (arithmetic operator) 150
* (SQL shorthand) 75
+ (arithmetic operator) 150
/ (arithmetic operator) 150
/b 179
/w 184
/x 185, 198, 225
/y 185, 198, 225
< (comparison operator) 155
<< (binary operator) 155
<= (comparison operator) 155
<> (comparison operator) 155
= (comparison operator) 155
=> (comparison operator) 155
> (comparison operator) 155
>> (binary operator) 155
>> marker 368
@0 257
@A 257
@h 257
@n 179, 183, 257
@S 179, 211, 257
@s 179, 186, 257
@t 179, 183, 257
@v 257
[]
components of askCombo 222
for loop header sequence 123, 134
index range 121
message field in GUI 179
_system 170
{} (title) 207
| (binary operator) 155
| (block separator) 223
| (label separator) 59, 180, 206
|| (block separator) 223
|| (concatenation operator) 101, 151, 217
|| (label separator) 67, 206
~ (binary operator) 155
§ § (substitution operator) 63, 98
Numerics
2D menu 212
A
abortDB (statement) 49
abs (numeric function) 150
addToDate (time function) 157
affinity typing 278
aggregate function 79
alter table (statement) 74
and (logical operator) 155
array (simulation) 104
ASCII 240
ask (statement) 41, 59, 63, 178
askCombo (statement) 222
askData (statement) 42, 178
askData-u (statement) 68
askText (statement) 182
askText-u (statement) 182
ask-u (statement) 68, 181
avg (aggregate function) 79
B
b64decode (character function) 149, 154
b64decode (statement) 149, 246
b64encode (character function) 149, 153, 288
b64encode (statement) 149, 246

bar (output format) 256
base64 encoding 149, 245, 288
base64Compliant (character function) 154
between (comparison operator) 74, 155
BfileToVariable (statement) 148, 238
binary file 236, 239, 242
Binary Large Object 237
binary value 148
blank line 53
BLOB 237
button Cancel 42, 137, 178
button Continue 200
button grouping 208
button OK 41, 60, 181
button Play 200, 352
bytecode 368
C
case (general function) 78, 159
cast (general function) 159
char2hex (character function) 154, 245
character string expression 151
check predicate 74
clock (time function) 158
closed interval 135
closeDB (statement) 38
closeHelp (statement) 202
closeTextFile (statement) 236
coalesce (general function) 78
color (system variable) 228
column alias 76
column break || 67
column type analysis 291
comment (statement) 36
commitDB (statement) 48
common table expression (statement) 299
complex construct 368
composite dialogue box 220
compute (statement) 100, 147, 355
condition (syntax) 113
constSeparator 214
constSeparator, 190
control character 179, 194
copyFile (statement) 174, 236
count (aggregate function) 79
count distinct (agregate function) 79
countInst (character function) 152
cp1252 241
cp850 241
create index (statement) 74
create table (statement) 46, 72
create trigger (statement) 315
create view (statement) 85, 299
createDB (statement) 45
createDictionary (statement) 332, 340
createOrReplaceDB (statement) 46
cryptography 248
CSV data export 284
CSV data import 284
CSV output format 259
CSV syntax 285
csv-header (parameter) 287
csv-maxerrornbr (parameter) 289
CTE (statement) 299
CTE limits 311
current_date (system variable) 78
current_date (time function) 156
current_date_string (time function) 157
current_time (system variable) 78
current_time (time function) 156
current_time_string (time function) 157
current_timestamp (system variable) 78
current_timestamp (time function) 156
current_timestamp_full (time function) 157
cyclic data 306
D
data (output format) 256
data validation 139

database schema 329, 340
date (system variable) 91
date (time function) 156
datePattern (time function) 158
datetime (time function) 156
day_in_month (time function) 158
dBase data import 293
dbDirectory (system variable) 171
DBF file format 294
DB-MAIN 52
deaccent (character function) 152
debugging 367
decrypt (character function) 154, 248
default value 72
delete (statement) 62, 73, 84
deleteDictionary (statement) 340
deleteFile (statement) 174
DIALOGbutton (system variable) 117, 181, 194
Dictionary (parameter) 339
directory name 170
dirExists (file/url function) 159
dirFileNames (statement) 172
dirFileNamesAll (statement) 172
dirNames (statement) 172
dirNamesAll (statement) 172
display parameter 184, 186, 195, 198
displayHelp (statement) 202
dispNull (parameter) 287
distinct (modifier) 76
doubleQuote (character function) 153
drop index (statement) 74
drop table (statement) 72
dropVariable (statement) 98
dynaColumns (system variable) 405
dynaMarker (system variable) 405
dynamic column 404
E
elapsed time 92
else (statement)) 110
embedded loops 140
empty label 210
encrypt (character function) 154, 248
endif (statement) 108
endsWith (character function) 153
Enter key 181
entryLength (system variable) 228
Esc key 181
except (SQL operator) 81, 84
except all (SQL operator) 81
execApp (statement) 101, 161, 177, 354
execProg (statement) 177, 202, 263, 356
execSQL (statement) 52, 263
exists (SQL function) 76
exit (statement) 138
exporting binary values 287
external library 161
external SQLfast application 354
external SQLfast function library 355
extract (statement) 68
extract (time function) 78
F
fastExec (statement) 49
field grouping 207
file for loop (block statement) 135, 235
file name 170
fileDirectory (system variable) 171, 286
fileExists (file/url function) 159
fileNames (statement) 172
fileNamesAll (statement) 172
fileToVariable (statement) 197, 235
for each row (SQL clause) 86
for each statement (SQL clause) 86
for loop (elementary) 123
foreign key 50, 53, 62, 72
for-endfor (block statement) 115, 134, 396
frame (askCombo) 224
frame (character function) 153
from (SQL clause) 38, 75
full name 170
function (statement) 102, 161, 177,

355
G
general function 159
generalDirectory (system variable) 170
gif file 196, 199, 349
goto (statement) 109
grid 224
group by (clause) 74
group by (SQL clause) 81
group_concat (aggregate function) 82, 346
group_concat2 393
group_concat2 (aggregate function) 83, 160, 346
GUI 177, 193, 205
H
hash (character function) 154
hashFile (statement) 248
having (SQL clause) 81
head (output format) 256
helpDirectory (system variable) 171
heterogeneous composite box 224
hex (character function) 153
hex2char (character function) 154, 245
hexadecimal encoding 148, 244
hexCompliant (character function) 154
HEXdecode (statement) 149, 244
HEXencode (statement) 149, 244
hour (time function) 158
HTML output format 270
I
if (statement) 108, 148
if-else-endif (block statement) 110
if-endif (block statement) 108
if-goto 368
image size 198
import (statement) 288
ImportErrors (system variable) 289
in (comparison operator) 155
incoding_file (parameter) 238, 241
indexed extract (statement) 121
indexed select (statement) 121, 134
infinite recursion 307
information schema 331
InMemory 55
In-memory database 54
insert (statement) 46, 61, 84
insertLOB (statement) 238
instr (character function) 101, 152
int (numeric function) 150
intersect (SQL operator) 81
intersect all (SQL operator) 81
isDate (time function) 158, 292
isInteger (numeric function) 151, 292
isNotDate (time function) 158
isNotInteger (numeric function) 151
isNotNumeric (numeric function) 151
isNumeric (numeric function) 151, 292
item (function) 392
item (general function) 160
itemDelete (function) 394
itemEqual (function) 396
itemEqualAll (function) 396
itemExcept (function) 395
itemExceptAll (function) 395
itemInclude (function) 396
itemIncludeAll (function) 396
itemIndex (function) 394
itemIndex (general function) 160
itemIndexS (general function) 160
itemInter (function) 395
itemInterAll (function) 395
itemJoin (function) 394
itemLen (function) 394
itemLen (general function) 159
itemLenS (general function) 160
itemS (general function) 160
itemSlice (function) 394
itemSort (function) 394
itemSpeed 397
itemUnion (function) 395

itemUnionAll (function) 395
J
jd (time function) 156
join (SQL operator) 79
join condition 79
JSON 280
JSON data export 296
JSON data import 296
JSON output format 280
julianday (time function) 156
K
Key-Value output format 263
L
label (statement) 109, 114, 123
label-separator (parameter) 180
large objects 234
lastPathNode (file/url function) 159
LateX output format 279
latinCompliant (character function) 154
length (character function) 101, 152, 292
LFile.py (external library) 161, 364
LGeo.py 365
LGeo.py (external library) 365
like (character function) 152
like (comparison function) 75
like (comparison operator) 74, 155
list for loop (block statement) 136, 396
list output format 265
list processing 393
list separator 392
listSeparator 396
loaderDebug 369
local name 170
logical expression 155
loop 115
loop (internals) 142
loop-based recursion 318
lower (character function) 101, 152
LStr.py (external library) 161, 363
LText.py (external library) 161
ltrim (character function) 101, 152
LWeb.py (external library) 161, 364
M
MacRoman 241
main() 355
max (aggregate function) 79
max (agregate function) 101
max (general function) 159
maxPictureX (system variable) 198
maxPictureY (system variable) 198
maxSelectWidth (system variable) 234
maxShowDataWidth (system variable) 234
message field 63, 178
metadata 251, 329
millisecond (time function) 158
min (aggregate function) 79
min (agregate function) 101
min (general function) 159
minute (time function) 158
month (time function) 158
multi-column box 205
MySQL 83, 346
N
new (trigger alias) 86
next (statement) 138
nextDate (time function) 157
nextTextLine (statement) 236
nodeExtension (file/url function) 159
normalizePath (file/url function) 159
normDate (time function) 158
not (logical operator) 155
not in (comparison operator) 155
not null 62, 72
numeric expression 150
numeric functions 100
O
old (trigger alias) 86
on delete (SQL clause) 73
on update (SQL clause) 73

onError (statement) 237
openDB (statement) 32, 36
openTextFile (statement) 236
or (logical operator) 155
order by (SQL clause) 74, 75
ORDERS.db creation script 49
ORDERS.db data 27
ORDERS.db schema 52
ORDERS.db semantics 23
outcoding_file (parameter) 242
outer join (SQL operator) 80
output 372
output (statement) 93
output channel 164
output format 255
output variables 167
output window 165
outputAppend (statement) 165
outputDirectory (system variable) 171
outputmode (parameter) 168
outputOpen (statement) 165, 262, 286
outputtype (parameter) 168
overlapping loop 140
P
parameter (statement) 95, 180, 261
pathPrefix (file/url function) 159
pause (statement) 193
PDFprocessor (system variable) 244
performance 137
picture box 198
picture size 198
portable script 170
power (numeric function) 151
predefined values 213, 215, 217
prevDate (time function) 157
primary key 46, 50, 53, 62, 72
procLevel 386
pureNodeName (file/url function) 159
Python 18, 354, 357, 358
Q
quoted string 40, 42
R
raise (SQLite function) 88
random (numeric function) 94, 150
random_f (numeric function) 151
random_i (numeric function) 150
range for loop (block statement) 134
recursive CTE 300
recursive programming 297
recursive trigger 87, 314
reduce (character function) 153, 234
reduceAll (character function) 153, 234
reference directory 170
relational division 530
relative name 170
renameFile (statement) 174
repeat (character function) 153
replace (character function) 101, 153
round (numeric function) 101, 150
RTF output format 273
rtrim (character function) 101, 152
S
schema-driven 278
schema-less table. 405
scriptDirectory (system variable) 171
scrollbar 183, 198
second (time function) 158
secondary script 50
select (SQL clause) 38, 75
select (statement) 32
select list 76, 129
select0,1,2,3,4 (system variables) 269
select-align (parameter) 261, 264
selectClose (system variable) 269
selectFile (statement) 172, 343
selectIntro (system variable) 269
selectList 190
selectList2 190
selectMany (statement) 188, 397
selectMany-u (statement) 189
selectOne (statement) 187, 344
selectOne-u (statement) 187

self-join 80
semi-colon 44
sendMail (file/url function) 159
set (statement) 39
set-n (statement) 97, 126
showData (statement) 195
showDrawing (statement) 199
showMessage (statement) 193
showPicture (statement) 196
showText (statement) 195
shrink (character function) 153
sign2 (numeric function) 151
sign3 (numeric function) 151
simplify (character function) 153
SQL 71
SQL for loop (block statement) 134
SQL for loop (statement) 123
SQL functions 77
SQL output format 278
SQL2 datatypes 73
SQL-DML data export 296
SQL-DML data import 296
SQLdraw 199, 349
SQLfast compilation 368
SQLfast constant 149
SQLfast dictionary 331
SQLfast error message 37
SQLfast expression 147
SQLfast list 392
SQLfast recursive script 321
SQLfast variable 38
SQLfast.exe 20
SQLfast.ini 180, 184, 198
SQLfastUDFlib.py library 160, 356
SQLite 18, 82, 87
SQLiteUDFlib.py library 161, 356
SQLtuto 18
sqrt (numeric function) 151
standard output file 167
start-timer (statement) 92
startWith (character function) 153
statement customization 65
statement formation rules 36
statement preparation 40
statistical function 79
stdTextX (parameter) 184
stdTextY (parameter) 184
stored procedure (schema object) 86
strftime (time function) 156
style (system variable) 227
subquery 76, 83
substitution (statement) 98
substitution operator 39, 63
substr (character function) 101, 151, 152, 234
substr2 (character function) 152
sum (aggregate function) 79
SYS_COLUMN 332
SYS_INTER_TABLE 335
SYS_KEY 334
SYS_KEYCOMP 334
SYS_TABLE 332
system variable 370
T
table alias 77
table CUSTOMER (description) 23
table CUSTORDER (description) 24
table DETAIL (description) 24
table PRODUCT (description) 24
target variable 134, 178
temporal expression 156
temporary index 75
temporary table 54, 75, 392
temporary trigger 75
temporary view 75
text file 235
textwrap (parameter) 184
thousandSep (character function) 153
time (system variable) 91
time (time function) 156
time expression 156
time_string (time function) 157
timer (system variable) 91
title (in dialogue box) 207
Tkinter 18
traceActive 389

traceAppend 389
traceSQL 389
tracing 367
Tracing settings 374
TRACING.db 372
tree processing 302, 315
tree traversal 313
trigger 367
trigger (schema object) 86
trigger instead of 86
trigger instead of (schema object) 85
trigger tracing 377
trim (character function) 101, 152
True (constant) 155
true branch 108
truncate (numeric function) 150
tuple output format 272
Turing machine 310
TXTprocessor (system variable) 265
U
UDF 160, 355
unicode 241
union (SQL operator) 81, 84
union all (SQL operator) 81, 301
unique constraint 62
unique key 72
update (statement) 61, 64, 85, 307
upper (character function) 101, 152
upper/lower case 55
urlExists (file/url function) 159
user defined function 160, 355
user variable 370
UTF-8 241
utf8Compliant (character function) 154, 292
UTIL-SELECT-parameters-for-CSV.sql 286
UTIL-SELECT-parameters-for-XXX 264
V
value type 113
variable (in loops and if) 141
variable (scope) 60
variable delimiters 39, 94
variable history 382
variable substitution 98, 149
variable substitution (ambiguity) 99
variable substitution (disabled) 96
variableToFile (statement) 148, 197, 235
vector graphics 349
view (schema object) 85
W
wait (statement) 94
when (SQL clause) 86
where (SQL clause) 38, 75
while (statement) 148
while (True) loop 139
while loop 136
while-endwhile (block statement) 115, 137
write (statement) 39
writeMessage() 378
writeParameters 371
writeSystemVariables 370
writeTrace() 378
writeVariables 370
WSprocessor (system variable) 263
X
XML output format 267
Y
year (time function) 157