

SQLfast Case Studies

Problem solving with databases

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 1/10/24

Foreword

Though SQLfast (as well as SQLpy) can be used to develop independent applications[1], its ultimate goal is to support a larger project devoted to *Problem solving with databases*. The idea that underlies this project will not impress database experts. Indeed, based on (and extending[2]) first-order logic and set theory, the database concepts are generally perceived as an ideal expression to describe in a rigorous way complex problems and their solutions. In addition, database technology translates these concepts into powerful software systems and intuitive languages, SQL being the most popular of them.

So far, as some kind of *proof of concept*, the *Problem solving* project has also produced a series of case studies that analyze, model and solve various classes of problems through database concepts and tools. Some examples:

- *Four hours to save the library*. Rapid prototype development. Writing a library application in less than 70 statements.[3]
- *Interactive SQL interpreters*. Helping novice users to practice SQL through a series of minimal interpreters.
- *Schema-less databases*. Non standard database structures. Modeling a rapidly evolving world.
- *Active databases*. Automating the behavior of a complex information system. Migrating the logic layer to the database.
- *Temporal databases*. Automated management and processing of historical data (transaction and valid times). Comprises a temporal algebra engine.
- *Kings of France*. Processing tree-structured data. Application of cyclic data structures and recursive algorithms.
- *The book of which you are the hero*. General graph algorithms. Graph analysis. Text-based adventure games. Automatic novel generation. The game engine (so to speak) uses two tables and comprises five SQLfast statements, among which three SQL queries.

1. For example, the *BIBLIO.exe* application that is currently used to manage the library of a large elementary school in Namur is entirely written in SQLfast.
2. Notably through 3-value logic, multiset algebra, incomplete information, implicit ordering relations, recursion and metadata.
3. Its Python/SQLpy translation (Project_1_-_The_Library_(SQL).py) is included in the SQLpy distribution.

- *Directory management*. Extract the descriptions of the content of selected folders from the host computer or external media. Analyzes, compare and describes their evolution. Search for identical files through their hash signatures.
- *Agent modelling*. Representing the evolution of a population of agents with tables (agent states) and triggers (event-based interaction). Several example models are developed and executed.
- *Conway's Game of Life*. Modeling the evolution of cellular automata. The state generator is made up of two SQL queries only.
- *From data bulk loading to database book writing*. Topological sorting. Also copes with sorting non-acyclic graphs.
- *Classifying objects*. Symbolic classification through *Formal Concept Analysis*.
- *Path finders, rovers and Ariadne's thread*. Shortest path algorithms and applications.
- *Blockchains*. Discovery of the blockchain technology. Includes elements of cryptography.

In many of these studies, the core of the solution is reduced to two or three SQL queries. Each case study is described in a detailed document and is implemented as a fully functional SQLfast program.

The documents are available on the web site of SQLfast (https://bit.ly/3x0fj2i) and the programs are included in the SQLfast distribution (https://bit.ly/3LfUa8z%7C).

The present book is the collection of the case study descriptions currently available.

Jean-Luc Hainaut - Namur, October 2024

Case 1. Four hours to save the library 1
1.1 An emergency situation 1
1.2 First mission: recovering the data 2
1.3 The basic functions 2
1.4 The main window 3
1.5 Registering a new reader 4
1.6 Registering a new book 5
1.7 Borrowing a book 6
1.8 Returning a book 7
1.9 Updating the data of a reader 8
1.10 Querying reader and book data 8
1.11 Conclusion 11
Case 2. Interactive SQL interpreters 1
2.1 Introduction 2
2.2 The smallest interactive interpreter in the world! 2
2.3 A simple but realistic SQL interpreter 5
2.4 Error management 5
2.5 Adding information fields 9
2.6 Adding functions 9
2.7 SQL interpreters as training support 11
2.8 Semantic correctness of a query 12
2.9 An interactive SQL tutor 14
2.9.1 The database 14
2.9.2 The interface 15
2.9.3 The tutor engine 17
2.9.4 Improving the SQL tutor 18
2.10 The scripts 20
Case 3. Schema-less databases - Part 1 1
3.1 Introduction 2
3.2 The Universal table model 2
3.2.1 Generating the Universal table data from source tables 3
3.2.2 From the Universal table back to the standard tables 3
3.2.3 Application 4
3.2.4 Origin of the Universal table model 6
3.2.5 Comments 6
3.3 Column-oriented data model 7
3.3.1 Generating the Column-oriented data from the source tables 7
3.3.2 From the Column-oriented tables back to the source tables 8

3.3.3 Column-oriented model with abstract primary keys 9
3.3.4 About performance 11
3.3.5 Not all Column-oriented models are created equal 11
3.3.6 Comments 12
Case 4. Schema-less databases - Part 2 1
4.1 Introduction 1
4.2 The Key-Value model - Version 1 2
4.2.1 Generating Key-Value data from the source tables 3
4.2.2 The case of composite primary keys 5
4.2.3 Rebuilding the source tables from their Key-Value expression 6
4.2.4 About performance 7
4.2.5 Application 9
4.3 The Key-Value model - Version 2 10
4.4 The Key-Value model - Version 3 11
4.4.1 Application to Description Logic 13
4.5 Not all Key-value models are created equal 18
4.6 The scripts 19
Case 5. Schema-less databases - Part 3 1
5.1 Object view of Key-Value data 1
5.2 Objects with complex properties 3
5.2.1 Multivalued properties 3
5.2.2 Composite multivalued properties 5
5.3 Indexing objects 7
5.4 Comparison of the data models 9
5.5 For once, most Object data models are created equal 10
5.6 The scripts 11
Case 6. Active databases 1
6.1 Introduction 4
6.2 The SQL data structures 4
6.2.1 Static structures 4
6.2.2 Basic constraints 5
6.2.3 Predicates (check constraints) 5
6.2.4 Derived data 5
6.2.5 Stored procedures 6
6.2.6 Triggers 6
6.2.7 Coping with the limitations of the trigger language 7
6.3 Basic applications 10
6.4 Integrity management 10

6.5 Non standard system behavior 12
6.6 Updatable view 13
6.7 Redundancy management and derived data 14
6.8 Data modification journaling 18
6.9 Alerters 21
6.10 Type-subtype implementation 24
6.10.1 About database design 24
6.10.2 The Type-subtype concept 25
6.10.3 Is-a relations in SQL3 26
6.10.4 Is-a relations in SQL2 27
6.10.5 The insert operations 30
6.10.6 The update operations 31
6.10.7 The delete operations 35
6.10.8 Automating type-subtype manager production 35
6.11 Other applications 36
6.12 Building a business application 37
6.12.1 About the case 37
6.12.2 About the case study 38
6.13 The static structure of the database 39
6.13.1 The base tables 40
6.13.2 The transaction tables 42
6.13.3 The initial state of the database 45
6.13.4 The transactions 46
6.14 Implementation of the classical application architecture 50
6.15 Implementation as an active database 52
6.16 Extending the data structures 52
6.17 The user interface 53
6.18 The transactions 55
6.18.1 Registering a customer order 56
6.18.2 Executing a supplier order 61
6.18.3 Registering the payment of an invoice 63
6.18.4 Event architecture of the active database 63
6.19 Verification of an active database 63
6.19.1 Static analysis: checking trigger circuits 64
6.19.2 Dynamic analysis: tracing the execution of the active database 65
6.19.3 Replaying transactions 70
6.20 Limits of this case study 71
6.21 Conclusions, history and extensions 72

6.22 The implementation 75
6.23 Appendix A - The initial state of the database 78
6.24 Appendix B - RETAIL application: task analysis 80
6.24.1 Register basic entities 80
6.24.2 Register a customer order 80
6.24.3 Register a customer payment 82
6.24.4 Execute a supplier order 82
6.25 Appendix C - Circuit detection in directed graphs 84
Case 7. Temporal databases - Part 1 1
7.1 Introduction 2
7.2 Representation of time 2
7.3 The concept of current state 4
7.4 History of an entity 5
7.4.1 Temporal primary, unique and foreign keys 8
7.4.2 Normalized history 9
7.5 Transaction and valid times 10
7.6 Managing transaction time historical data 11
7.6.1 Current state of entities 14
7.6.2 Creating an entity 14
7.6.3 Evolution of an entity 17
7.6.4 Deleting an entity 19
7.7 Managing valid time historical data 21
7.7.1 Current state of entities 21
7.7.2 Creating of an entity 22
7.7.3 Evolution of an entity 22
7.7.4 Deleting an entity 25
7.8 From entity-based to tuple-based history 27
7.9 Alternative temporal database models 27
7.9.1 Table horizontal splitting 27
7.9.2 Table vertical splitting 28
7.9.3 Attribute-based history 28
7.9.4 Event-based history 32
7.9.5 Document-oriented history 33
7.10 The scripts 35
7.11 Some references 35
Case 8. Temporal databases - Part 2 1
8.1 Introduction 2
8.2 Temporal relations 2

8.3 The example temporal databases 4
8.4 Non temporal queries 5
8.5 Temporal queries with temporal result 6
8.6 Temporal queries with non temporal result 10
8.7 Complex temporal queries 12
8.8 Temporal projection 12
8.8.1 Entity-based temporal projection 13
8.8.2 Temporal projection with null values 16
8.8.3 Generalized temporal projection 17
8.9 Temporal join 23
8.10 Temporal outer join 27
8.11 Temporal aggregation 34
8.11.1 A first simple problem: counting 35
8.11.2 Value-based statistics: avg, max and min 40
8.11.3 Value-based statistics: sum 42
8.11.4 Multidimensional statistics 44
8.11.5 Temporal data with non-partitioned intervals 48
8.11.6 Efficient computing of VALUED_STABLE_INTERVAL 53
8.11.7 Temporal data with fixed and unit intervals 54
8.12 Temporal library LTemp 56
8.13 Temporal data format conversion 63
8.13.1 Entity-based to attribute-based history conversion 64
8.13.2 Entity-based to event-based history conversion 64
8.13.3 Entity-based to document-oriented history conversion 67
8.14 Performance analysis and optimization 70
8.14.1 Temporal projection 71
8.14.2 Temporal inner join 72
8.14.3 Temporal outer join 73
8.14.4 Aggregation 74
8.15 Temporal data management in current DBMS 79
8.15.1 Application-time temporal table 80
8.15.2 System-versioned temporal table 82
8.15.3 Future directions 83
8.16 The scripts 83
8.17 References 84
Case 9. Kings of France - Part 1 1
9.1 Introduction 1
9.2 Data structures 3

9.3 Warming up: some easy queries 5
9.4 Rebuilding royal families 7
9.5 King succession 11
9.6 Seeking missing kings 11
9.7 Royal conflicts 13
Case 10. Kings of France - Part 2 1
10.1 The ancestor/descendant relationships 1
10.2 Loop-based computing of the transitive closure 6
10.3 Counting descendants 6
10.4 Showing the descendants of a member 8
10.5 Graphical representation of king genealogy 10
10.6 Recovering the source data from their closure 11
10.7 Extracting the hierarchy of kings 13
10.8 The scripts 17
Case 11. Complement on Kings of France 1
11.1 Graphical representation of king genealogy - version 1 1
11.2 Graphical representation of king genealogy - version 2 5
Case 12. The book of which you are the hero 1
12.1 Interactive adventure books and computer games 2
12.2 Choosing sample game books 2
12.3 Designing a text-based interactive adventure game 5
12.4 Story telling 8
12.5 Structure of a game 10
12.5.1 Episodes, branches, paths and runs 10
12.5.2 Starting and ending episodes 11
12.5.3 Circuits 11
12.6 Analyzing a game 12
12.6.1 Searching for game abnormalities 12
12.6.2 Does the game include circuits? 13
12.7 Extracting circuits 16
12.8 Counting runs 20
12.9 Are all episodes reachable? 24
12.10 Do all episodes contribute to a solution? 26
12.11 Merging episodes 26
12.12 Toward a better game engine 27
12.13 The scripts 28
Case 13. Directory management 1

13.1 Of files and directories 2
13.2 Examining directories 2
13.3 Modeling the problem 3
13.4 Data structures 5
13.5 Loading a directory instance 8
13.5.1 Creating a directory 8
13.5.2 Creating the current instance 8
13.5.3 Creating the nodes of the current instance 9
13.5.4 Computing the path of a node 10
13.6 Applications 12
13.7 Application 1: general statistics 12
13.8 Selecting a directory instance 14
13.9 Application 2: Structure of a directory instance 15
13.10 Application 3: Examining the contents of an instance 16
13.11 Application 4: Duplicate nodes in a directory instance 17
13.11.1 List of duplicate nodes in a directory instance 18
13.11.2 Description of duplicate nodes in a directory instance 20
13.12 Application 5: Comparing two directory instances 21
13.13 Application 6: Searching for clone files 23
13.13.1 The problem 24
13.13.2 Saved by (computer) science 24
13.13.3 Hashing files 25
13.13.4 Identifying clone candidates in a directory instance 26
13.13.5 Identifying clone candidates among two directory instances 27
13.14 A touch of optimization 29
13.15 Building the DIRECTORY application 31
13.16 Suggestions 32
Case 14. Agent-based modeling and simulation 1
Case 15. Part 1 1
15.1 Introduction 4
15.2 Architecture of a simulation 8
15.2.1 The controller 8
15.2.2 The world 9
15.2.3 SQL implementation 10
15.3 The controller 11
15.3.1 Principles 11
15.3.2 SQL code 11
15.3.3 Firering order 13

15.4 Agent behavior 14
15.4.1 SQL modeling 14
15.5 Graphical representation of the world 15
15.6 Five (not so) easy pieces 15
15.7 Agents are moving: single-agent worlds 17
15.7.1 Trajectory styles 17
15.7.2 SQL modeling 19
15.7.3 SQL code 23
15.7.4 Experimentation: The Lawnmower 29
15.8 A first multi-agent world: Aggregation 33
15.8.1 SQL modeling 33
15.8.2 SQL code 34
15.8.3 Experimentation: the Aggregation simulator 35
15.9 Attractors and repellers 38
15.10 Agents and attractors 38
15.10.1 SQL modeling of attractors 38
15.10.2 SQL code for attractors 39
15.10.3 Experimentation: the Attractors-Repellers simulator 44
15.11 Agents and repellers 45
15.11.1 SQL modeling of repellers 45
15.11.2 SQL code for repellers 46
15.11.3 Experimentation: the Attractors-Repellers simulator 48
15.12 Agent life cycle 50
15.12.1 Towards autonomous agents 50
15.12.2 SQL modeling 51
15.12.3 SQL code 53
15.12.4 Experimentation: the LifeCycle simulator 57
15.13 Agent life cycle with birth control 61
15.13.1 The LifeCycle model is unstable 61
15.13.2 Adding birth control to the LifeCycle model 62
15.13.3 Experimentation: the Agent LifeCycle with birth control simulator 64
15.13.4 Refining the birth control model 65
15.13.5 Toward learning agents 67
15.14 Logging elementary operations and debugging 68
15.15 SQL coding in SQLfast 71
15.16 Predator-Prey systems 75
15.16.1 The Fox-rabbit system 75
15.16.2 An analytical solving method: the Lotka-Volterra equations 75

15.16.3 Modeling the rabbit society 75
15.16.4 Modeling the fox society 75
15.16.5 Modeling the fox-rabbit interactions 75
15.17 Implementation of the Fox-rabbit system 75
15.17.1 Principles of the simulation 75
15.17.2 Creating the world 75
15.17.3 Populating the world 75
15.17.4 Starting a simulation 75
15.17.5 Collecting statistics 75
15.18 Other applications 75
15.18.1 Pandemy 75
15.19 References 75
15.20 Introduction 75
15.21 Études de cas 76
15.22 Enter the predators! 76
15.23 References 80

Case 16. Conway's Game of Life 1
16.1 Introduction 2
16.2 World and automata representation 3
16.3 Computing the next state 5
16.4 Graphical display of automaton evolution 5
16.5 **Generation of SQLdraw scripts**. 7
16.6 **Packaging the LIFE application 7**
16.7 First performance analysis 9
16.8 Optimization 1: indexing tables 10
16.9 Optimization 2: windowing table access 12
16.10 Optimization 3: building a cheap world 14
16.11 Optimization 4: let SQL generate vector graphics 16
16.12 Optimization 5: Rewriting the application in Python 17
16.13 Some representative Conway's automata 18
16.14 Family life 19
16.15 Lessons learned 20
16.16 The scripts 22

Case 17. From data bulk loading
to database book writing 1
17.1 Introduction 2
17.2 The database loading problem 2
17.3 Representing the table graph of a database 5

17.4 Topological sorting of the table graph 6
17.4.1 Topological sorting abstract graphs 7
17.5 Coping with non acyclic graphs 8
17.5.1 Non acyclic graphs 8
17.5.2 Checking the acyclicity of a graph 9
17.5.3 The cyclic kernel of a graph 10
17.6 Cyclic vs hard cyclic kernel of a table graph 11
17.7 Contracting the circuits of the hard kernel 13
17.8 Finding the independent circuits of a graph 13
17.8.1 The independent circuits of a graph 13
17.8.2 What is a circuit after all? 14
17.9 Finding the super circuits of a graph 16
17.9.1 Computing the closure of a table graph 16
17.9.2 Finding the reachable set of a node 17
17.9.3 How to denote the super circuits of a graph? 18
17.9.4 Identifying the super circuits of a graph 19
17.9.5 Structure of the solution 21
17.9.6 Revisiting the optional links 22
17.9.7 Intermediate summary 22
17.10 Building the final acyclic graph 23
17.11 Optimizing the final acyclic graph 24
17.12 Loading data: a complex abstract example 26
17.13 The complete data loading generation procedure 30
17.14 Loading the data of a super circuit 31
17.15 Loading data: a real example 33
17.16 Brute force data loading technique 34
17.17 Performance 34
17.18 Complement: extracting the table graph of a database 35
17.19 Topological sorting: application to book development 35
17.20 Other applications 40
17.21 The scripts 41
Case 18. Classifying objects 1
18.1 Introduction and motivation 4
18.2 Principles of FCA - Finding the concepts 11
18.3 Principles of FCA - Building the concept hierarchy 15
18.4 Looking for a systematic procedure 16
18.4.1 Finding the concepts 16
18.4.2 Building the concept hierarchy 19

18.5 The Chein algorithm 20
18.5.1 Analysis of the algorithm 20
18.5.2 Procedural expression of the Chein algorithm 22
18.6 An SQL implementation of the Chein algorithm 23
18.6.1 Operations on sets in SQL 23
18.6.2 SQL implementation of the Chein algorithm 27
18.7 A Python implementation of the Chein algorithm 31
18.8 Building the concept hierarchy 33
18.8.1 An SQL implementation 33
18.8.2 A Python implementation 34
18.9 An alternative FCA implementation: ISA recovery 35
18.9.1 The basic algorithm 35
18.9.2 An SQL translation of the ISA recovery algorithm 36
18.9.3 The case of empty concepts 37
18.9.4 Python translation of the ISA recovery algorithm 44
18.10 Experimentation 45
18.10.1 The scripts 45
18.10.2 Sample formal contexts 48
18.10.3 The context generator 51
18.11 Evaluation of the FCA implementations 51
18.11.1 Simplicity of algorithms 52
18.11.2 Preservation of the source concepts 53
18.11.3 Nature of the attributes of the concepts 53
18.11.4 Creation of empty intermediate concepts 53
18.11.5 Memory space requirements 54
18.11.6 Runtimes 57
18.11.7 Other techniques 58
18.12 Improving the performance of the algorithms 59
18.12.1 Chein algorithm (Python implementation) 59
18.12.2 Chein algorithm (SQL implementation) 61
18.12.3 Implementation of the ISA recovery algorithm 63
18.12.4 Building of the concept hierarchy (SQL implementation) 64
18.12.5 Building of the concept hierarchy (Python implementation) 64
18.13 On the representation of concept hierarchies 65
18.14 Querying the concept hierarchy 70
18.15 Applications 72
18.16 A short bibliography 73
Case 19. Path finders, rovers and Ariadne’s thread 1

19.1 Shortest paths between two cities 1
19.2 Dijkstra's Algorithm 2
19.3 Representation of the graph 6
19.4 Computing the shortest paths from a starting city 8
19.5 Building a general purpose shortest path engine 11
19.6 Ariadne's thread: solving mazes 14
19.7 Exploring a planet 17
19.8 Performance and optimization 21
19.9 The scripts 22
Case 20. Blockchains 1
20.1 Introduction 1
20.2 Elements of cryptography 2
20.3 The blockckain paradigm 6
20.4 Application: cryptocurrencies 9
20.5 The BLOCKCHAIN toolbox 12
20.6 The database 13
20.6.1 The transactions (table BTRANSACTION) 13
20.6.2 The blocks (table BLOCK) 14
20.6.3 The public key directory (table DIRECTORY) 15
20.7 Experimental data 16
20.8 Account registration 17
20.9 Amount deposit 18
20.10 Money transfer 19
20.11 Validating a transaction 21
20.12 Creating and mining a block 21
20.13 TExamining the blockchain data 24
20.13.1 Examining an account 26
20.13.2 Examining the transactions 26
20.13.3 Examining the blockchain 27
20.14 Wrapping it all: the BLOCKCHAIN toolbox 28

Case study 1

Four hours to save the library

Objective: This case study describes the emergency writing of a small application that was to implement the core functions of the management of a small library. The challenge was to replace the current software, lost in a recent crash of the server. All that was left after the accident was the last backup of the database, unfortunately in an unknown format.

Keywords: rapid application development, application prototyping, application architecture, GUI

1.1 An emergency situation

This one is a true story.

A friend of mine manages a public library that lends comic books. That library is very popular, counting about 3,000 readers and 20,000 books. Its activity is estimated to about 4,000 transactions (lending and return) per week.

Some time ago, he called me, desperate, to tell that his server was destroyed during the storm of the last weekend. This server hosted the library management software he has been using twice a week. Everything had disappeared: the software sources and binaries, the documentation, the database and even the name of the language in which the application was written. The software was developed decades ago by *the nephew of a friend of the brother in law of the fitness instructor of his wife* (just synthesizing) and is therefore what we can call both *proprietary* and

orphan. It seemed that the author of the application is now breeding sheeps in Corrèze and that he has long abandoned his software development activity.

All that was left was the last backup of the database on a USB key.

1.2 First mission: recovering the data

The first objective was to extract as much usable data as possible from the backup database.

That database is a collection of 18 files with extensions *.lis, *.fic, *.ndx, *.mmo, *.id and *.tab. It turns out that files *.fic were *WinDev* data files and that all the other files could be ignored. Unfortunately, no (free!) converter utility for Windev data files, nor a description of the structure of these files could be found on the web.

A visual inspection of *.fic files through notepad++ showed that some data values are Latin-1 character strings but that others are coded as pure bit strings. In particular, foreign keys are not expressed as character strings as is usual, but as 24-bit pointers. We wrote a small Python program to extract these data and to convert them into SQL `insert` statements. Three files, comprising the core of the data, were processed in this way, namely the *reader* file, the *book* file and the *borrowing* file, and were transferred into three similar relational tables (see Script 1.1).[1] The other files were ignored. The resulting database was about 100 MB large.

By analyzing the data stored in these tables, we determined the data types and the uniqueness constraints, from which we derived the primary keys. In table BORROW, we detected two foreign keys, respectively to table BOOK and table READER. However, we decided not to declare them to allow for the recording of *dangling* (i.e., corrupt) borrowings for which either the book or the reader (or both) were missing.

As is usual in most DBMS, an index is automatically associated with each primary key. In addition, we create an index for each foreign key, be it declared or implicit. Since foreign key {BookID} is a prefix of the primary index of BORROW,[2] the latter also supports this foreign key. We just had to create an index on {ReaderID}, the second foreign key.

1.3 The basic functions

Since the accident, the trace of the day-to-day operations has been painfully written down in a paper notebook. The most urgent tasks were, as expected, book borrowing

1. In a further chapter, we will develop a more powerful library management application. This one can be considered a prototype version of the latter. Hence the name LIBRARY-**proto**.db of the database.
2. and since this index is implemented by a B-tree ...

and book return. However, registering new readers and new books and consulting the data was also considered critical.

We decided to write a small, no-frills, interactive program that supports these activities and that made it possible to convert the handwritten backlog into electronic data. Three core tasks were identified: *registering new readers* and *new books*, *borrowing and returning books*, *querying reader and book data*.

```
create table BOOK(
   BookID     varchar(32) not null primary key,
   Title      varchar(96) not null,
   Collection varchar(32),
   Authors    varchar(64),
   Publisher  varchar(32) );

create table READER(
   ReaderID   varchar(32) not null primary key,
   RegDate    char(24)    not null,
   Name       varchar(48) not null,
   Address    varchar(64) );

create table BORROW(
   BookID     varchar(32) not null,
   ReaderID   varchar(32) not null,
   BorrowDate date not null,
   ReturnDate date,
   primary key (BookID,BorrowDate) );

create index ndx_borrow_reader on BORROW(ReaderID);
```

Script 1.1 - The LIBRARY-proto.db database [Script LIBRARY-Create-DB-proto.sql]

1.4 The main window

The main window is the control panel of the application. It lets the user select a function (Figure 1.1). Button **OK** starts the selected function while button **Cancel** closes the application. Setting the `commit` mode to *autocommit* (each data modification query is wrapped into its own transaction) allows us to ignore explicit `commitDB` statements.

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 1.2). 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.

There is no need to close the database explicitly since it is automatically closed when the main script finishes.

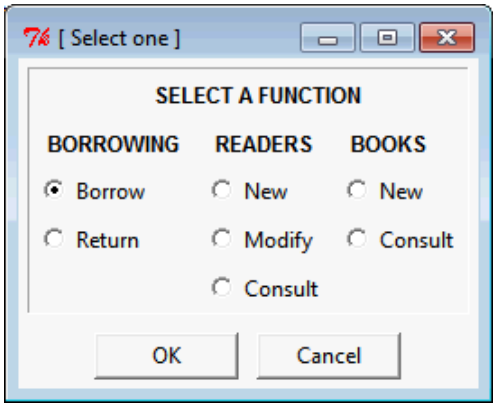


Figure 1.1 - The main window lets the user select a function

```
parameter commitmode  = autocommit;
openDB LIBRARY-proto.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 Library/_LIB-BORROWING-BORROW.sql;
  if ($oper$ = 2) execSQL Library/_LIB-BORROWING-RETURN.sql;
  if ($oper$ = 3) execSQL Library/_LIB-READER-NEW.sql;
  if ($oper$ = 4) execSQL Library/_LIB-READER-MODIFY.sql;
  if ($oper$ = 5) execSQL Library/_LIB-READER-CONSULT.sql;
  if ($oper$ = 6) execSQL Library/_LIB-BOOK-NEW.sql;
  if ($oper$ = 7) execSQL Library/_LIB-BOOK-CONSULT.sql;
endwhile;
```

Script 1.2 - Code of the control panel [Script LIBRARY-Main.sql]

1.5 Registering a new reader

The data entry box for new readers shown in Figure 1.2 was found quite appropriate.

The code is straightforward (Script 1.3):

- initializing the field variables (notably to the current date)
- acquiring field values from the user (`ask-u`)

– checking the validity of these values; looping until the values are valid
– inserting the values in the database (`insert`); field Address is converted into a *null* value if empty
– checking the result, committing (`commitDB`) or reporting errors (`showMessage`), if any.

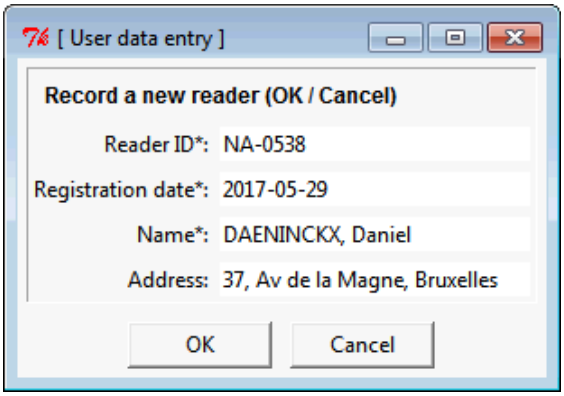


Figure 1.2 - Data entry box for a new reader

```
set rid,dat,nam,add = ,$date$,,;
while (True);
  ask-u rid,dat,nam,add = [/bRecord a new reader (OK/Cancel)]
        Reader ID*:|Registration date*:|Name*:|Address:;
  if ('$DIALOGbutton$' = 'Cancel') return;
  if (trim('§rid§') <> '' and '$dat$' <> ''
      and trim('§nam§') <> '') exit;
endwhile;
insert into READER(ReaderID,RegDate,Name,Address)
   values('§rid§','$dat$','§nam§',
          case when '§add§' = '' then null else 'add§' end);
if ('$SQLdiag$' <> 'OK')
   showMessage Recording error ($SQLdiag$);
```

Script 1.3 - Code of the procedure to register a new reader [Script _LIB-READER-NEW.sql]

1.6 Registering a new book

The procedure that registers a new book is quite similar to that of reader registering. The data entry box is shown in Figure 1.3. The value of BookID of a book is printed

as a barcode on a label stuck on the cover of the book. This way, a book can be identified by merely scanning its barcode.

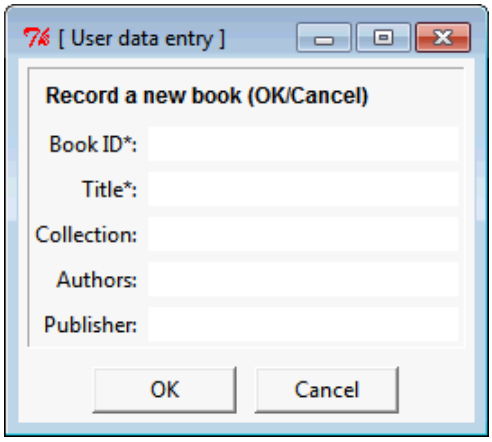


Figure 1.3 - Data entry box for a new book

1.7 Borrowing a book

The ID of the book and that of the reader are entered through the box shown in Figure 1.4. The book ID is collected through the codebar scanner while the reader is identified through a predefined list of reader names (actually ReaderID + Name).

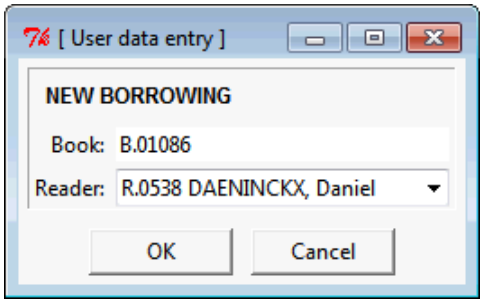


Figure 1.4 - Borrowing a book

The code of this operation is shown in Script 1.4. Statement **`ask`** returns a value of BookID and a value of ReaderID. These values can be considered valid due to the way they are collected: through barcode scanning and predefined value list. Therefore no validation code has been included.

However, we must ensure that the database records this book as being available, that is, there is no BORROW row for this book with a *null* ReturnDate value.

Searching the database for such a row must fail (`'$SQLdiag$' = 'NONE'`), in which case a new row describing the borrowing can be inserted and committed.

The value of BorrowDate is that of **`current_date`** while the return date (ReturnDate) will be left as *null* until the book is returned.

```
ask book,read = [/bNEW BORROWING] Book:
    |Reader:[!select ReaderID||' '||substr(Name,1,20),ReaderID
             from   READER order by Name];
if ('$DIALOGbutton$' = 'Cancel') return;

extract r = select ReaderID from BORROW
            where  BookID = '$book$' and ReturnDate is null;

if ('$SQLdiag$' = 'NONE');
   insert into BORROW(BookID,ReaderID,BorrowDate)
               values('$book$','$read$',current_date);
else;
   showMessage This book is currently borrowed by $r$;
endif;
```

Script 1.4 - Code of the procedure to borrow a book [script _LIB-BORROWING-BORROW.sql]

1.8 Returning a book

Returning a book translates into assigning the current date to column ReturnDate to the row describing the current borrowing of the book. This row then becomes a historical record for both the book and the reader.[3]

The box of Figure 1.5 comprises a single field, in which the value of BookID has to be entered through barcode scanning.

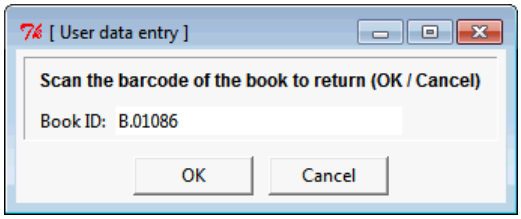


Figure 1.5 - Returning a book

The code of the operation is shown in Script 1.5. Once the book ID has been collected (**`ask`**), the borrowing date (BorrowDate) of the book is extracted

3. In some countries, maintaining historical data on readers may be illegal for privacy reason.

(**extract**). Since columns {BookID, BorrowDate} form the primary key of the table, the values of variables **book** and **dat** can be used to identify the row in which the value of ReturnDate is set to the current date (update).

```
ask book = [/bScan the barcode of the book (OK/Cancel)]
            Book ID:;
if ('$DIALOGbutton$' = 'Cancel') return;
extract dat = select BorrowDate from BORROW
              where  BookID = '§book§'
              and    ReturnDate is null;
if ('$SQLdiag$' = 'OK');
   update BORROW
   set    ReturnDate = current_date
   where  BookID = '§book§' and BorrowDate = '$dat$';
else;
   showMessage The book §book§ is not currently borrowed;
endif;
```

Script 1.5 - Code of the procedure to return a book [script _LIB-BORROWING-RETURN.sql]

1.9 Updating the data of a reader

Updating a reader is performed in two steps: first selecting a reader from a list of names, then presenting the data of this reader to let the user change some values (Script 1.6).

1.10 Querying reader and book data

We chose a procedure which is both simple and general. For both READER and BOOK tables, the user selects a *column* and an *operator*, then enters a *value*, considered case-insensitive (Figure 1.6).

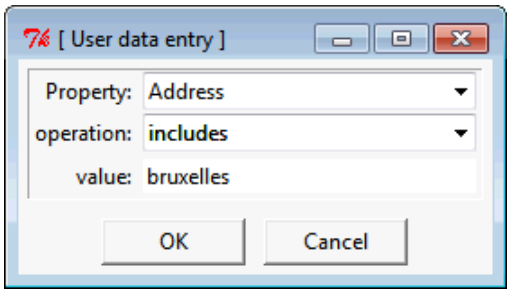


Figure 1.6 - Consulting the readers whose **address includes** the word '***Bruxelles***'

```
ask rid = [/bSelect a reader to modify]
           Reader ID:[!select ReaderID||' - '||Name, ReaderID
                      from   READER order by Name];
if ('$DIALOGbutton$' = 'Cancel') return;
extract dat,nam,add = select RegDate,Name,Address from READER
                      where  ReaderID = '§rid§';
while (True);
  ask-u dat,nam,add = [/bModify the reader (OK/Cancel)]
                       Reader ID*:|Registration date*:|Name*:;
  if ('$DIALOGbutton$' = 'Cancel') return;
  if (trim('$dat$') <> '' and trim('§nam§') <> '') exit;
endwhile;

update READER
set RegDate = '§rid§', Name = '§nam§',
    Address = case when '§add§' = '' then null else '§add§' end
where ReaderID = '§rid§';

if ('$SQLdiag$' <> 'OK')
   showMessage Recording error ($SQLdiag$);
```

Script 1.6 - Code of the procedure to modify the data of a reader [script _LIB-READER-MODIFY.sql]

The list of operators to select from comprises the usual comparison operators (<, >, <=, >=, <>) + `includes`, a reduced version of SQL predicate `like`.

A first version of the function produced the result of Figure 1.7.

ReaderID	RegDate	Name	Address
R.0538	2016-11-13	SMITH Bernard	Place Bockstael, Bruxelles
R.0572	2016-11-21	JONES F. J.	108, Sq de Meeus, Bruxelles
R.0668	2016-12-21	TRAVIS K. K.	9, Rue H. Stockel, Bruxelles

Figure 1.7 - The readers living in Brussels

It was considered insufficient since the main consultation needs were related to the past and current borrowings of each reader or of each book. Something like Figure 1.8, reporting on each reader and on their borrowings (same for books) would be better.

The code of both procedures is exactly the same, except for the *name of the table* and the list of *column names*. Therefore, we split the code into three procedures: _LIB-READER-CONSULT, _LIB-BOOK-CONSULT and _LIB-CONSULT. The latter does all the work while the first two just prepare the specific arguments (script 1.7 for consulting readers).

ReaderID	RegDate	Name	Address	Book	BorrowDate	ReturnDate
R.0538	2016...	SMITH..	...			
				B.01196	2017-05-08	2017-05-14
				B.01377	2017-05-10	2017-05-14
R.0572	2016...	JONES..	...			
				B.01086	2017-05-20	--
R.0668	2016...	TRAVIS..	...			
				B.01196	2017-05-20	--
				B.01208	2017-05-26	--
				B.01196	2017-05-01	2017-05-08

Figure 1.8 - The readers living in Brussels and the history of their borrowings

```
set table  = READER;
set fields = ReaderID,RegDate,Name,Address;
execSQL LIBRARY/_LIB-CONSULT.sql;
```

Script 1.7 - Code of the procedure to consult readers [script _LIB-READER-CONSULT.sql]

Procedure _LIB-CONSULT is a bit more complex (Script 1.8). From the data collected by the dialogue box (**ask**), the procedure builds the selection condition (**set condition = ...**) that will be inserted in the SQL **select** query.

This query comprises two subqueries: the first one extracts the data of READER rows that satisfy the condition (the blue one in Figure 1.8) and the second one extracts the data of BORROW rows that depend on each READER row selected (the red ones in Figure 1.8).

READER (or BOOK) subquery

Let us examine the first subquery (in blue in Script 1.8). The **from** and **where** clauses are straightforward:

```
from  $table$
where $condition$
```

The **select** clause comprises two lists of column names. The first one is the list of column names of table READER (or BOOK) while the second one acts as a *padding* for the columns of BORROW, that must be empty. The number and names of the latter columns are known and are the same for tables READER and BOOK. The `select` list comprises an additional column (**Sort**) that will be explained later.

BORROW subquery

The second subquery (in red in Script 1.8) generates the depending rows of BORROW. Here, the *padding* columns are the first ones. They comprises an empty column for each column of READER (or of BOOK). Their `select` sublist is built in variable **empty** by statement **compute**, that works as follows:

- variable **fields** contains the list of column names, separated by commas
- function **itemLen**(S,sep) returns the number of items in list **S** with separator **sep**; so, **itemLen**('$fields$',',') computes the number of column names in list **fields**
- function **repeat**(S,n) returns a string formed by **n** instances of substring **S**
- for table READER, that comprises four columns, variable **empty** contains '','','','', that is, **4** instances (= number of columns) of substring '',
- the arguments of **compute** comply with the SQL syntax for character constants, according to which a constant must be quoted and each internal quote must be doubled; therefore, substring '', must be coded '''',.[4]

Now, the structure of the **compute** statement should be clearer!

The subquery is a join between table BORROW, from which data are extracted, and table READER (or BOOK), reduced to its selected rows ($condition$).

Sorting rows

The two subsets of rows are simply *union*ed in the **from** clause. The order of the rows resulting from this union is undefined and does not produce the nice hierarchical ordering shown in Figure 1.8, in which each READER row is directly followed by its dependent BORROW rows (same for BOOK).

We must force the rows to appear in this definite order. In order to do so, we build an artificial column, called **Sort**, comprising column ReaderID (more generally $table$ID) plus suffix -0 for READER (or BOOK) and -1 for table BORROW. In this way, all the rows with the same value of ReaderID are grouped, and, in each such group, the row of READER appears in the first position.

The particular structure of the **from** clause, as the union of two subqueries, is necessary to hide column **Sort**, which is meaningless for users.

Finally, we decided to send the resulting data in a text window. Those data are first stored in variable **result**, which is then displayed through statement **showText**.

1.11 Conclusion

The librarian claimed to be quite happy with this small application. Despite its unpolished interface and simplistic logic, it allowed him to keep his activity alive and to encode the backlog of recent transactions written on paper.

Two weeks later, on the basis of his comments, some improvement was carried out to the application, that, from then, has been used for about one year, until the data were migrated to a more professional software.

4. For any complaint, contact www.iso.org

```
ask key,op,val = Property:[($fields$)]
                 |operation:[(=,<,>,<=,>=,<>,includes)]|value:;
if ('$DIALOGbutton$' = 'Cancel') return;
if ('$op$' in ('=','<','>','<=','>=','<>'))
   set condition = lower($key$) $op$ lower('$val$');
if ('$op$' = 'includes')
   set condition = lower($key$) like '%'||lower('$val$')||'%';
compute empty = repeat(''''',',itemLen('$fields$',','));
outputOpen result.var;
select $fields$,RID as Reader,BID as Book,
                BorrowDate,ReturnDate
from
(select $fields$,'' as RID,'' as BID,
     '' as BorrowDate,'' as ReturnDate,$table$ID||'-0' as Sort
 from   $table$ where $condition$
   union
 select $empty$ B.ReaderID as RID,B.BookID as BID,
                BorrowDate,ReturnDate,B.$table$ID||'-1' as Sort
 from   BORROW B, $table$
 where  B.$table$ID = $table$.$table$ID and $condition$
)
order by Sort,ReturnDate;
outputAppend window;
showText result = [/w0/x100/y20];
```

Script 1.8 - Code of the procedure to consult a reader or a book [script _LIB-CONSULT.sql]

Though data recovery took some time (about 12 hours), due to the lack of documentation, writing the complete application just cost:

– less than 70 instructions

– 4 hours.[5]

Of course, the context in which application LIBRARY has been developed is just a minor issue. Actually, it was a nice opportunity to experiment with the concept of *rapid application development* (*aka* RAD) and *application prototyping*.

Case study *The human factor* tackles a similar application (library management) in which, due to the wider variety of users, the way they interact with the program leads to new problems.

5. Probably twice as much as what a professional application programmer would have spent!

A last remark

The scripts developed in this document are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

Case study **3**

Interactive SQL interpreters

Objective: An interactive SQL interpreter is this kind of graphical interface through which one enters an SQL query and that displays the result of its execution in a text window. They are used, for instance, to learn SQL or to test and tune SQL queries that are to be integrated in application programs. There are many of them available for free on the internet or included in relational DBMS. In this study, we will build, step by step, our own SQL interpreter by implementing the functions and features we want to use, notably (but not exclusively) to train students in writing SQL queries.

Starting from a very tiny interpreter (just 7-character long!) we will build a series of more comprehensive and versatile versions, up to the last one, that will be able, not only to execute the queries submitted by the user, but also to evaluate their correctness.

All these versions are available as two ready to run applications.

Keywords: SQL interpreter, GUI, learning SQL, query evaluation, multiset, set operator, base64

3.1 Introduction

The graphical interface of the SQLfast environment is particularly fit for experimenting with the SQL language: we type a query in the main window, then we click on button **Run** and finally we examine the result in the output window. That is fairly convenient to learn the concepts of databases and SQL.

This interface is quite general but may be felt too complex or, on the contrary, too simplistic, as a support to SQL training. So, why not develop our own graphical interface that meets our specific needs? For example we would like, among others:

- to integrate the query and output text fields into a single window
- to discard all these useless buttons and menu items that clutter the interface
- to add new functions specifically devoted to SQL learning
- to customize the layout of the interface
- to display more user-friendly error messages
- to integrate tutorial and exercise modules
- and (let us have a dream) to automatically evaluate student's answers to the exercises.

The goal of this study is to explore some techniques to implement these wishes into what we will call *Interactive SQL interpreters*.

3.2 The smallest interactive interpreter in the world!

For this first try, let us work with the *Basic* interface (check the top right label of the main window). We open database ORDERS.db, we type the statements of Script 3.1 then we click on button **Run**. The first statement invites the user to type an SQL query and stores it in variable **Query**. The contents of this variable forms the second statement, which is then executed. The result appears in the standard output window (Figure 3.1).

```
ask Query;
$Query$;
```

Script 3.1 - An elementary interactive SQL interpreter

Note (useless, just for fun)

This script can even be made more compact (as low as **7 characters**, space and semi-colons excluded) by reducing the name of the variable to a single letter:

```
ask Q;
$Q$;
```

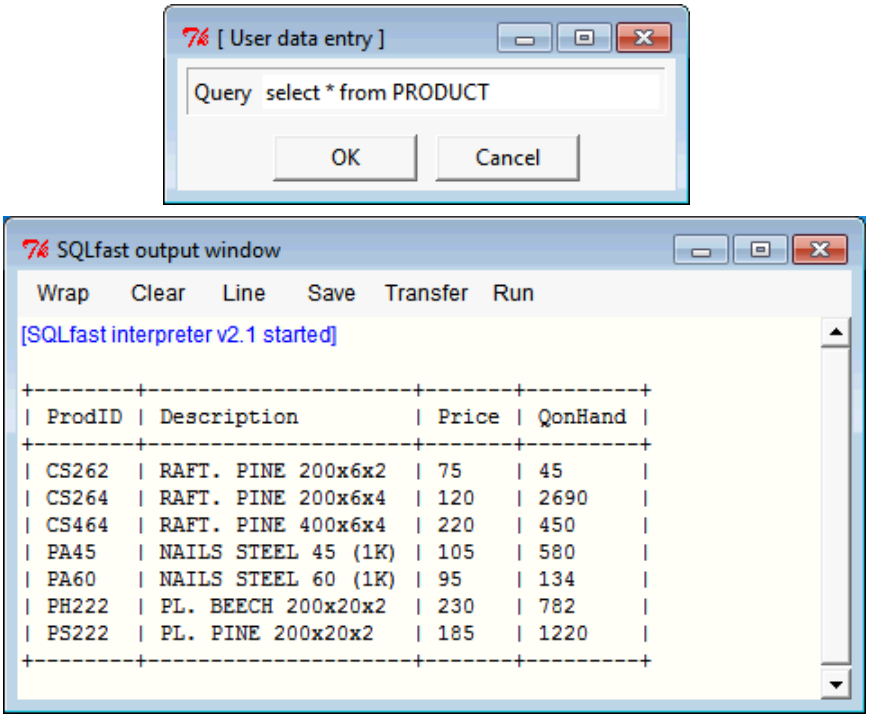


Figure 3.1 - The elementary query entry box and the output window

Extending this script by making it *iterative* is quite easy (Script 3.2). The query entry box opens repeatedly, until the user has no query to execute any more and clicks on button **Cancel**.

```
while (True);
  ask Query;
    if ('$DIALOGbutton$' = 'Cancel') exit;
  $Query$;
endwhile;
```

Script 3.2 - An elementary interactive SQL interpreter

Now we will learn to develop more sophisticated interpreters.

Though quite operational, this first trial is not particularly elegant nor easy to use for any serious experimentation.

First, queries are limited to one short line. Fine for some elementary queries from beginners, but submitting more complex queries will be awkward. Let us replace the data entry box by a **text entry box** (Figure 3.2). We just replace statement **`ask`** with **`askText`** (Script 3.3).

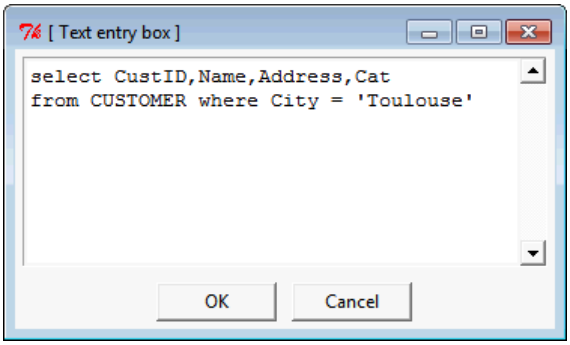


Figure 3.2 - Getting the query to submit through a text entry box

```
while (True):
  askText Query;
    if ('$DIALOGbutton$' = 'Cancel') exit;
  $Query$;
endwhile;
```

Script 3.3 - Entering larger queries in a text box

Now, let us integrate the query and output windows. To do so, we abandon the standard output window and we add a new text field in the query entry box. The result set of the query is sent in variable **result** by redirecting the output channel to this variable:

```
outputAppend result.var;
```

The contents of **result** is then shown by statement **showText**. Both text fields are assembled vertically by an **askCombo** statement:

```
askCombo [askText Query]|[showText result];
```

The code of this version is shown in Script 3.4.

```
set result = ;
outputAppend result.var;
while (True);
  askCombo [askText Query]|[showText result];
    if ('$DIALOGbutton$' = 'Cancel') exit;
  $Query$;
endwhile;
```

Script 3.4 - Integrating query and output text boxes

3.3 A simple but realistic SQL interpreter

It is time to leave these first experiments for a more professional approach.

First, we adjust the **size of the text fields**. Eight lines for the query field and twenty four lines for the result field seem fairly comfortable. In addition, **word wrapping** in the query field is what we expect (long lines are always visible) while it would spoil the presentation of wide result sets in the result field. Hence the different parameter settings for these fields:

```
[askText Query = [/w1/y8]]|[showText result = [/w0/y24]]
```

Now, let us address the question of the **contents** displayed in the text fields. We decide that the query that has been executed **remains** in the **query field**, so that the user can modify it for further execution, for example to fix errors or to run variants. We replace statement **askText** by **askText-u**, that displays the contents of variable **Query** for modification (**-u** for *update*).

As to the contents of the **result field**, we find it useful to **copy the query** before execution (**write-ab** $Query$). This will produce a nice report in which each query is followed by the result set of its execution.

Finally, the layout in which users enter their queries in the query field is not always elegant. So, we rewrite it both in the query and the result fields. For this, we use the *pretty print* function of the **LStr** library:

```
function Query = LStr:PrettyPrint {$Query$};
```

If this query is typed in a single line in the query field:

```
select CustID,Name,Address,Cat from CUSTOMER where City =
'Toulouse'
```

then it is rewritten in both text fields as:

```
select CustID,Name,Address,Cat
from   CUSTOMER
where  City = 'Toulouse'
```

These features are translated in Script 3.5. Figure 3.3 shows the state of the SQL interpreter when the query mentioned above has been entered and executed, then executed again for new value `'Paris'`.

3.4 Error management

If an erroneous SQL query is typed in (e.g., `select * from KUSTOMER`), as is natural from SQL learners, an error message will pop up, like that of Figure 3.4.

```
set Query,result = , ;
outputAppend result.var;
while (True);
  askCombo [askText-u Query = [/w1/y8]]
           |[showText result = [/w0/y24]];
    if ('$DIALOGbutton$' = 'Cancel') exit;
  function Query = LStr:PrettyPrint {$Query$};
  write-ab $Query$;
  $Query$;
endwhile;
```

Script 3.5 - At last, a realistic SQL interpreter

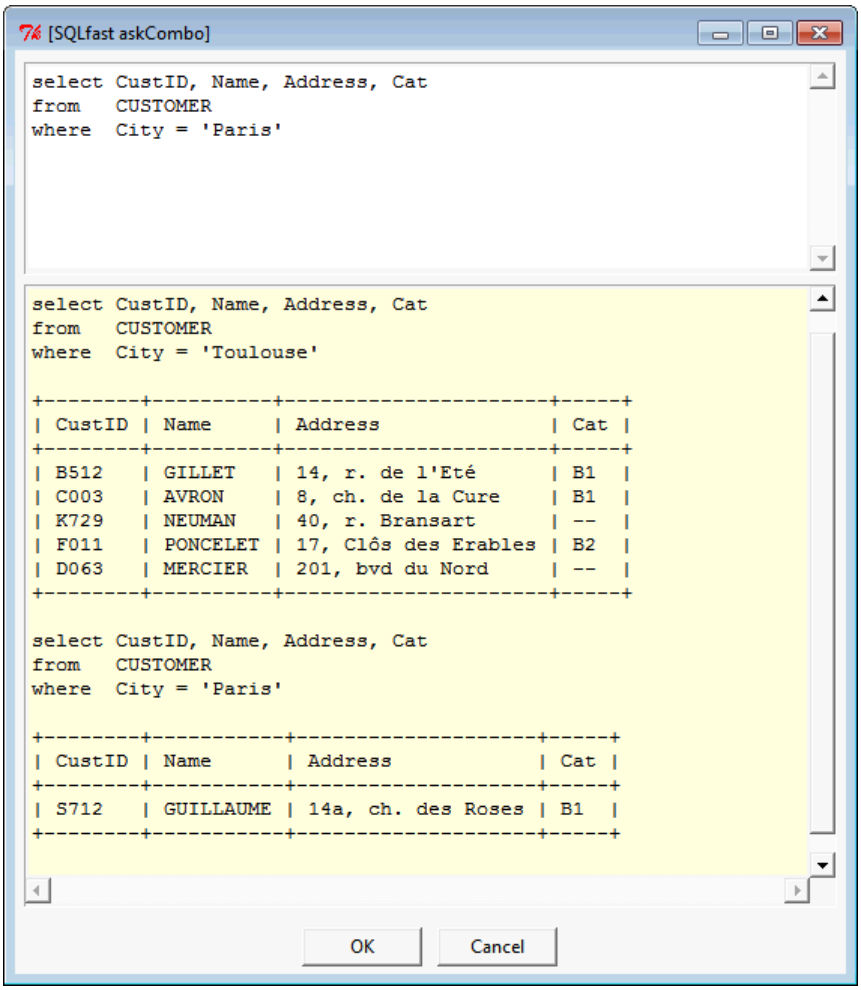


Figure 3.3 - State of the SQL interpreter of Script 3.5

This standard message is quite detailed, but it refers to the script we have executed and not to the operation the student wanted to carry out. Message "no such table: KUSTOMER" is fine, but the rest of the text will just confuse the student.

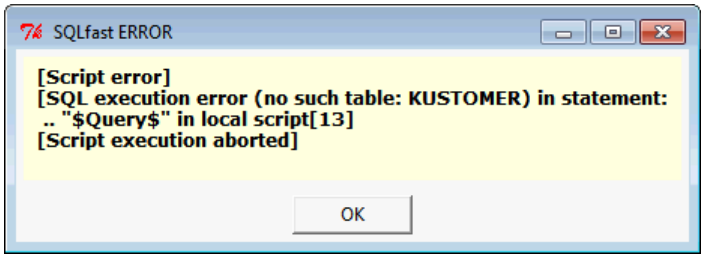


Figure 3.4 - Not exactly the kind of message a novice user would like to read!

First, we disable the standard error management, letting the script continue its execution in case of error:

```
onError continue;
```

Then, we can examine the error messages returned by the SQLfast engine through three system variables:

- **Error** is a simple indicator telling whether the last statement has been successfully executed (Error = 0) or not (Error = 1)
- **SQLdiag** provides a concise diagnostic on the execution of the last query: '**OK**' if everything went fine, '**NONE**' if no row was found, '**ID**' for a uniqueness error, etc.
- **EXTENDEDdiag** gives more detail on the error. Generally, it includes the nature of the query and the error message provided by the SQL engine.

We agree on two successful messages, namely '**OK**' and '**NONE**', all the other ones being considered as unsuccessful execution. Let us replace the simple execution statement

```
$Query$;
```

by this sequence

```
onError continue;
  $Query$;
  if ($Error$) write $SQLdiag$; $EXTENDEDdiag$;
onError stop;
```

Now the message is much sober:

```
SCHEMA; select: no such table: KUSTOMER
```

The contents of these two variables provide us with the necessary information to build a more *student-friendly* error message, as shown in Figure 3.5.

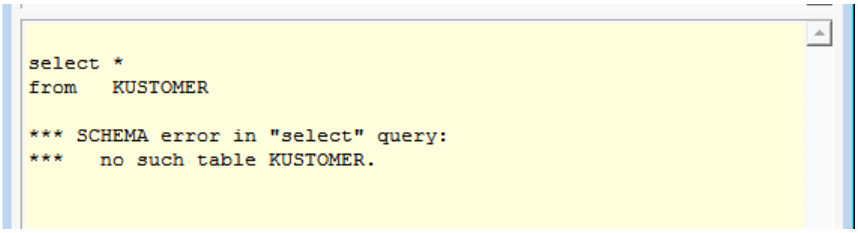


Figure 3.5 - A simpler and more informative error message

To cope with the different errors that may occur in the execution of all SQL queries, it is best to create a specific procedure that identifies the error and that creates the appropriate error message. Now, the modification of the main script looks like this:

```
onError continue;
  $Query$;
  if ($Error$) execSQL SQL-ErrorProcessing.sql;
onError stop;
```

The code of script **ErrorProcessing.sql** could be that of Script 3.6.

```
compute verb = lower(item('§Query§',1,' '));
if ('§verb§' not in ('select','insert','update','delete'))
   set verb = unknown;
goto $verb$;
label select;
   compute mess = '*** $SQLdiag$ error in "$verb$"'
                ||'" query:@n***  '
                ||item('$EXTENDEDdiag$',2,':')
                ||item('$EXTENDEDdiag$',3,':')||'.';
   goto Display;
label delete;
   ...
   goto Display;
...
label unknown
   set mess = *** Unknown operation "$verb$". Should be:
            @n***  "select", "insert", "update" or "delete".;
label Display;
   write $mess$;
```

Script 3.6 - The procedure processing SQL errors

3.5 Adding information fields

First time users could be puzzled by the *complexity* of the dialogue box! Let us add short informative messages to help them identify the different components of the dialogue box (Script 3.7).

```
...
askCombo [/bInteractive SQL interpreter.
            Enter an SQL query then click on OK.]
          [askText-u Query = [/w1/y8/bMy SQL query]]
         |[showText result = [/w0/y24/bThe result of my query]];
...
```

Script 3.7 - Adding information messages

3.6 Adding functions

To make the life of students easier and more comfortable, we will add some functions to the interface. First, copying the query in the result field will be made an option. Same idea for appending the result of the next query to the text already written in the result field. So, we create two buttons:

- **Show query**: if checked, the next queries will be copied in the result field before each of their result set.
- **Preserve history**: if checked, the query just executed and its result set (or diagnostic) will be *appended* to the contents of the result field (hence the term *history*). Otherwise, this material will *replace* the current contents.

It would be nice too to add a *short tutorial* to help first time students to start using the interpreter. We add a button (**Show help**) that, when checked, opens a help document the next time button OK is clicked on. If unchecked, this document does not appear.

A fourth function will allow students to ask for a *report* of their session, comprising the trace of their activities. This report is the contents of the result field and will be written in a text file on exit of the interpreter. When checked, button **Save history on exit** automatically activates function **Preserve history**.

Now, the dialogue box creation statement looks like Script 3.8. The complete script is shown in Script 3.9. It produces the interface of Figure 3.6.

Two comments on this script:

- Saving the history depends on the value of indicator **S** on exit. However, this value will be not be updated when the box is closed with button **Cancel**. There-

fore, button **Save history on exit** must be checked before some queries are submitted, which is fairly natural.[1]

– The save file is given a standard name like SQL-Save-2017-10-28_16-08-32.txt. This name is built from time registers **date** and **time**. Since the time format includes semi-colons (according the ISO format), the latter are converted into simple dashes.

```
askCombo [/bInteractive SQL ...]
 [selectMany-u S = Save history on exit]
|[askText-u Query = [/w1/y8/bMy SQL query]]
|[selectMany-u Q,P,H = Show query||Preserve history||Show help]
|[showText result = [/w0/y24/bThe result of my query]];
```

Script 3.8 - Four function check buttons are added

```
outputAppend result.var;
set Query,result,log = ,,;
set S,Q,P,H = 0,1,1,0;
while (True);
   askCombo ...;
      if ('$DIALOGbutton$' = 'Cancel') exit;
   if ($S$) set P = 1;
   function Query = LStr:PrettyPrint {$Query$};
   if (not $P$) set result = ;
   if ($Q$) write-ab $Query$;
   onError continue;
      $Query$;
      if ($Error$) execSQL _SQL-ErrorProcessing.sql;
   onError stop;
   if ($H$) displayHelp SQL-interpreter.tuto;
   if (not $H$) closeHelp;
endwhile;
if ($S$);
   compute tim = '$date$'||'_'
                 ||substr(replace('$time$',':','-'),1,8);
   outputOpen SQL-Save-$tim$.txt;
   write $result$;
endif;
```

Script 3.9 - Interactive SQL interpreter - Final version

1. In other words, checking this button just before leaving the interpreter is useless!

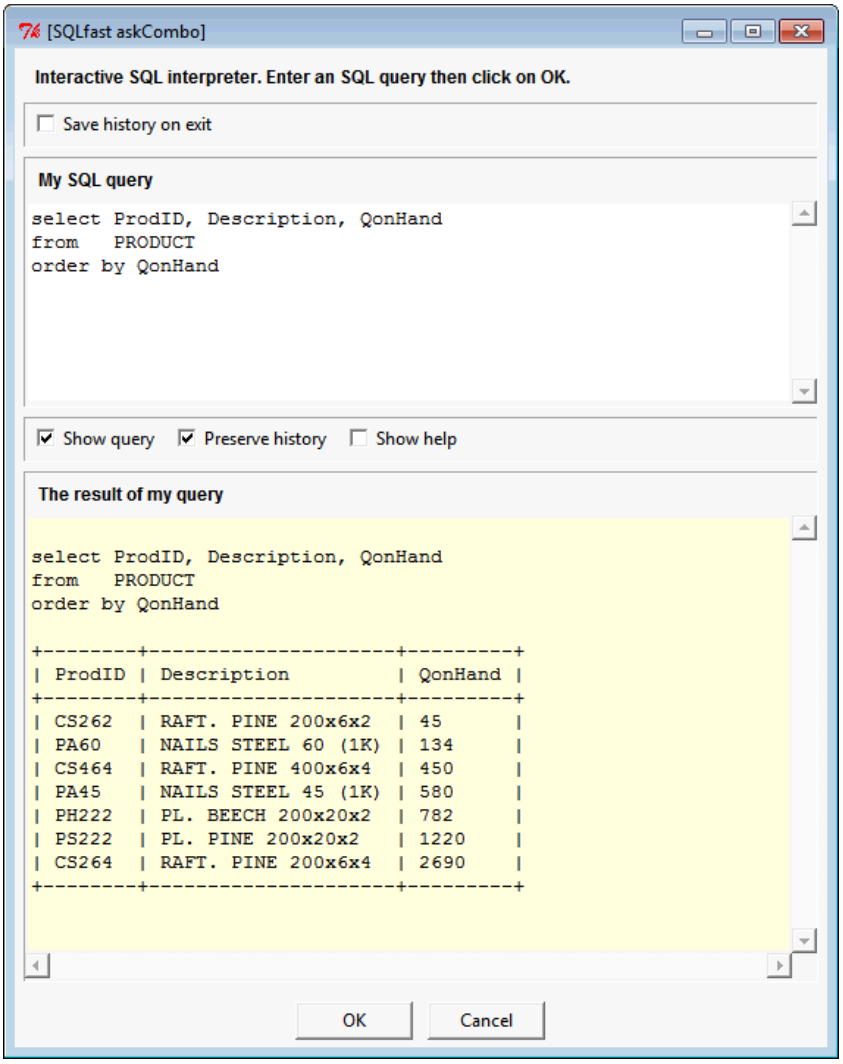


Figure 3.6 - Interface of the final version of the SQL interpreter

3.7 SQL interpreters as training support

One of the most common uses of SQL interpreters is to *support SQL learning*. The training scenario suggested by the SQL interpreter we have developed is fairly simple:

– the student reads the text of an exercise in some paper or electronic document (*pdf* or *web page* for instance), generally as a query expressed in plain language,
– the student enters the SQL query that answers it,
– the interpreter checks the syntactic validity of the SQL query,
– the interpreter executes the SQL query
– if the query is syntactically correct the interpreter displays its result set,
– otherwise it displays an error message.

If, as an answer to this exercise text:

what are the cities in which at least two customers live?

the student enters this **erroneous** SQL query:

```
select City from CUSTOMER where CustID >= '2'
```

the interpreter will find it syntactically correct and will display its result set, be it correct or not. The interpreter is happy with this answer, ... and so will be the student!

3.8 Semantic correctness of a query

The problem we put in light is the *semantic correctness* of the student's answer: does the SQL query **exactly translates** in the (formal) SQL language the *intension* of the (informal) query of the exercise?

A first solution that could come in mind would be to store in an *exercise database* both the text of each exercise and the expected SQL query that translates it. So, we could compare the query of the student with this reference query.

Unfortunately, this naive approach does not work (at all). This problem have proved very difficult to solve for two main reasons. First, natural languages are ambiguous. As every teacher has experimented, whatever the care with which one writes the text of an exercise, there will almost always be an student who will give it a plausible but unexpected interpretation. Secondly, even when we agree on the meaning of an exercise text, the richness and flexibility of the SQL language allows a wide variety of equivalent SQL queries to be written.

Let us show it through the following example, that seems quite simple and unambiguous:

which are the orders placed in 2017?

What does exactly mean expression "*which are the orders*"? Does it mean "*the value of column CustID of table CUSTORDER*"? or "*the values of all the columns of this table*"? Can we add the name, address and city of the customers who placed the orders? Can we add the total amount of each order? Can we sort the result set?

Expression "*placed in 2017*" is unambiguous but can be translated in many ways into SQL conditions:

```
DateOrd between '2017-01-01' and '2017-12-31';

DateOrd >= '2017-01-01' and DateOrd <= '2017-12-31';

DateOrd > '2016-12-31' and DateOrd < '2018-01-01';

DateOrd like '2017-%';

cast(DateOrd as char) like '2017-%';

substr(cast(DateOrd as char),1,4) = '2017';

year(DateOrd) = 2017;

extract(year from DateOrd) = 2017;
```

. . . and some more

The idea of storing a reference query that we can compare with the student's answer is absolutely and definitively unrealistic. So, let us drop it.

Instead of examining the query submitted by the student, we could execute it and **compare its result set** with that of the reference query. If they are identical, then the answer of the student is likely to be correct.

This is easy to check, but the conclusion of this comparison must be drawn more carefully:

- First, we should compare sets and not sequences, since row ordering, at least when not specified by the query, depends on the execution strategy chosen by the SQL engine.
- If the sets are different, we can conclude that the student's query is semantically incorrect; however, our diagnostic can be more precise to help the student modify her answer:
  - what is the size of each result set
  - how many rows does each result set contain that are not in the other one.
- If the sets are identical, the student's query may be correct, but this also may be a probabilistic accident! Indeed, two independent queries may produce the same result set though they have different meaning.
- The result sets may be the same while their sizes differ; this means that the student's result set includes duplicates (we suppose that the reference query discards duplicates).
- If both sets are empty, their comparison would be highly inconclusive.

In addition, showing the result set of their own answer and that of the reference query (of course without disclosing the latter) may help the students develop their answer or correct it.

3.9 An interactive SQL tutor

We consider the short analysis of the previous section as the set of minimal requirements for a variant of the SQL interpreter that we will call *Interactive SQL tutor*.

3.9.1 The database

For each exercise, we store its *full text* in plain language, its *reference solution* and the name of the *database* against which the query must be executed. Since the text may be rather long, a short summary, acting as a *title*, can be useful to allow the student to select the exercise in a list.

In addition, it is useful to classify exercises into *categories*. A category is a collection of exercises that share common properties, such as the structure of the solution (basic queries, sub-queries, join, grouped data, recursive queries, etc.) or the nature of the application domain (statistics, tree processing, text processing, active database, temporal database, etc.)

Finally, within each category, we could assign a *difficulty level* to each exercise.

This gives us the structure of table EXERCISE in which we will store the description of the exercises (Script 3.10).

```
create table EXERCISE(
       ExID        varchar(64) not null primary key,
       DB          varchar(128) not null,
       Category    varchar(32),
       Level       integer,
       Title       varchar(64),
       ProblemText varchar(512),
       RefQuery    varchar(512) );
```

Script 3.10 - Structure of the exercise database

Script 3.11 shows the definition of an exercise (medium difficulty: Level = 2) in the **sub-query** category (i.e., the solution of which is suggested to use a sub-query). It must be executed on database ORDERS.db. Its title, *Customers whose account is greater than the average of their city*, will be used by the student to select it.

3.9.2 The interface

Now, let us define the way students will use the SQL tutor. We call a *session*, the activity during which a student is invited to solve a set of exercises. We suggest that students proceed in two steps:

1. selecting the subset of exercises of the session; this subset is defined by one or several levels within a category (Figure 3.7),
2. trying to solve each (or some) of them (Figure 3.8).

```
insert into EXERCISE values(
   'Q02-07',
   'ORDERS.db',
   'sub-query',2
   'Customers whose account is greater than the average
    of their city',
   'Display the list of the Id of the customers whose account
    is greater than the average account of the customers
    of their city.',
   'select CustID,Name,City,Account from CUSTOMER as CUST
    where  Account > (select avg(Account) from CUSTOMER
    where City = CUST.City)'
);
```

Script 3.11 - Definition of an exercise

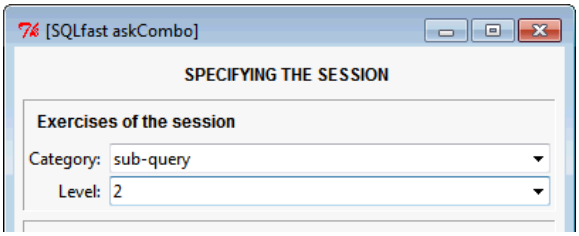


Figure 3.7 - Selecting the exercises of the session

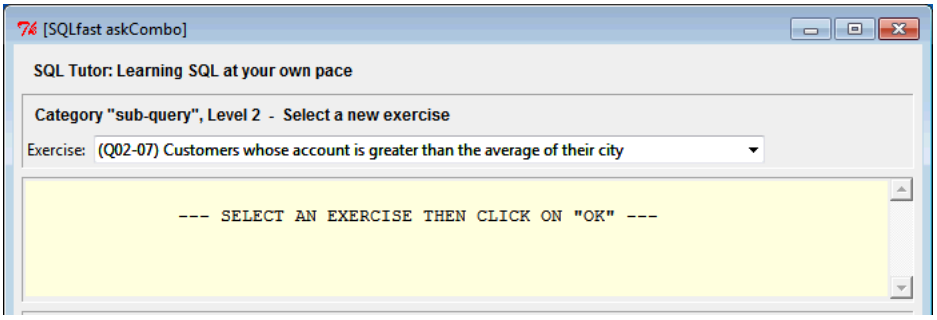


Figure 3.8 - Selecting an exercise to solve

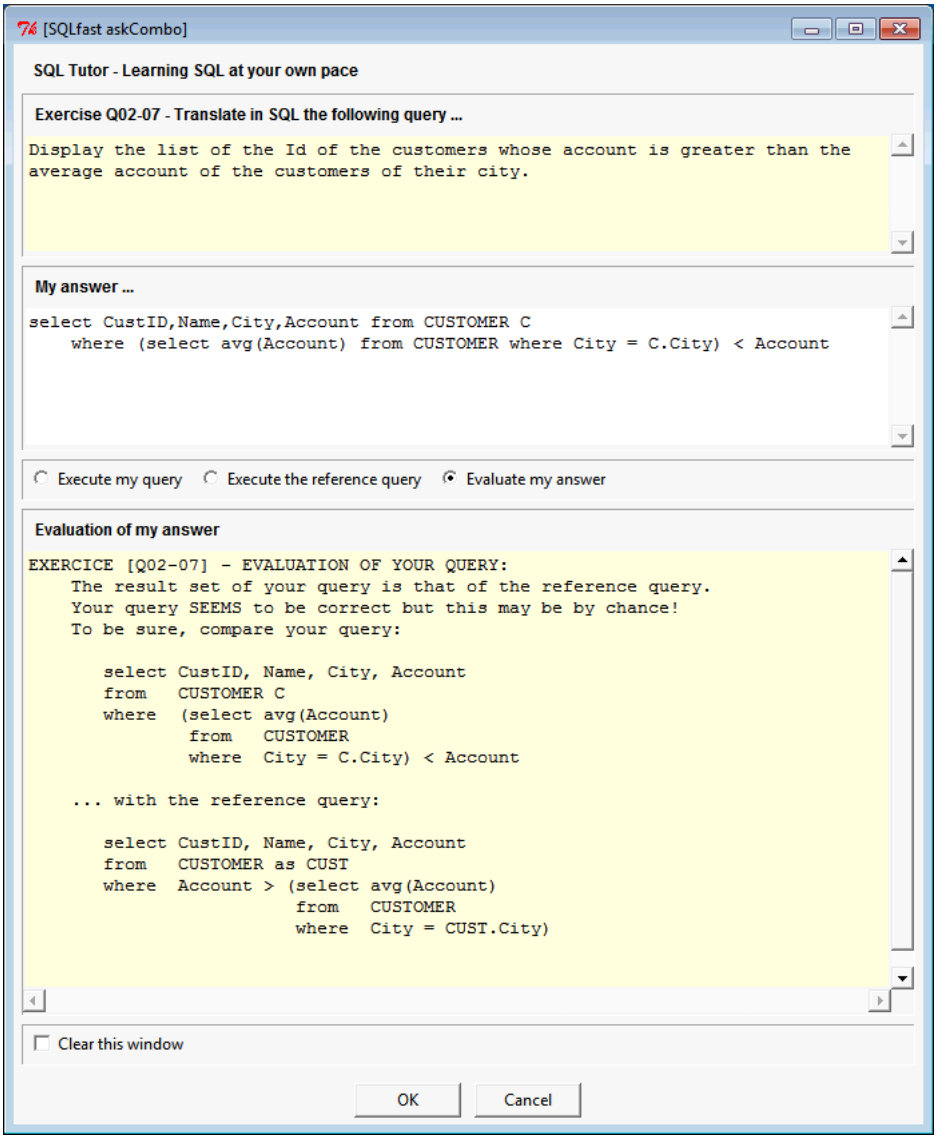


Figure 3.9 - The main dialogue box of the SQL tutor

The window of the SQL interpreter shown in Figure 3.6 is no longer sufficient to let the student practice the exercises of the session. We suggest the new layout of Figure 3.9. The first text area displays the text of the selected exercise. The student enters the SQL query that translates this text in the second area, check button **Evaluate my answer** then clicks on button **OK** to submit it. The tutor evaluates this SQL query and displays its diagnostic in the third text area.

The interface also allows the student to execute the query she just entered (button **Execute my query**) or to execute the reference solution (button **Execute the reference query**).

3.9.3 The tutor engine

Now, we examine the way the tutor will evaluate the SQL query submitted by the student. The analysis developed in Section 3.8 is based on a set of metrics defined as the size of the result sets of combinations of the student's query and the reference query.

Let **rQuery** be the reference query and **sQuery** the query submitted by the student (actually *the names of the variables that contain these queries*). The first one is extracted from the database and the second one from the second text area of the interface. The metrics are computed in Script 3.12:

- **N1**, the size of the result set of **rQuery**
- **N2**, the size of the result set of **sQuery**
- **N12**, the size of the difference of the result sets of **rQuery** and **sQuery**
- **N21**, the size of the difference of the result sets of **sQuery** and **rQuery**

We notice that metrics **N12** and **N21** are not computed on the source queries but on views defined on them. This trick is necessary to avoid set operator confusion when the source queries are themselves based on such operators.

We also note that metrics **N1** and **N2** are defined on multisets (in which duplicates, if any, are preserved) while metrics **N12** and **N21** are defined on pure sets (`union`, `intersect` and `except` eliminate duplicates).

```
extract N1 = select count(1) from ($rQuery$);
extract N2 = select count(1) from ($sQuery$);

create temp view _sview as select * from ($sQuery$);
create temp view _rview as select * from ($rQuery$);

extract N12 = select count(1) from (select * from _rview
                                      except
                                    select * from _sview);

extract N21 = select count(1) from (select * from _sview
                                      except
                                    select * from _rview);
```

Script 3.12 - Computing the raw metrics

The evaluation of **sQuery** derives from the reasoning rules of Script 3.13. They identify seven patterns:

1. *The reference result set is empty*: the diagnostic may be quite imprecise.

2. Considered as pure sets, the result sets are identical: the student's query *may be* considered correct.
3. In addition, considered as multisets, *the result sets are not identical*: the student's query may be considered correct but its result set includes duplicates.
4. The *reference result set contains rows that are not in the student's result set*.
5. The *student's result set contains rows that are not in the reference result set*.
6. The *reference result set is not empty but none of its rows appears in the student's result set*.
7. The r*eference result set includes some of the rows of the student's result set but not all*.

```
if (N1 = 0):
   < reference result set empty; imprecise diagnostic >
else:
   if (N12 + N21 = 0):
      < No row missing; answer probably correct >
      if (N1 = N2):
         < Same size >
      else:
         < Some duplicates >
   else:
      < Row(s) missing in some result set(s). Answer incorrect >
      if (N12 > 0 and N21 = 0):
      < Rows missing in student result set >
      if (N12 = 0 and N21 > 0):
         < Erroneous row(s) in student result set >
      if (N12 > 0 and N21 > 0):
         if (N1 = N12):
            < All the rows of student result set are erroneous >
         if (N1 > N12):
            < Some rows of student result set are erroneous >
```

Script 3.13 - Algorithm of the diagnostic

The state of the interface shown in Figure 3.9 illustrates the case of a student's query that is formally different from the reference query but that produces the same result set without duplicates (pattern n° 2). The diagnostic is appreciative but nevertheless invites the student to compare her query with the reference query. This shows that the tutor is more apt at detecting erroneous answers than identifying correct queries, an approach that is not uncommon among (human) teachers!

3.9.4 Improving the SQL tutor

The SQL tutor we have developed is just a prototype that can be enriched and improved in many ways. Let us mention some of them.

1. **Reducing false positives**
   The main weakness of the SQL tutor is the correct interpretation of patterns n° 1, 2 and 3, when result sets are identical, be they empty of not. The student's query *may* be correct but this can just be a statistical accident, that is, a *false positive*! We can reduce the uncertainty of such diagnostic by executing the queries against **several databases** with the same schema but with different contents. The value of column DB in table EXERCISE will then be a list of database names. The metrics collected for each database are consolidated to build a more reliable diagnostic.

2. **Refining the learning process**
   The training scenario enacted by the SQL tutor is fairly primitive. It would be more flexible, and therefore more efficient, if the tutor could help the student who fails to find the right answer or who explicitly asks for help. Some simple help features can be added: a *short reminder* of the underlying theory or of the SQL syntax, *hints* through which one or two levels of suggestion can be displayed when needed and *comments* associated to the reference solution.

3. **Encrypting reference queries**
   Honest students are those who try to solve the problems by themselves without resorting to dishonest sources. A particular dishonest source would be that provided by this script:

```
openDB SQLtutor.db;
select RefQuery from EXERCISE
where  ExID = 'Q02-07';
```

   The most obvious countermeasure to such work-around is to *encrypt* the values of column RefQuery, as shown below:

```
insert into EXERCISE values(
   'Q02-07',
    ...,
    encrypt('select CustID,Name,City,Account from CUSTOMER
             as CUST where Account > (select avg(Account)
             from CUSTOMER where City = CUST.City',
             'my_password'))
);
```

   The inverse operation will restore the original value:

```
select decrypt(RefQuery,'my_password') from EXERCISE
where  ExID = 'Q02-07';
```

   The database is now more secure, but not by far! A clever student could be tempted to have a look at the *source code* of the scripts, in which she will quickly discover the value of the decrypting key.
   A more secure technique can be developed as a couple of functions stored in Python module UserUDFlib_for_SQL.py or UserUDFlib_for_SQLfast.sql (the source code of which being, of course, unavailable!). The first function appears

to be an innocuous function, for instance that returns the current time (e.g., get_time()). It is called by the main script at start time. In addition to its apparent action, this function stores a kind of *cookie* in the UDF module. The second function (e.g., get_key()) is called to get the decrypting key. It returns the actual key if the cookie exists or a fake key otherwise. Understanding this mechanism and reproducing it to get the reference queries is much more complicated, at least for novice hackers. Considering that this application is not critical, this technique can be quite sufficient to ensure a fairly good level of security.

In the prototype application available in the SQLfast distribution, the reference queries can be, if desired, converted in the database in the *base64* coding scheme:

```
update EXERCISE set RefQuery = b64encode(RefQuery,'std');
```

4. Student management

The SQL tutor focuses on the availability of collections of exercises as well as the evaluation of student's answers. In the scenarios provided by this application, the student is just an external, unidentified user. This extension suggests to integrate students in the tutoring process. This encompasses two aspects:

- first allowing students to suspend, resume and record their sessions
- secondly, allowing teachers to register students, to examine their scores and to evaluate them.

5. Exercise management

A last service application will be welcome: a data entry interface to allow teachers to examine, enter, check, modify and delete exercises.

3.10 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory SQLfast/Scripts/Case-Studies/Case_SQL_Interpreters. Actually, they can be run from main script **SQL-Interpreter-MAIN.sql**, that displays the selection box of Figure 3.10.

The scripts of the SQL tutor have been stored in subdirectory **SQL-Tutor**. The student component (the only one so far) of the SQL tutor is run by executing script **SQL-Tutor-STUDENT.sql**. This script executes an exercise module script that loads in the database the set of exercises the student is interested. in. This script is given a name starting with prefix **'SQL-Exercises-'**. Several exercise scripts, both in English and in French, have been included in the SQLfast distribution.

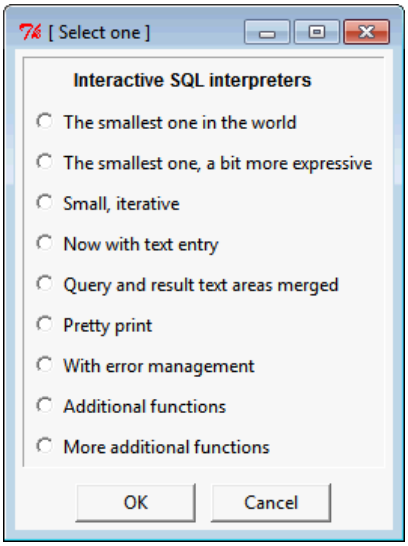


Figure 3.10 - Selecting an SQL interpreter

The scripts of this case study are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

Case study **4**

Schema-less databases - Part 1

Objective: This document is the first of a series of three case studies that explore alternative data models that organize the data in structures that provide more flexibility than standard relational tables. We observe that this flexibility makes it easier to dynamically modify the schema of the database but, as an unfortunate consequence, the schema is less expressive. In some of these data models, the schema is practically devoid of any information, hence the name *schema-less*.

In this study, we explore two extreme data models. In the *Universal table model*, according to which the data of all the source tables are stored in a single table comprising the union of the columns of the source tables. In the second model, called *Column-oriented model*, each column of the source tables is implemented in an independent table.

Scripts migrating the data of the ORDERS.db database to these models, and conversely, are developed.

Keywords: non-relational data model, NoSQL, schema-less database, universal table model, universal relation, column-oriented data model, Cassandra, data migration, schema conversion

4.1 Introduction

The organization of data in columns and rows stored in tables, the so-called *standard relational model*, is the most usual way to structure a database. Each table collects the information on a set of real world entities (customers, products, employees, movies, etc.). In a table, each entity is described by one and only one row and each column stores a definite property of these entities. The name of the table is close to the name of the entity set (CUSTOMER, EMPLOYEE) and the name of each column evokes a property (Name, Address, City, Price, Skill).

However, this natural way to organize data is not the only one. When performance and scalability become premium criteria, current relational database managers may prove inflexible to cope with high volume of rapidly evolving and growing data.

This is where the so-called **NoSQL** data managers begin to enter the scene. Their architecture favors scalability (they allow data to grow several orders of magnitude without administrative burden nor loss of performance), distribution, redundancy and flexible data structures. However, there is a price to pay: practically no centralized control of data quality, no high-level query language, no standard data structures and interface, no strong transactions. Most often these databases have no expressive data structures, in such a way that data integrity, quality control, semantic validation and the like are the responsibility of the application programs.

This document and the next two ones are a short but practical introduction to some of these alternative data models. Instead of working with native NoSQL DBMS (which one to choose among the nearly hundred of them?), we will express these models in relational data structures, develop scripts to migrate *source tables* (the four tables of database ORDERS.db) into these new models (and back) and write some queries for these data.

A last word: the title of these case studies is standard but misleading. Each database, whatever its model, *has a schema* but this schema can be more or less abstract and generic. *Schema-less* qualifies databases the structure of which tends to be simplistic, unspecific and merely technical. In other words, the names of objects of the schema (tables and columns) no longer denote entities and properties of the real world.[1]

4.2 The Universal table model

This first alternative data model organizes the data of a database in **a single table** comprising one *not null* column (the table *primary key*) and a series of *nullable*

1. "*The absence of a schema has some flexibility advantages, although for querying the data, the absence of a schema presents some challenges to people accustomed to a classic RDBMS*. "–Iran Hutchinson. http://www.odbms.org/blog/2014/01/big-data-three-questions-to-intersystems/

columns. Each entity of the application domain (i.e., the part of the real world about which we want to record information) is described by a row where the relevant columns have been assigned a value while all the other ones are set to *null*. If needed, an additional column, let us call it **EType**, can be added to identify the type of the entity. These rows are assigned an abstract technical Id that act as a global primary key, independently of the primary keys of the source tables.

Script 4.1 shows an example of creation query for this table, called **DATA**. The abstract Id is called **Entity** since it uniquely denotes each entity of the application domain, whatever its type.

```
create table DATA(
   Entity      integer not null primary key autoincrement,
   EType       char(18) not null,
   CustID      char(10),
   Name        char(32),
   Address     char(60),
   City        char(30),
   Cat         char(2),
   Account     decimal(9,2),
   ProdID      char(15),
   Description char(60),
   Price       integer,
   QonHand     integer,
   OrdID       char(12),
   DateOrd     date,
   Qord        integer);
```

Script 4.1 - Table DATA collecting rows of different patterns (from the standard ORDERS database) in a single table.

4.2.1 Generating the Universal table data from source tables

Script 4.2 migrates the data of database ORDERS and Figure 4.1 shows the contents of table DATA. The fact that most columns are set to *null* is less of a problem than it seems to be: in many DBMS implementations, a *null* value just costs one bit.

When possible, the same column is used to represent different roles according to the entity described. It is the case for CustID, OrdID and ProdID, that each appear in two entity types, once as a primary key and once as a foreign key.

4.2.2 From the Universal table back to the standard tables

Recovering source data from universal table DATA is straightforward. Script 4.3 declares a set of views that present the data according to the standard source tables.

```
insert into DATA(EType,CustID,Name,Address,City,Cat,Account)
select 'CUST',CustID,Name,Address,City,Cat,Account from CUSTOMER;
insert into DATA(EType,ProdID,Description,Price,QonHand)
select 'PROD',ProdID,Description,Price,QonHand from PRODUCT;
insert into DATA(EType,OrdID,CustID,DateOrd)
select 'ORD',OrdID,CustID,DateOrd from CUSTORDER;
insert into DATA(EType,OrdID,ProdID,Qord)
select 'DET',OrdID,ProdID,Qord from DETAIL;
```

Script 4.2 - Migration of the ORDERS database to table DATA

```
create view CUSTOMER as
select  CustID,Name,Address,City,Cat,Account
from    DATA where EType = 'CUST';
create view CUSTORDER(OrdID,CustID,DateOrd) as
select  OrdID,CustID,DateOrd
from    DATA where EType = 'ORD';
. . .
```

Script 4.3 - Rebuilding the tables of the ORDERS database through views defined on table DATA

4.2.3 Application

As an application example, *Customer* entities and their associated *order* entities can be retrieved efficiently in one, join-less, **`select`** operation (Script 4.4). The result of these queries is shown in Figure 4.2. At the physical level, this model allows indexes to be defined on columns that appear in different entity types, providing a function somewhat similar to Oracle *clusters*.

```
select  CustID,Name,Address,City,OrdID,DatOrd
from    DATA
where   EType in ('CUST','ORD')
and     CustID = 'C400';

select  CustID,Name,Address,City,OrdID,DatOrd
from    DATA C, DATA O
where   C.EType = 'CUST'
and     O.EType = 'ORD'
and     C.CustID = 'C400'
and     O.CustID = C.CustID;
```

Script 4.4 - Extracting data of customer C400 and his orders in join-less query (top) and its join-based equivalent

Ety	EType	CustID	Name	Address	City	Cat	Acc	OrdID	DateOrd	Qord	ProdID	Description	Price	QonH
1	CUST	B112	HANSENNE	23, r. Dumont	Poitiers	C1	1250	--	--	--	--	--	--	--
2	CUST	C123	MERCIER	25, r. Lemaître	Namur	C1	-2300	--	--	--	--	--	--	--
3	CUST	B332	MONTI	112, r. Neuve	Genève	B2	0	--	--	--	--	--	--	--
4	CUST	F010	TOUSSAINT	5, r. Godefroid	Poitiers	C1	0	--	--	--	--	--	--	--
5	CUST	K111	VANBIST	180, r. Florimont	Lille	B1	720	--	--	--	--	--	--	--
6	CUST	S127	VANDERKA	3, av. des Roses	Namur	C1	-4580	--	--	--	--	--	--	--
7	CUST	B512	GILLET	14, r. de l'Eté	Toulouse	B1	-8700	--	--	--	--	--	--	--
8	CUST	B062	GOFFIN	72, r. de la Gare	Namur	B2	-3200	--	--	--	--	--	--	--
9	CUST	C400	FERARD	65, r. du Tertre	Poitiers	B2	350	--	--	--	--	--	--	--
10	CUST	C003	AVRON	8, ch. de la Cure	Toulouse	B1	-1700	--	--	--	--	--	--	--
11	CUST	K729	NEUMAN	40, r. Bransart	Toulouse	--	0	--	--	--	--	--	--	--
12	CUST	F011	PONCELET	17, Clôs des Erables	Toulouse	B2	0	--	--	--	--	--	--	--
13	CUST	L422	FRANCK	60, r. de Wépion	Namur	C1	0	--	--	--	--	--	--	--
14	CUST	S712	GUILLAUME	14a, ch. des Roses	Paris	B1	0	--	--	--	--	--	--	--
15	CUST	D063	MERCIER	201, bvd du Nord	Toulouse	--	-2250	--	--	--	--	--	--	--
16	CUST	F400	JACOB	78, ch. du Moulin	Bruxelles	C2	0	--	--	--	--	--	--	--
17	ORD	K111	--	--	--	--	--	30178	2013-12-21	--	--	--	--	--
18	ORD	C400	--	--	--	--	--	30179	2013-12-22	--	--	--	--	--
19	ORD	S127	--	--	--	--	--	30182	2013-12-23	--	--	--	--	--
20	ORD	C400	--	--	--	--	--	30184	2013-12-23	--	--	--	--	--
21	ORD	F011	--	--	--	--	--	30185	2014-01-02	--	--	--	--	--
22	ORD	C400	--	--	--	--	--	30186	2014-01-02	--	--	--	--	--
23	ORD	B512	--	--	--	--	--	30188	2014-01-03	--	--	--	--	--
24	DET	--	--	--	--	--	--	30178	--	25	CS464	--	--	--
25	DET	--	--	--	--	--	--	30179	--	20	PA60	--	--	--
26	DET	--	--	--	--	--	--	30179	--	60	CS262	--	--	--
27	DET	--	--	--	--	--	--	30182	--	30	PA60	--	--	--
28	DET	--	--	--	--	--	--	30184	--	120	CS464	--	--	--
29	DET	--	--	--	--	--	--	30184	--	20	PA45	--	--	--
30	DET	--	--	--	--	--	--	30185	--	15	PA60	--	--	--
31	DET	--	--	--	--	--	--	30185	--	600	PS222	--	--	--
32	DET	--	--	--	--	--	--	30185	--	260	CS464	--	--	--
33	DET	--	--	--	--	--	--	30186	--	3	PA45	--	--	--
34	DET	--	--	--	--	--	--	30188	--	70	PA60	--	--	--
35	DET	--	--	--	--	--	--	30188	--	92	PH222	--	--	--
36	DET	--	--	--	--	--	--	30188	--	180	CS464	--	--	--
37	DET	--	--	--	--	--	--	30188	--	22	PA45	--	--	--
38	PROD	--	--	--	--	--	--	--	--	--	CS262	RAFT. PINE 200x6x2	75	45
39	PROD	--	--	--	--	--	--	--	--	--	CS264	RAFT. PINE 200x6x4	120	2690
40	PROD	--	--	--	--	--	--	--	--	--	CS464	RAFT. PINE 400x6x4	220	450
41	PROD	--	--	--	--	--	--	--	--	--	PA45	NAILS STEEL 45 (1K)	105	580
42	PROD	--	--	--	--	--	--	--	--	--	PA60	NAILS STEEL 60 (1K)	95	134
43	PROD	--	--	--	--	--	--	--	--	--	PH222	PL. BEECH 200x20x2	230	782
44	PROD	--	--	--	--	--	--	--	--	--	PS222	PL. PINE 200x20x2	185	1220

Figure 4.1 - Contents of the Universal table associated to the ORDERS database

CustID	Name	Address	City	OrdID	DateOrd
C400	FERARD	65, r. du Tertre	Poitiers	--	--
C400	--	--	--	30179	2013-12-22
C400	--	--	--	30184	2013-12-23
C400	--	--	--	30186	2014-01-02

CustID	Name	Address	City	OrdID	DateOrd
C400	FERARD	65, r. du Tertre	Poitiers	30179	2013-12-22
C400	FERARD	65, r. du Tertre	Poitiers	30184	2013-12-23
C400	FERARD	65, r. du Tertre	Poitiers	30186	2014-01-02

Figure 4.2 - Result of the execution of queries 4.4

4.2.4 Origin of the Universal table model

This data model is derived from the concept of *Universal relation* of the relational theory, though the two constructs are quite different. The *Universal table* is the union of source tables while the *Universal relation* would be obtained by outer joins of the source tables. One of the main goal of the *Universal relation* is to offer a simpler relational algebra, devoid of the join operator.

4.2.5 Comments

This model brings much flexibility for schema evolution. For example,

- Since all columns are nullable, new columns can be added without impact on the contents of the table,
- Each entity can be structured independently of the others of its type (e.g., a definite customer entity can be assigned a value of column DESCRIPTION),
- New entity types can be added without structural modification, provided the required columns are available.

Such loosely structured data model is appropriate when the nature and the structure of entities are rapidly evolving.

However, it does not provide an easy support for the native SQL integrity constraints (unique keys, foreign keys, not null column). First conclusion: more flexibility implies lesser expressivity of the schema. More on this later.

The scripts of this section are available in file **Schemaless1-Universal.sql** in directory **Case_Schemaless**.

4.3 Column-oriented data model

This model represents *each property by a specific table* and can be considered the opposite of the *Universal table* model. As compared to the four source tables of database ORDERS.db, this model will include one table (called *column-table*) per source column. For instance, the data of source table CUSTOMER are distributed among five column-tables, each comprising the primary key (**CustID**) plus one source column, respectively for Name, Address, City, Cat, Account values. Primary key CustID is common to these tables to allow the reconstruction of source table CUSTOMER through joins. The second column is simply named **Value**, so that all the column-tables (but one!) deriving from a source table have the same schema. Script 4.5 creates the tables for storing the data of database ORDERS.db.

```
create table CUST_NAME(
   CustID    char(10) not null primary key,
   Value     varchar(32) not null);
create table CUST_ADDRESS(
   CustID    char(10) not null primary key,
   Value     char(60) not null);
create table CUST_CITY(
   CustID    char(10) not null primary key,
   Value     char(30) not null);
...
create table ORD_CUSTID(
   OrdID     char(12) not null primary key,
   Value     char(10) not null);
create table ORD_DATEORD(
   OrdID     char(10) not null primary key,
   Value     date not null);
...
```

Script 4.5 - Creating column-tables

4.3.1 Generating the Column-oriented data from the source tables

Script 4.6 migrates the data from the source database. Figure 4.3 shows the contents of tables **CUST_NAME** and **CUST_ADDRESS**.

We observe that we have not created a specific table for the primary key of the source tables. Indeed, its values are stored in each of the column-table. This structure is valid provided (1) the source table comprises at least one column in addition to the primary key and (2) *null* values are also represented. If condition (1) is not satisfied, then the source table can be taken as a degenerated column-table.

```
insert into CUST_NAME(CustID,Value)
       select CustID,Name from CUSTOMER;
insert into CUST_ADDRESS(CustID,Value)
       select CustID,Address from CUSTOMER;
insert into CUST_CITY(CustID,Value)
       select CustID,City from CUSTOMER;
...
insert into ORD_CUSTID(OrdID,Value)
       select OrdID,CustID from CUSTORDER;
insert into ORD_DATEORD(OrdID,Value)
       select OrdID,DateOrd from CUSTORDER;
insert into DET_QORD(OrdID,ProdID,Value)
       select OrdID,ProdID,Qord from DETAIL;
insert into PROD_DESCR(ProdID,Value)
       select ProdID,Description from PRODUCT;
insert into PROD_PRICE(ProdID,Value)
       select ProdID,Price from PRODUCT;
...
```

Script 4.6 - Loading column-tables

CustID	Value
B112	HANSENNE
C123	MERCIER
B332	MONTI
F010	TOUSSAINT
K111	VANBIST
S127	VANDERKA
B512	GILLET
B062	GOFFIN
C400	FERARD
...	...

CustID	Value
B112	23, r. Dumont
C123	25, r. Lemaître
B332	112, r. Neuve
F010	5, r. Godefroid
K111	180, r. Florimont
S127	3, av. des Roses
B512	14, r. de l'Eté
B062	72, r. de la Gare
C400	65, r. du Tertre
...	...

Figure 4.3 - Contents of column-tables CUST_NAME and CUST_ADDRESS

We observe that source table DETAIL is transformed into a column-table the schema of which is different from that of the other column-tables. Indeed, this table has a composite primary key and only one column in addition to its primary key. Therefore, the source table and the unique column-table are exactly the same, bar some of their column names.

4.3.2 From the Column-oriented tables back to the source tables

Script 4.7 recovers source table CUSTOMER by joining the tables that represent its columns.

```
select C1.CustID,C1.Value as Name,C2.Value as Address,
       C3.Value as City,C4.Value as Cat,C5.Value as Account
from CUST_NAME C1,CUST_ADDRESS C2,
     CUST_CITY C3,CUST_CAT C4,CUST_ACCOUNT C5
where C2.CustID = C1.CustID
and   C3.CustID = C1.CustID
and   C4.CustID = C1.CustID
and   C5.CustID = C1.CustID;
```

Script 4.7 - Rebuilding source table CUSTOMER from column tables

4.3.3 Column-oriented model with abstract primary keys

In the column-tables, column CustID is intended to denote entities. This technique is fine for CUSTOMER table for two reasons:

- the customer Ids are short and meaningless
- the primary key of the table comprises one column only.

Performance problems may arise for large primary keys, which are duplicated in each column-table. In addition, the primary indexes will also be quite large and therefore will entail lookup time penalties.

Composite primary keys are more tricky to cope with. In such cases, column tables are made up of more than two columns: at least two for the primary key + one. So, there is no common table format, which makes the model no longer uniform.

Hence the idea to denote entities with abstract Ids, as we did in table DATA of the *Universal table* model (Script 4.1). This way, we can define a column-table for each source column, be it a component of a primary key or not, making the representation quite uniform (Script 4.8). We observe that table CUST_CUSTID has a spacial structure: Column Entity is declared *auto increment* and a unique predicate is stated on column Value.

To load the column-tables, we proceed in two steps. First, we load the table derived from the primary key in order to generate the abstract Ids:

```
insert into CUST_CUSTID(Value)
    select CUSTID from CUSTOMER order by CUSTID;
```

Then, we load the other column-tables with the abstract Ids that have been produced:

```
insert into CUST_NAME(Entity,Value)
  select Entity,Name
  from   CUST_CUSTID I, CUSTOMER C where I.Value = C.CustID;
```

```
create table CUST_CUSTID(
   Entity    integer not null primary key autoincrement,
   Value     char(10) not null unique);
create table CUST_NAME(
   Entity    integer  not null primary key,
   Value     char(32) not null);
create table CUST_ADDRESS(
   Entity    integer  not null primary key,
   Value     char(60) not null);
create table CUST_CITY(
   CustID    integer  not null primary key,
   Value     char(30) not null);
...
create table DET_PK(
   Entity    integer not null primary key autoincrement,
   Value1    char(12),
   Value2    char(15));
create table DET_ORDID(
   Entity    integer  not null,
   Value     char(12) not null);
create table DET_PRODID(
   Entity    integer  not null,
   Value     char(15) not null);
create table DET_QORD(
   Entity    integer  not null,
   Value     integer  not null);
```

Script 4.8 - Creating column-tables based on abstract primary keys

The case of composite primary keys will be tackled in a slightly different way. We create a special dictionary table the role of which is to generate the abstract Ids and to map the primary key composite values to these abstract values. For table DETAIL, it is declared as follows:

```
create temp table DET_PK(
   Entity    integer not null primary key autoincrement,
   Value1    char(12),
   Value2    char(15));
```

The dictionary table is filled by the primary key values of DETAIL:

```
insert into DET_PK(Value1,Value2)
   select OrdID,ProdID from DETAIL order by OrdID,ProdID;
```

The column-tables are then loaded with the abstract Ids that have been produced:

```
insert into DET_ORDID(Entity,Value)
   select Entity,OrdID from DET_PK K, DETAIL D
   where K.Value1 = D.OrdID and K.Value2 = D.ProdID;
```

The uniqueness property of the source primary key of table CUSTOMER is ensured by the **unique** predicate in column-table CUST_CUSTID.

Unfortunately, the expression of composite primary keys is no longer straightforward, since they hold on joins of column tables. The following query lists the primary key values that are not unique among the column-tables of DETAIL:

```
select O.Value as OrdID,P.Value as ProdID,count(*) as Duplicate
from   DET_ORDID O,DET_PRODID P
where  O.Value = P.Value
group by O.Value,P.Value
having Duplicate > 1;
```

4.3.4 About performance

We have measured the time needed to load the column-tables and to rebuild the source table for a CUSTOMER table containing 32,768 rows (= **32K** rows). These figures are collected in Figure 4.4 (times in **ms**.). If the first technique, based on source primary keys, is a bit faster for the generation of column-tables, the second technique, based on abstract Ids, appears to be *four times faster* for rebuilding the source table rows. These times are linear *wrt* the table size: doubling the number of rows doubles the execution times.

Entity Id	Load time	Rebuild time
primary key	279	167
abstract entity id	318	42

Figure 4.4 - Execution time of the main conversion processes for the two approaches

4.3.5 Not all Column-oriented models are created equal

The term *column-oriented* also names a family of so-called NoSQL data managers, the main representative of which certainly is *Cassandra*. In the latter case, this name is misleading. The original data model of *Cassandra* is particularly convoluted, using seemingly new concepts under proprietary vocabulary. Actually, this historical model is being abandoned and replaced by an alternative model, CQL, made up of concept like keyspace (= database), type, table, column, primary key, index, trigger as well as operators select, insert, update and delete, that is, ironically, ... a subset of SQL!

4.3.6 Comments

This model may bring several advantages. First, the evaluation of a query only gets access to the columns that are relevant and to no other ones. This reduces the volume of data read from the external memory. This is particularly important for tables that comprise many, possibly large, columns. Another advantage is that, by storing the columns on different devices, their access can be performed in parallel, thus reducing response time.

On the other hand, the expression of some common integrity constraints may prove complex.

The way *null* values are represented is worth a short discussion. The first technique, that has been chosen in this discussion, consists in inserting, for each source row, a row in each column-table, be its value *null* or not *null*. According to another way, *null* values are represented by a missing row in the column table. Both approaches are consistent but when we rebuild the source table, or a part of it, the first representation will need *inner joins* while the second one will require *outer joins*.

The scripts of the column-oriented model are available in directory **SQLfast/Scripts/Case-Studies/Case_Schemaless**. They can be run from main script **Schemaless-MAIN.sql** (see Part 3).

A discussion and links to *column-oriented database managers* can be found in reference: http://en.wikipedia.org/wiki/Column-oriented_DBMS.

Case study **5**

Schema-less databases - Part 2

Objective: This document studies a third family of alternative data models, namely the *Key-Value* data structure. In these models, the information is represented by triples that each defines the value of an attribute of an entity. Several approaches are described, with increasing levels of genericity. As in Part 1, migration and exploitation scripts are developed for the ORDERS.db database.

Keywords: non-relational data model, NoSQL, schema-less database, key-value model, triple, triplestore, RDF, SPARQL, description logic, A-BOX, OWL, Redis, Berkley DB, data migration, schema conversion

5.1 Introduction

The *Key-Value model* is at the basis of many NoSQL DBMS (though we will see that this name is misleading) but also of knowledge representation formalisms such as OWL. The basic idea is the following: instead of storing value FERARD into column **Name** of the CUSTOMER row describing customer C400, we store the triple (C400, Name, FERARD) into a 3-column table named, say, **M_CUSTOMER**.

A representative of this family of data model has been studied in Chapter 16 of the SQLfast tutorial (*Key-Value output format*), in which the data of a table are transformed into the format key = value through style script UTIL-SELECT-parameters-for-KEY-VALUE:

```
CustID = B112
Name = HANSENNE
Address = 23, r. Dumont
etc.
```

In this section we will examine several ways to exploit the *Key-value* format, in which some schema information (metadata), such as *column names*, are moved to user data: Name is no longer the name of a column of the schema but just plain user data.

The term *Key* being too technical for the domain we will discuss, we will use the term ***attribute*** instead (**Name** is not a *key* of customers but an *attribute* or a property thereof). However, we will keep the name *Key-Value* to designate this family of models, a term which has become standard.

5.2 The Key-Value model - Version 1

The first approach consists in storing the data about each entity type in a specific table. So, we create tables **M_CUSTOMER**, **M_CUSTORDER**, **M_DETAIL** and **M_PRODUCT**, each containing the data of the corresponding table of the source database.

Each table comprises three columns:

- **Entity**, which denotes an entity of the application domain (a customer, an order, a detail, a product), either through some kind of primary key (CustID for customers for example) or through an abstract, technical identifier such as a simple integer,
- **Attribute**, which contains the name of an attribute (e.g., Name, City, DateOrd, Price),
- **Value**, which contains the value of this attribute for this entity (e.g., Férard, London, 2017-03-28, 123.5).

For instance, row (C400, Name, FERARD) of table **M_CUSTOMER** tells that the customer denoted by **C400** has an attribute called **Name**, the value of which is **FERARD**. Such rows are called *triples*. Excerpts of this table, which comprises 16 x 6 = **96** triples, are shown in Figure 5.1.[1]

```
+--------+-----------+----------------------+
| Entity | Attribute | Value                |
+--------+-----------+----------------------+
| B062   | CustID    | B062                 |
| B062   | Name      | GOFFIN               |
| B062   | Address   | 72, r. de la Gare    |
```

1. The way *null* values are displayed in the result of `select` statements (here `--`) is specified by parameter dispNull in initialization file SQLfast.ini.

B062	Account	-3200.0
B062	Cat	B2
B062	City	Namur
B112	CustID	B112
B112	Name	HANSENNE
B112	Address	23, r. Dumont
B112	City	Poitiers
B112	Cat	C1
B112	Account	1250.0
...	...	...
C400	CustID	C400
C400	Name	FERARD
C400	Address	65, r. du Tertre
C400	City	Poitiers
C400	Cat	B2
C400	Account	350.0
D063	CustID	D063
D063	Name	MERCIER
D063	Address	201, bvd du Nord
D063	City	Toulouse
D063	Cat	--
D063	Account	-2250.0
...	...	...

Figure 5.1 - Excerpts of table M_CUSTOMER

5.2.1 Generating Key-Value data from the source tables

Script 5.1 creates the first two tables, M_CUSTOMER and M_CUSTORDER. We notice that, apart from their names, these tables have exactly the same definition. We also observe that column Value has been declared **`varchar`**, whatever the type, numeric or character, of the source values that we will store in it.

Loading the data from the source database into the Key-value tables is carried out by Script 5.2, which is simple and intuitive, but may not be very efficient for large tables with many columns since it parses table CUSTOMER once for each of its columns.

Script 5.3 suggests an alternative technique to migrate the source data. It parses source table CUSTOMER only once. For each source row and for each source column, it inserts an attribute and its value in target table M_CUSTOMER.

The fact that source values are first stored in SQLfast variables `cus`, `nam`, `add`, `cit`, `cat` and `acc` before being inserted as constants in `insert` queries requires the use of special variable-value substitution delimiter **`§`** instead of **`$`**. This way, internal single quotes are doubled in character string constants, as required by the SQL syntax.

```
create table M_CUSTOMER(
   Entity    varchar(32) not null,
   Attribute varchar(32) not null,
   Value     varchar(32),
   primary key (Entity,Attribute));
create table M_CUSTORDER(
   Entity    varchar(32) not null,
   Attribute varchar(32) not null,
   Value     varchar(32),
   primary key (Entity,Attribute));
. . .
```

Script 5.1 - Structure of tables M_CUSTOMER and M_CUSTORDER

```
insert into M_CUSTOMER (Entity,Attribute,Value)
       select CustID, 'CustID', CustID from S_CUSTOMER;
 union select CustID, 'Name', Name from S_CUSTOMER;
 union select CustID, 'Address', Address from S_CUSTOMER;
 union select CustID, 'City', City from S_CUSTOMER;
 union select CustID, 'Cat', Cat from S_CUSTOMER;
 union select CustID, 'Account',
              cast(Account as char) from S_CUSTOMER;
```

Script 5.2 - Generation of the contents of table M_CUSTOMER (pure SQL version)

```
for cus,nam,add,cit,cat,acc = [select CustID,Name,Address,City,
                               Cat,Account from CUSTOMER]
   insert into M_CUSTOMER values
          ('§cus§','CustID','§cus§'),
          ('§cus§','Name','§nam§'),
          ('§cus§','Address',§add§),
          ('§cus§','City','§cit§'),
          ('§cus§','Cat',
           case when ('§cat§' = '') then null else '§cat§' end),
          ('§cus§','Account',cast($acc$ as char));
```

Script 5.3 - Generation of the contents of table M_CUSTOMER (procedural version)

Observing that column **Cat** is *nullable*, we have to decide the way *null* values will be represented: either by an attribute triple with a *null* value or by discarding this triple for this entity. As suggested in Figure 5.1, we choose the first approach. Since the `SQL-for` loop returns an empty string for each *null* value, we must translate this

empty string into a *null* value before inserting the triple in M_CUSTOMER. The SQL conversion expression is simple:

```
case when '§cat§' = '' then null else '§cat§' end
```

We also note that the values of column Account, being numeric, must be converted in character strings, which is required by the type of column Value.

5.2.2 The case of composite primary keys

All this works fine for source tables the primary key of which comprises one column only. The case of table DETAIL is a bit more complex. The first idea would be to create a Key-value table with four columns (Script XXX).

```
create table M_DETAIL(
   Entity1   varchar(32) not null,
   Entity2   varchar(32) not null,
   Attribute varchar(32) not null,
   Value     varchar(32),
   primary key (Entity1,Entity2,Attribute));
```

Script 5.4 - M_DETAIL: strange triples with four components!

This seems a bad idea in that it produces tables with different schemas, therefore spoiling the nice uniform data structures. So, we better try to preserve the uniformity of the concept of entity Id. Two valid techniques:

1. Denoting entities with an abstract identifier, such a pure integer, as we did in some techniques of Part 1.
2. Building a single-valued id from the components of the primary key. For table DETAIL, considering the syntax of OrdID and ProdID, their concatenation would be fine:

```
OrdID||'-'||ProdID
```

With the second technique, the result is quite satisfying (Figure 5.2).

Entity	Attribute	Value
30178-CS464	OrdID	30178
30178-CS464	ProdID	CS464
30178-CS464	Qord	25
30179-CS262	OrdID	30179
30179-CS262	ProdID	CS262
30179-CS262	Qord	60
...	...	...
30188-PH222	OrdID	30188

```
| 30188-PH222 | ProdID     | PH222 |
| 30188-PH222 | Qord       | 92    |
+-------------+------------+-------+
```

Figure 5.2 - Building triples from a table with a composite primary key

5.2.3 Rebuilding the source tables from their Key-Value expression

Script 5.5 rebuilds source table CUSTOMER by a multiple self-join of table M_CUSTOMER.

```
insert into TMP_CUSTOMER(CustID,Name,Address,City,Cat,Account)
select M1.Value as CustID, M2.Value as Name,
       M3.Value as Address,
       M4.Value as City, M5.Value as Cat,
       cast(M6.Value as decimal) AS Account
from   M_CUSTOMER M1, M_CUSTOMER M2, M_CUSTOMER M3,
       M_CUSTOMER M4, M_CUSTOMER M5, M_CUSTOMER M6
where  M2.Entity = M1.Entity and M3.Entity = M1.Entity
and    M4.Entity = M1.Entity and M5.Entity = M1.Entity
and    M6.Entity = M1.Entity
and    M1.Attribute = 'CustID'
and    M2.Attribute = 'Name'
and    M3.Attribute = 'Address'
and    M4.Attribute = 'City'
and    M5.Attribute = 'Cat'
and    M6.Attribute = 'Account'
order by CustID;
```

Script 5.5 - Generation of the contents of table CUSTOMER (SQL version 1)

Script 5.6, though a bit less intuitive at first glance, should be better, since it (hopefully) scans the table only once. It forms a group of triples for each entity. In each group, it extracts the highest value of each attribute. Among the triples of this group, only one has a true value for this attribute while all the others are assigned value `''`:

```
max(case when Attribute = 'Name' then Value else '' end)
```

Script 5.7, based on a simple scan of the source table, exhibits some interesting programming patterns. The rows are sorted by **Entity** values, in such a way that all the attribute rows of each entity are read consecutively. The **`extract`** statement starts the procedure by getting the first customer Id. Then, a loop parses all the rows of M_CUSTOMER. Variable `ce` denotes the row currently being built. The successive rows with the same Entity value form one row of the **TMP_CUSTOMER** table (a

clone of table CUSTOMER). The result of reading these rows is stored in variables `cus`, `nam`, `add`, `cit`, `cat` and `acc`, that are then used to insert the current TMP_CUSTOMER row.

When the loop terminates, these variables contain the data of the last row that has still to be inserted.

The algorithm of this script is the usual way that has been used for decades to create and process groups of records in sequential file processing. This approach is interesting from a historical point of view (it may also help to solve some special intricate problems) but it prevents the SQL engine to optimize the generation of triples.

.

```
insert into TMP_CUSTOMER(CustID,Name,Address,City,Cat,Account)
max(case when Attribute = 'CustID'
         then Value else '' end) as CustID,
max(case when Attribute = 'Name'
         then Value else '' end) as Name,
max(case when Attribute = 'Address'
         then Value else '' end) as Address,
max(case when Attribute = 'City'
         then Value else '' end) as City,
max(case when Attribute = 'Cat'
         then Value else null end) as Cat,
max(case when Attribute = 'Account'
         then cast(Value as decimal) else -9999.9 end) as
Account
from  M_CUSTOMER group by Entity;
```

Script 5.6 - Generation of the contents of table CUSTOMER (SQL version 2)

5.2.4 About performance

Figure 5.3 compares the execution time of each variant of the two conversion processes for increasing size of table CUSTOMER. These figures have been computed by the script **Schemaless2-Key-Value-Performance.sql** in directory **Case_Schemaless**.

– *Generating column-tables*: the pure SQL formula, that convert 32,768 source rows in 0.8 second, is much faster than the procedural technique that requires nearly one minute. However, the latter figure is not really significant. The procedure has been written in SQLfast, whose execution overhead is (much) higher than that of standard languages such as C, Java or even Python.

– *Rebuilding CUSTOMER table*: the multi-join SQL is the best technique to recover the source table: it converts about 200,000 triples into 32,768 source rows in less than 0.33 s. The SQL single scan is also quite good, with an execu-

tion time of less than 0.45 s. The procedural technique has soon lost the race, for the same reason we have cited above.

Nevertheless, our goal merely is to experiment the concepts of alternative models and not to develop high performance scripts for multi-terabyte databases!

```
extract ce = select Entity from M_CUSTOMER order by Entity;
for e,a,v = [select * from M_CUSTOMER order by Entity];
   if ('$e$' = '$ce$');
      if ('$a$' = 'CustID')  set cus = $v$;
      if ('$a$' = 'Name')    set nam = $v$;
      if ('$a$' = 'Address') set add = $v$;
      if ('$a$' = 'City')    set cit = $v$;
      if ('$a$' = 'Cat')     set cat = $v$;
      if ('$a$' = 'Account') set acc = $v$;
   else;
      insert into TMP_CUSTOMER values('§cus§','§nam§','§add§',
      '§cit§',case when '§cat§'='' then null else '§cat§' end,
       cast($acc$ as decimal));
      set ce = $e$;
      if ('$a$' = 'CustID')  set cus = $v$;
      if ('$a$' = 'Name')    set nam = $v$;
      if ('$a$' = 'Address') set add = $v$;
      if ('$a$' = 'City')    set cit = $v$;
      if ('$a$' = 'Cat')     set cat = $v$;
      if ('$a$' = 'Account') set acc = $v$;
   endif;
endfor;
insert into TMP_CUSTOMER values('§cus§','§nam§','§add§',
      '§cit§',case when '§cat§'='' then null else '§cat§' end,
       cast($acc$ as decimal));
```

Script 5.7 - Generation of the contents of table CUSTOMER (procedural version)

Size	16	256	1,024	4,096	32,768
Load M_CUSTOMER (SQL version)	1	4	23	81	829
Load M_CUSTOMER (procedural)	17	264	1,146	5,046	56,978
Rebuild CUSTOMER (SQL 1)	1	2	10	34	321
Rebuild CUSTOMER (SQL 2)	1	3	11	47	445
Rebuild CUSTOMER (Procedural)	307	5,395	23,313		

Figure 5.3 - Execution time, in ms., of the conversion processes applied to source CUSTOMER table for various number of rows

5.2.5 Application

The *Key-value* model allows the same queries to be formulated as those working on the source database. Scripts 5.8 to 5.9 are some examples.

```
select C1.Entity as CustID,C1.Value as Name,C2.Value as City
from   M_CUSTOMER C1, M_CUSTOMER C2
where  C1.Entity = C2.Entity
and    C1.Attribute = 'Name'
and    C2.Attribute = 'City'
order by C1.Entity;
```

Script 5.8 - Extracting data CustID, Name and City of the rows of table M_CUSTOMER

```
select Entity as CustID, Value as Name
from   M_CUSTOMER
where  Attribute = 'Name';
```

Script 5.9 - Extracting data CustID and Name from the rows of table M_CUSTOMER

However, new kinds of queries can be written, based on the fact that the values of different attributes are stored in the same column (Script 5.10) and on the fact that attribute names are pure values which can be queried as well (Script 5.11)

```
select Entity,Attribute,Value
from   M_CUSTOMER
where  Value like '%B1%'
order by Entity;
```

Script 5.10 - Which customers have a property value including characters *B1*?

```
select Entity,Attribute,Value
from   M_SHIPMENT
where  lower(Attribute) like '%date%'
order by Entity;
```

Script 5.11 - Extracting all the data of table M_SHIPMENT where some attribute names include the word **date** (whatever the case), such as *ShipmentDate* and *PaymentDate*

5.3 The Key-Value model - Version 2

In this variant of the *Key-Value* model, all the data of a database are stored in a **single table**, whatever the entity type they belong to. Thus, there is one table for the whole source database. The three columns **Entity**, **Attribute** and **Value** still are valid, but a fourth column, **EType** is necessary if we want to keep the notion of *entity type*. Considering the source ORDERS.db database, it is natural to call this unique table **ORDERS**. This table can be declared by Script 5.12.

```
create table ORDERS(
   EType      varchar(18) not null,
   Entity     varchar(15) not null,
   Attribute  varchar(32) not null,
   Value      varchar(64),
   primary key (EType,Entity,Attribute));
```

Script 5.12 - Table ORDERS hosts all the data of database ORDERS.

Figure 5.4 is an excerpt of the contents of table ORDERS. The entity ID of an entity is the primary key value of the source row.

EType	Entity	Attribute	Value
CUSTOMER	B112	CustID	B112
CUSTOMER	B112	Name	HANSENNE
CUSTOMER	B112	Address	23, r. Dumont
CUSTOMER	B112	City	Poitiers
CUSTOMER	B112	Cat	C1
CUSTOMER	B112	Account	1250
...	...	...	...
PRODUCT	CS262	ProdID	CS262
PRODUCT	CS262	Description	RAFT. PINE 200x6x2
PRODUCT	CS262	Price	75
PRODUCT	CS262	QonHand	45
...	...	...	...
CUSTORDER	30178	OrdID	30178
CUSTORDER	30178	CustID	K111
CUSTORDER	30178	DateOrd	2013-12-21
...	...	...	...
DETAIL	30178-CS464	OrdID	30178
DETAIL	30178-CS464	ProdID	CS464
DETAIL	30178-CS464	Qord	25
...	...	...	...

Figure 5.4 - Contents of table ORDERS

Rebuilding the CUSTOMER table from ORDERS is a problem similar to that addressed in Section 5.2.3. Script 5.13 derives from variant 2 of Script 5.6.

```
select
max(case when Attribute = 'CustID'
         then Value else '' end) as CustID,
max(case when Attribute = 'Name'
         then Value else '' end) as Name,
max(case when Attribute = 'Address'
         then Value else '' end) as Address,
max(case when Attribute = 'City'
         then Value else '' end) as City,
max(case when Attribute = 'Cat'
         then Value else null end) as Cat,
max(case when Attribute = 'Account'
         then cast(Value as decimal) else -9999.9 end)
         as Account
from   ORDERS
where  EType = 'CUSTOMER'
group by Entity;
```

Script 5.13 - Reconstruction of table CUSTOMER from table ORDERS (variant 2)

5.4 The Key-Value model - Version 3

We can keep the triple format by merging columns **EType** and **Entity**, in such a way that the values of this column are unique among all the entities of the application domain (Figure 5.5). In this approach, the type of an entity is just an additional attribute of the latter. For example, customer entity **B112** can be uniquely denoted by **CUSTOMER-B112**. Its type is '**CUSTOMER**', while its name is '**HANSENNE**'.

By using abstract entity Ids we can shrink column Entity and derive a more concise triple table. However, the generation of these entity Ids is a bit more complex than it was in the context of a single table. Here, all the entities of the database are assigned a *globally unique* Id, whatever their type. This can be done in several ways. One of them consist in creating an *Id dictionary* table in which, before inserting a new entity, we record its type and primary key (column EType and Entity as in Figure 5.4), so that a global abstract id, declared *autoincrement*, is automatically generated. The latter is then used to denote this entity in table ORDERS.

A trade-off between these entity identification techniques consists in coding both the entity type and the entity within its type. This requires a small dictionary table that assigns to each table name a numeric code and records the *last entity number* created for this table. Before inserting an entity, we extract the table code from this dictionary and we increment the last entity number. Figure 5.6 shows excerpts of table ORDERS where column Entity denotes entities with this coding technique.

Figure 5.7 suggests a graphical interpretation of the composition of CUSTOMER entity 01-00012.

Entity	Attribute	Value
CUSTOMER-B112	EType	CUSTOMER
CUSTOMER-B112	CustID	B112
CUSTOMER-B112	Name	HANSENNE
CUSTOMER-B112	Address	23, r. Dumont
CUSTOMER-B112	City	Poitiers
CUSTOMER-B112	Cat	C1
CUSTOMER-B112	Account	1250
...	...	...
PRODUCT-CS262	EType	PRODUCT
PRODUCT-CS262	ProdID	CS262
PRODUCT-CS262	Description	RAFT. PINE 200x6x2
PRODUCT-CS262	Price	75
PRODUCT-CS262	QonHand	45
...	...	...

Figure 5.5 - Contents of database ORDERS reduced to triples (1)

Entity	Attribute	Value
01-001	EType	CUSTOMER
01-001	CustID	B112
01-001	Name	HANSENNE
01-001	Address	23, r. Dumont
01-001	City	Poitiers
01-001	Cat	C1
01-001	Account	1250
...	...	...
04-001	EType	PRODUCT
04-001	ProdID	CS262
04-001	Description	RAFT. PINE 200x6x2
04-001	Price	75
04-001	QonHand	45
...	...	...

Figure 5.6 - Contents of database ORDERS reduced to triples (2)

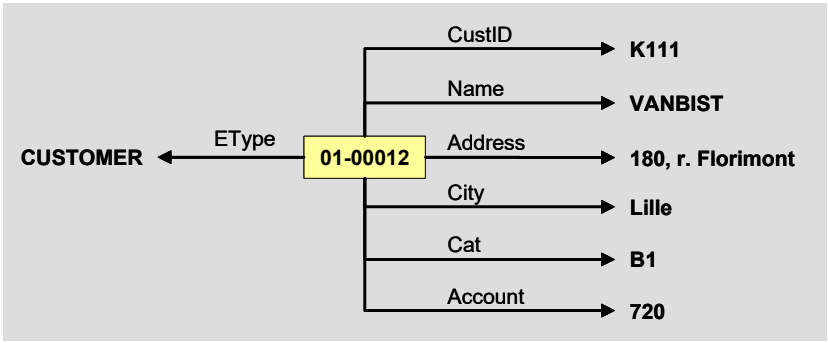


Figure 5.7 - Graphical representation of CUSTOMER entity 01-00012

It is important to observe that the entity Ids in column Entity, despite the way they are built, basically are **meaningless**. They could be replaced by arbitrary numbers or character strings without information loss.

The generation of table ORDERS from the tables of database ORDERS.db is left as an exercise.

This flexible way to represent facts about an application domain has been used to support several popular knowledge representation languages and systems. RDF (Resource Description Framework) is one of them.[2] It represents facts through a graph of resources. A resource is an *object* of any kind available locally or, more generally, on the web. Resources are identified by their URI (Universal resource identifier) or by a simple literal (number or character string). HTTP URL are a special kind of URI. Edges between two resources specifies that the first one (the *subject*) has, as *property*, the second one (the *object*).

RDF data generally are stored in triplestore[3], databases optimized to efficiently store, manage and process large triplestores (currently several trillions triples).[4] A graph query language, SPARQL, derived from SQL, is associated with this data model.

5.4.1 Application to Description Logic

Description Logic (DL) is a formal language derived from *First-order Logic* (FOL) and intended to precisely represent knowledge about a certain *universe* (what we have called the *application domain*) and to reason on it. DL exists in several variants, depending on the part of FOL they translate. DL is used to define facts about the universe, just like a database does. In addition, it allows general rules to be stated, either to specify those of these facts that are valid (kind of *integrity constraints*) and to infer new facts from those we already know.[5]

DL is the formal basis of ontology languages, in particular for the Semantic Web, such as DAML, OIL, OWL and, more recently OWL2.

Several syntax have been proposed and inference engines have been developed to define, store and process facts and general rules. The variant of DL we will talk about in this section is particular simple, since it addresses the representation of facts only. It comprises three constructs:

– *individual*: constant that is intended to denote a concrete or abstract thing of the universe. For instance, 01-00001 and Toulouse are individuals.

2. https://www.w3.org/RDF/
3. https://en.wikipedia.org/wiki/Triplestore
4. https://www.w3.org/wiki/LargeTripleStores
5. https://en.wikipedia.org/wiki/Description_logic [July 2017]; Baader, F., Horrocks, C., Sattler, U., *An Introduction to Description Logics*, Cambridge University Press, 2017

– *concept*: unary predicates that assigns a class to individuals. CUSTOMER(01-00001) asserts that individual 01-00001 belongs to class CUSTOMER and STRING(Toulouse) tells that individual Toulouse belongs to class STRING.

– *roles*: binary predicates used to assert a relationship between two individuals. City(01-00001,Toulouse) tells that Toulouse is the (name of) city of customer (denoted by) 01-00001.

The instantiation of concepts and roles with individuals forms a set of assertions that describe the universe. In the DL vocabulary, they are collectively called the A-BOX.

Actually, table ORDERS, that we have generated in this version of the Key-value model, contains everything necessary to generate the A-BOX of the DL description of our small universe comprising *customers*, *orders*, *details* and *products*. Generating the concept and role assertions requires two simple SQL queries (Scripts 5.14 and 5.15).

```
select distinct Value||'('''||Entity||''')' as Individuals
from   ORDERS
where  Attribute = 'EType';
```

Script 5.14 - Generating concept assertions from table ORDERS

```
select Attribute||'('||''''||Entity||''','''
       ||coalesce(doubleQuote(Value),'')||''')' as Roles
from   ORDERS
where  Attribute <> 'EType';
```

Script 5.15 - Generating role assertions from table ORDERS

Excerpts of their result sets is shown in Figures 5.8 and 5.9. Some comments on these queries:

– the individual notations are quoted to allow commas inside values

– consequently, internal quotes must be doubled. This is the role of UDF function **doubleQuote**.

– **coalesce** is a standard SQL function that returns the first non null value within its argument list.

```
CUSTOMER('01-00001')
CUSTOMER('01-00002')
CUSTOMER('01-00003')
...
PRODUCT('02-00001')
PRODUCT('02-00002')
...
CUSTORDER('03-00001')
CUSTORDER('03-00002')
```

```
...
DETAIL('04-00001')
DETAIL('04-00002')
...
```

Figure 5.8 - Excerpts of the assertions of the A-BOX (the individuals and the concepts)

```
CustID('01-00001','B062')
Name('01-00001','GOFFIN')
Address('01-00001','72, r. de la Gare')
City('01-00001','Namur')
Cat('01-00001','B2')
Account('01-00001','-3200')
...
CustID('01-00010','F011')
Name('01-00010','PONCELET')
Address('01-00010','17, Clôs des Erables')
City('01-00010','Toulouse')
Cat('01-00010','B2')
Account('01-00010','0')
...
OrdID('03-00005','30185')
CustID('03-00005','F011')
DateOrd('03-00005','2016-01-02')
...
OrdID('04-00007','30185')
ProdID('04-00007','CS464')
Qord('04-00007','260')

OrdID('04-00008','30185')
ProdID('04-00008','PA60')
Qord('04-00008','15')

OrdID('04-00009','30185')
ProdID('04-00009','PS222')
Qord('04-00009','600')
...
ProdID('02-00005','PA60')
Description('02-00005','NAILS STEEL 60 (1K)')
Price('02-00005','95')
QonHand('02-00005','134')
...
ProdID('02-00007','PS222')
Description('02-00007','PL. PINE 200x20x2')
Price('02-00007','185')
QonHand('02-00007','1220')
...
ProdID('02-00003','CS464')
Description('02-00003','RAFT. PINE 400x6x4')
Price('02-00003','220')
QonHand('02-00003','450')
...
```

Figure 5.9 - Excerpts of the assertions of the A-BOX (the roles)

In a relational database, a relationship between two entities is expressed by a *foreign keys* between the rows that describe these entities. If the *foreign key value* of the

source row is equal to the *primary key value* of the target row, then a relationship exists between these rows, and, as a consequence, between the entities described by these rows. This complicated way to state inter-entity relationship becomes much more simple in Description logic.

For example, the fact that the **order** denoted by **`x`** (whatever its CustID value) has been placed by the **customer** denoted by **`y`** (whatever their CustID value) will be simply stated by the assertion:

```
placed_by(x,y)
```

In this representation, the assertions that represent the columns of foreign keys become useless and can be discarded.

Figure 5.10 collects the assertions that express the relationships between orders and customers. The first expression tells us that the order denoted by `'03-00001'` has been placed by the customer denoted by `'01-00012'`.

```
placed_by('03-00001','01-00012')
placed_by('03-00002','01-00007')
placed_by('03-00003','01-00015')
placed_by('03-00004','01-00007')
placed_by('03-00005','01-00010')
placed_by('03-00006','01-00007')
placed_by('03-00007','01-00004')
```

Figure 5.10 - Derivation of *placed_by* role assertions

Actually, `placed_by` assertions can be derived from the assertions of Figure 5.9 through a conversion process illustrated in Figure 5.11.

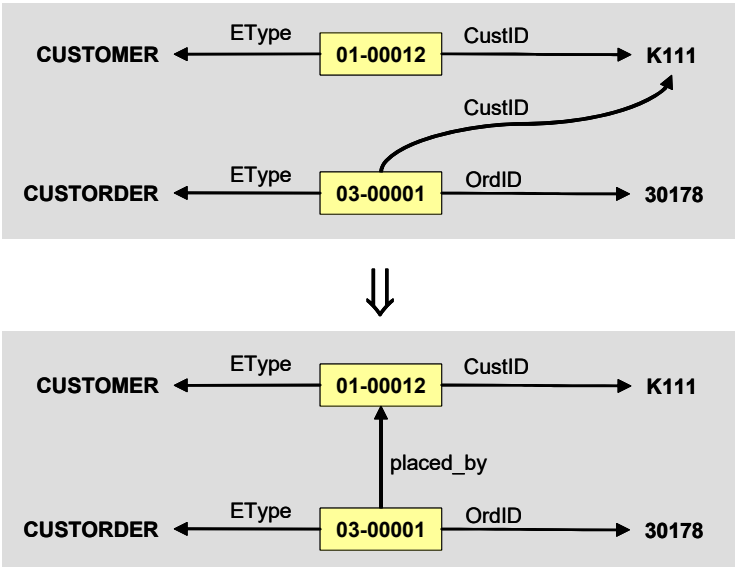


Figure 5.11 - Production of assertions of role *placed_by*

Since the description of the rule language of DL is outside the scope of this study, we will express this derivation rule as an equivalent[6] SQL query on the triples of Figure 5.5.

To help building this query, we will use the example of order 30178 that has been placed by customer K111. The triples describing these two entities are reminded in Figure 5.12. The four triples that will be used are marked with tags `O1`, `O2`, `O3`, `O4`. The reasoning is as follows:

- the order is identified by three properties:
  - it is of type CUSTORDER [`O1.Attribute = 'EType' and O1.Value = 'CUSTORDER'`]
  - the value of its attribute CustID is **V** [`O2.Attribute = 'CustID'`]
  - its entity Id is **W** in the two triples [`O2.Entity = O1.Entity`]
- the customer is identified by three properties:
  - it is of type CUSTOMER [`O3.Attribute = 'EType' and O3.Value = 'CUSTOMER'`]
  - the value of its attribute CustID is **X** [`O4.Attribute = 'CustID'`]
  - its entity Id is **Y** in the two triples [`O4.Entity = O3.Entity`]
- the link between them is expressed by property:
  - **V = X** [`O2.Value = O4.Value`]
- the role assertion can then be built:
  - `placed_by(W,Y)` [`O1.Entity, O3.Entity`]

The derivation query is shown in Script 5.16.

Entity	Attribute	Value	
01-00012	**EType**	**CUSTOMER**	**O3**
01-00012	**CustID**	**K111**	**O4**
01-00012	Name	VANBIST	
01-00012	Address	180, r. Florimont	
01-00012	City	Lille	
01-00012	Cat	B1	
01-00012	Account	720	
...	...	...	
03-00001	**EType**	**CUSTORDER**	**O1**
03-00001	OrdID	30178	
03-00001	**CustID**	**K111**	**O2**
03-00001	DateOrd	2015-12-21	
...	...	...	

Figure 5.12 - Two sets of roles expressing two entities and their relationship

6. Triples and role assertions are equivalent and both SQL and DL are based on FOL

```
select 'placed_by('''||O1.Entity||''','''||O3.Entity||''')'
from  ORDERS O1, ORDERS O2, ORDERS O3, ORDERS O4
where O1.Attribute = 'EType'  and O1.Value = 'CUSTORDER'
and   O2.Attribute = 'CustID' and O2.Entity = O1.Entity
and   O3.Attribute = 'EType'  and O3.Value = 'CUSTOMER'
and   O4.Attribute = 'CustID' and O4.Entity = O3.Entity
and   O2.Value = O4.Value;
```

Script 5.16 - SQL query that create *placed_by* assertions

Finally, what are the differences between triples, RDF, OWL and DL?

All this may seem a bit complicated. Let us try to clarify (a little bit) the relationship between these concepts.[7]

– *Triple* is a data format. A *triplestore* is a specialized database in which triples can be stored and retrieved efficiently.

– *RDF* is a logical model and languages (including *SPARQL*) suited to interpret triples as pieces of knowledge independently of the underlying storage technology. Just like the relational model (including SQL) interprets the records stored in the files of a database.

– *OWL* can be seen as an extension of RDF based on the concept of *ontology*.[8]

– *Description logics* is a mathematical theory derived from the FOL, that gives RDF and OWL a sound formal interpretation. Just like the theoretical relational model of E. F. Codd gives RDBMS a sound basis to interpret the result of any query. In particular, DL provides mechanism to define inference rules.

5.5 Not all Key-value models are created equal

Some popular NoSQL DBMS are said to belong to the *Key-value* data manager family. Once again, this name is completely misleading. *Key-value* DBMS propose to organize data according to a low level model in which a unique key (the *Key*) is associated to (generally) unstructured chunks of data (the *Value*). These data can be retrieved by their *Key* value. Point!

7. A quite interesting discussion on this topic is available at https://stackoverflow.com/questions/1740341/what-is-the-difference-between-rdf-and-owl.

8. An *ontology* is a structured collection of terms that describe the *things* of a certain application domain in terms of entities, classes of entities and relationships between them. As compared with databases, an ontology is, to some extent, similar to the conceptual schema of a database (though an ontology may include instances as well).

Practically, The schema of any *Key-value* database could be simulated by a relational table:

```
create table RECORD(Key varchar(2048) not null primary key,
                    Value blob);
```

The structure and the meaning of the *Key* and *Value* parts of records are up to the application. There is no query language. The data manipulation primitives are reduced to **get**, **put** and **delete**. A *Key-value* database is just an abstract repository in which data of arbitrary complexity can be stored and retrieved by a unique key. The main role of these DBMS is the efficient management of large volumes of data massively distributed and replicated among many data servers. *Redis* and *Berkeley DB* are among the most popular.

Quite often, a *Key-value* data manager is the bottom layer of more sophisticated manager, such as the NoSQL DBMS provided by Oracle, that comprises *Berkeley DB* topped by a JSON upper layer.

5.6 The scripts

The scripts of the *Key-value* models are available in directory **SQLfast/Scripts/Case-Studies/Case_Schemaless**. They can be run from main script **Schemaless-MAIN.sql** (see Part 3)

Case study **6**

Schema-less databases - Part 3

Objective: This document studies a fourth family of alternative data models, namely the *object* (or *document)* models. In these models, the information is represented by complex objects in which properties can be multivalued and composite.

Keywords: non-relational data model, key-value model, object model, NoSQL, schema-less database, multivalued property, composite property, document-oriented DBMS, MongoDB, CouchDB, Azure, Datastore Oracle, metadata, index, data migration, schema conversion

6.1 Object view of Key-Value data

At the opposite of the *Key-Value* view of the data, the *object view* aggregates all the data of each entity as a complex object associated with this entity. If we imagine these data stored in a table, then each entity is described by a single (but complex) row.

An interesting property of the *Key-Value* model developed in Part 2 (that comprises table M_CUSTOMER for example) is that one can easily use it derive this object[1] view. With the object model, application programs read **complex records** each comprising an entity Id (represented by, say, column **Entity**) and a list of *Key-Value* couples represented by column **Attributes** (plural!).

Figure 6.1 shows some excerpts from a table in which the data of source table CUSTOMER are structured as objects. This table has been generated by Script 6.1 from *Key-Value* table M_CUSTOMER.

Entity	Attributes
B062	{"Account":"-3200.0","Address":"72, r. de la Gare","Cat":"B2", "City":"Namur","CustID":"B062","Name":"GOFFIN"}
B112	{"Account":"1250.0","Address":"23, r. Dumont","Cat":"C1", "City":"Poitiers","CustID":"B112","Name":"HANSENNE"}
...	...
C123	{"Account":"-2300.0","Address":"25, r. Lemaître","Cat":"C1", "City":"Namur","CustID":"C123","Name":"MERCIER"}
C400	{"Account":"350.0","Address":"65, r. du Tertre","Cat":"B2", "City":"Poitiers","CustID":"C400","Name":"FERARD"}
D063	{"Account":"-2250.0","Address":"201, bvd du Nord", "City":"Toulouse","CustID":"D063","Name":"MERCIER"}
...	...
K111	{"Account":"720.0","Address":"180, r. Florimont","Cat":"B1", "City":"Lille","CustID":"K111","Name":"VANBIST"}
K729	{"Account":"0.0","Address":"40, r. Bransart", "City":"Toulouse","CustID":"K729","Name":"NEUMAN"}
L422	{"Account":"0.0","Address":"60, r. de Wépion","Cat":"C1", "City":"Namur","CustID":"L422","Name":"FRANCK"}
...	...

Figure 6.1 - Object view of Key-Value data (JSON format)

```
select Entity,
       '{'
       ||group_concat('"'||Attribute||'":"'||Value||'"',',')
       ||'}' as Attributes
from   M_CUSTOMER
group by Entity;
```

Script 6.1 - Generating an object view of Key-Value data (JSON format)

The list of name:value pairs can be built according to a proprietary or standard format such as Key-Value, CSV, XML or JSON (*JavaScript Object Notation*), the format chosen in this example.

We notice that attribute **Cat** does not appear when its value is *null* (in entities D063 and K729). This is consistent with a model according to which all (and only) the information known about the entity is available in its record.

We also observe that the value of attribute **Account** is double-quoted, which is useless since it is a number. This can be fixed easily with Script 6.2.

1. The term *object* refers to the concepts of *Object-oriented* data models, namely the ability to build and manage complex data structures, as compared with the flat structure of standard relational tables. These data aggregates perfectly fit to (true) objects in Java programs for instance, thus reducing the so-called *impedance mismatch* between databases and OO programs without resorting to *Object/Relational Mapping* (O/RM) middleware.

```
select Entity,
       '{'||group_concat('"'||Attribute||'":'
                         ||case when Attribute in ('Account')
                                then Value
                                else '"'||Value||'"'
                           end, ',')
    ||'}' as Attributes
from M_CUSTOMER
group by Entity
```

Script 6.2 - Generating an object view of Key-Value data (JSON format); fixing the quoted representation of *numeric* values.

6.2 Objects with complex properties

So far, the attributes of the objects we have built have elementary values. Now we will examine how to represent and compute complex properties, the instances of which are multivalued, composite or both.

6.2.1 Multivalued properties

We want to add to customer data the list of *amounts* of the orders they have placed. Of course, these amounts can be computed from the data available in the source tables, but here, we want to store them as attributes in table M_CUSTOMER, just to illustrate the point. Script 6.3 will insert these new rows. Of course, customers who have not placed any order also have no Amount attribute.

```
insert into M_CUSTOMER(Entity,Attribute,Value)
select CustID,'Amount',sum(Qord * Price)
from   CUSTORDER O, DETAIL as D, PRODUCT as P
where  O.OrdID = D.OrdID and D.ProdID = P.ProdID
group by CustID,O.OrdID;
```

Script 6.3 - Adding multivalued attribute *Amount* to table M_CUSTOMER

Not surprisingly, this query fails ("*SQL execution error (columns Entity, Attribute are not unique)*"). Indeed, the primary key declared for M_CUSTOMER prevents us from storing more than one attribute with a given name, which make customer C400 uncomfortable, with her three orders! To allow multiple attributes with the same name, we must drop this constraint[2] (Script 6.4). Now, the script works fine (Figure 6.2).

```
create table M_CUSTOMER(
   Entity    varchar(32)   not null,
   Attribute varchar(32)   not null,
   Value     varchar(32)
   primary key (Entity,Attribute));
```

Script 6.4 - Modifying the structure of table M_CUSTOMER to let it accommodate multivalued attributes

Entity	Attribute	Value
...	...	...
C400	CustID	C400
C400	Name	FERARD
C400	Address	65, r. du Tertre
C400	City	Poitiers
C400	Cat	B2
C400	Account	350
C400	**Amount**	**6400**
C400	**Amount**	**28500**
C400	**Amount**	**315**
F011	CustID	F011
F011	Name	PONCELET
F011	Address	17, Clôs des Erables
F011	City	Toulouse
F011	Cat	B2
F011	Account	0
F011	Amount	169625
...	...	...

Figure 6.2 - A customer can have more than one attribute of the same type

It would be nice if we could produce JSON objects in which, for each customer, the amount values are presented as an **array**, as in Figure 6.3.

```
{"Account":"350", "Address":"65, r. du Tertre", "Cat":"B2",
 "City":"Poitiers", "CustID":"C400", "Name":"FERARD",
 "Amounts":[6400,28500,315]}
```

Figure 6.3 - A complex JSON object with multivalued attribute **Amount**

This is quite possible in SQL, but it is a bit more complex than the generation of flat objects as those in Figure 6.1, because we have to combine two levels of aggregation, the highest for entity grouping and the innermost for array grouping.

2. We could have extended the primary key by adding to it column `Value`. However this would have been valid only for distinct values of this attribute, which is clearly not the case in general for attribute Amount.

The simplest way to proceed is to preprocess the data in such a way that entities have single-valued attributes only, so that standard Script 6.2 applies. This is performed by a **view** (Script 6.5) that first defines new attribute **Amounts** through a `group_concat` function, then adds all the other attributes. Figure 6.4 shows the contents of this view.

Entity	**Attribute**	**Value**
...	...	...
C400	CustID	C400
C400	Name	FERARD
C400	Address	65, r. du Tertre
C400	City	Poitiers
C400	Cat	B2
C400	Account	350
C400	**Amounts**	**[6400,28500,315]**
...	...	...

Figure 6.4 - Collecting the multiple values of **Amount** in an array

```
create temp view CUST_AMOUNT(Entity,Attribute,Value) as
select Entity,'Amounts','['||group_concat(Value,',')||']'
from   M_CUSTOMER
where  Attribute = 'Amount'
group by Entity
  union
select Entity,Attribute,Value
from   M_CUSTOMER
where  Attribute <> 'Amount';
```

Script 6.5 - Transforming sequences of **Amount** values in array **Amounts**

Script 6.6 then extracts the 2-level objects. It is an extension of Script 6.2 in which numeric values are left unquoted.

6.2.2 Composite multivalued properties

Let us now try to generate even more complex objects, such as that of Figure 6.5. In object C400, besides elementary attributes **CustID**, **Name** and **City**, we have introduced attribute **Orders**, which itself is an *array of objects*, each one containing the data of an order of the current customer. This attribute is both *multivalued* and *composite*. It results from the integration of **CUSTORDER** data in the object representing a customer.

```
select Entity,
       '{'
       ||group_concat('"'
                 ||Attribute
                 ||'":'
                 ||case when Attribute in ('Account','Amounts')
                        then Value
                        else '"'||Value||'"'
                   end,
                 ',')
       ||'}' as Attributes
from  CUST_AMOUNT
group by Entity;
```

Script 6.6 - Producing complex JSON objects including an array

```
{"CustID":"C400","Name":"FERARD","City":"Poitiers",
 "Orders":[{"OrdID":"30179","DateOrd":"2015-12-22"},
           {"OrdID":"30184","DateOrd":"2015-12-23"},
           {"OrdID":"30186","DateOrd":"2016-01-02"}]
}
```

Figure 6.5 - A JSON object comprising an array of composite objects

We could extract the information from Key-value table M_CUSTORDER but is easier to start from source table CUSTORDER. Script 6.7 defines view **CUST_ORDERS** that returns the data of Figure 6.6.

Entity	Attribute	Value
B512	Orders	[{"OrdID":"30188","DateOrd":"2016-01-03"}]
C400	Orders	[{"OrdID":"30179","DateOrd":"2015-12-22"},
		{"OrdID":"30184","DateOrd":"2015-12-23"},
		{"OrdID":"30186","DateOrd":"2016-01-02"}]
F011	Orders	[{"OrdID":"30185","DateOrd":"2016-01-02"}]
K111	Orders	[{"OrdID":"30178","DateOrd":"2015-12-21"}]
S127	Orders	[{"OrdID":"30182","DateOrd":"2015-12-23"}]

Figure 6.6 - Build the values of attribute Orders as an array of JSON objects

```
create temp view CUST_ORDERS(Entity,Attribute,Value) as
select CustID,'Orders',
     '['||group_concat('{'
        ||'"OrdID":"'||OrdID||'","DateOrd":"'||DateOrd||'"'
        ||'}', ',')||']'
from   CUSTORDER
group by CustID;
```

Script 6.7 - Creating attribute **Orders** for each customer entity (customers who have not placed orders are not represented)

To build the objects of Figure 6.5, we must slightly adapt the generation query so that it works on the *union* of **M_CUSTOMER** and **CUST_ORDERS** (Script 6.8).

```
select Entity,
       '{'
       ||group_concat('"'
                      ||Attribute
                      ||'":'
                      ||case when Attribute in ('Orders','Account')
                             then Value
                             else '"'||Value||'"'
                        end,
                      ',')
       ||'}' as Attributes
from  (select Entity,Attribute,Value
       from M_CUSTOMER
         union
       select Entity,Attribute,Value
       from CUST_ORDERS)
where Entity in ('C400','B062','B512')
group by Entity;
```

Script 6.8 - Producing complex JSON objects including an array of objects

6.3 Indexing objects

Let us consider that the JSON objects we have created above are stored in table **CUSTOMER_JSON** with columns **Entity** and **Attributes**. We also suppose that there are millions of rows in this table. Remembering how indexing a relational table is easy through **`create index`** statements, we could worry about how we could index complex, multivalued and/or composite, attributes such as those stored in column **Attributes**. For instance, if we want to retrieve data on the customers who *live in*

Poitiers or who have *placed an order on 2016-01-02*, we have so far no straightforward way to get them but by selecting the objects through, say, a **like** predicate. The corresponding query is simple but particularly inefficient, since it requires a full scan of the table:

```
select *
from   CUSTOMER_JSON
where  Attributes like '%"City":"Poitiers"%';
```

Actually, indexing such structures is much easier than we could think. Let us consider table **M_CUSTOMER** again. As is usual in most DBMS, an index is automatically created on its primary key. We decide to create a second index on columns (**Attribute**, **Value**) to get a fast access to all the entities from their attribute values:

```
create index XATTVAL on M_CUSTOMER(Attribute,Value);
```

As a consequence, the following query will return the set of desired **Entity** values corresponding to customers living in Poitiers in a matter of tens of milliseconds:

```
select Entity
from   M_CUSTOMER
where  Attribute = 'City'
and    Value = 'Poitiers';
```

Better, if we are only interested in indexing data on attribute City, we can create a *partial (*or *filtered) index*, which is much smaller and therefore more efficient

```
create index XCITYVAL on M_CUSTOMER(Value)
where Attribute = 'City';
```

We then write the query of Script 6.9, that returns the attribute values of these entities in JSON format.

```
select Entity,Attributes
from   CUSTOMER_JSON
where  Entity in (select Entity
                  from   M_CUSTOMER
                  where  Attribute = 'City'
                  and    Value = 'Poitiers');
```

Script 6.9 - Which customers live in Poitiers?

6.4 Comparison of the data models

We have described a series of alternative data models that offer more generic data structures than the standard relational model. By *generic*, we mean that some of the data structures described by the schema are not specific to the application domain the database is devoted to (e.g., columns Attribute and Value, which are domain independent). To better understand this concept, let us identify the objects of a standard relational schema the existence or the name of which is specific to the application domain, therefore providing information on this domain:

- **database**: the name of the database usually suggests the nature of the application domain (e.g., **ORDERS.db**)
- **table**: each table of a normalized database contains the data of a set of entities of the same type. The name of the table should be close to that of this entity type.
- **column**: each column designates a property of the entities described by its table. Here again, the name of the column should suggest that of this property.
- **data type**: some data types give hints on the meaning of the values of a column (e.g., date, character, integer and the user-defined domains)
- **unique key**: defines an integrity constraint that translates an important rule of the application domain (just think of the primary key (OrdID,ProdID) of table DETAIL).
- **foreign key**: generally defines relationships between rows, and, therefore, between entities.

Actually, the databases implemented in each of the alternate data models express (almost) the same types of facts of the application domain. They differ on how and where these fact types are specified: in the schema or in the data. For example, in the *Key-Value* model, *version 1*, the table name identifies an entity type but its attributes are defined by the value set of column **Attribute**.

This comparison is summarized in Table 6.1 below. The answer is **S** if the schema provides the information, **D** if the information is given by the data, **s** if the information is incompletely specified in the schema, **d** if the information is incompletely specified in the data and ∅ if the information is not specified.

Considering the standard relational model as the reference, this table evaluates to what extent the information contents of the schema of each alternative model is preserved (we ignore the database name, which is present in each model). Another way of describing this analysis could be: *what do I learn on the application domain when I read the schema*? And, as a consequence, *do the data include the missing information*?

Clearly, the schema of alternate data models is less rich and less expressive than the relational model. This weakness is compensated by a greater flexibility: new entity types and new attributes can be dynamically added without modifying the schema.

Table 6.1 - Data/metadata distribution in various data models

	Entity type	Property	Datatype	Unique key	Relationship
Relational	**S**	**S**	**S**	**S**	**S**
Universal table	**D**	**S**	**S**	∅	∅
Column tables	**s**[a]	**S**	**S**	**s**[b]	**s**[c]
Key-value 1	**S**	**D**	∅	∅	∅
Object-oriented	**S**	**D, s**[d]	∅	∅	∅
Key-value 2	**D**	**D**	∅	∅	∅

a.Maybe, through some parts of the name of the column table
b.Possible for single-component unique keys only
c.Possible for single-component foreign keys only
d.Maybe, through the name of the aggregate

6.5 For once, most *Object data models* are created equal

The data structures described in this study have been adopted by the most recent (and most popular) NoSQL DBMS, often under the family name *Document-oriented* DBMS. Indeed, hierarchical structures of arbitrary complexity are typical of documents and forms. This model nicely mimics the organization of a document, much more than flat relational structures, in which the components of documents are scattered among several tables. As an example, the data of customer orders are distributed in tables CUSTORDER and DETAIL of database ORDERS.db, while, in the object models, they are assembled into significant aggregates.

Among the most representative document-oriented DBMS we can mention MongoDB (the most popular in 2017) but also CouchDB, Azure, Datastore and Oracle NoSQL.

Interestingly, these hierarchical data structures which are at the basis of these modern DBMS, are very ancient. First, they are part of to most standard programming languages, such as COBOL (in which so-called *record structures* are particularly sophisticated), Pascal, C and, to some extent, Python.

A well known proverb teaches us that there is no free lunch. While the ultimate desirable form of relational databases comprises normalized tables[3], that are (1) devoid from any internal redundancies and (2) neutral *wrt* data application requirements, object databases generally include duplicated data and often are dedicated to a specific application (and therefore less easily reusable). In addition, the manipula-

3. The concept of relational normalization will be studied in a further case study.

tion languages of object data structures are more complex and less powerful than SQL.

6.6 The scripts

The scripts of the *Object* models as well as those of the models studied in Parts 1 and 2 are available in directory **SQLfast/Scripts/Case-Studies/Case_Schemaless**. They can be run from main script **Schemaless-MAIN.sql** (Figure 6.7).

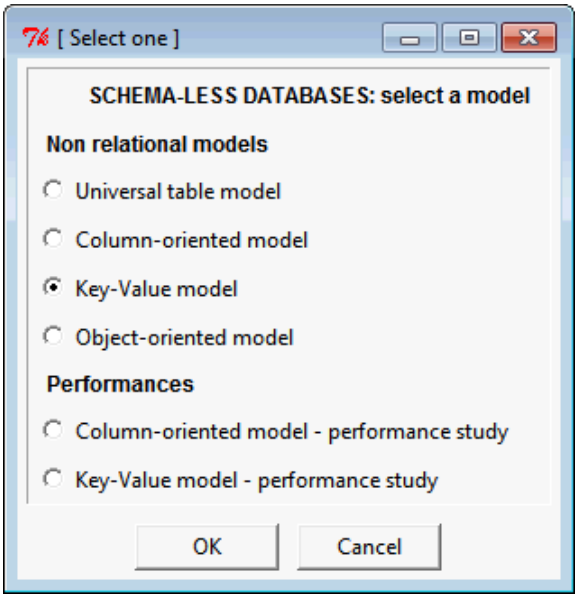


Figure 6.7 - Selecting a data model and experimenting with performances

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

Case study 8

Active databases

This study shows how advanced data structures of SQL can be used to built smarter databases, in particular *active databases*, that are able to react to external stimuli. It starts with a short reminder of the SQL data structures, including some of these advanced constructs, namely check constraints, views, generated columns and triggers. Then, it presents some usual applications of active databases, such as integrity control, redundancy management, updatable views, data modification logging, alerters, type-subtype implementation, repair rules, temporal databases and access control. Finally, it explores, through a simple but representative business application, the power of active databases as compared with traditional application development. In this application, we observe the impact of moving the control of business rules from the programs to the database. This unusual application architecture requires special static and dynamic validation techniques. In its conclusion, this study briefly sketches the history of the trigger concept and evaluates the benefits and disadvantages of its use in application development.

Keywords. ECA rules, trigger, business rules, active database, DAG, finding circuits, advanced SQL, data structure, check predicate, updatable view, derived data, inventory management, 2-tier architecture, 3-tier architecture

Table of content

8.1 Introduction
8.2 The SQL data structures
- 8.2.1 Static structures
- 8.2.2 Basic constraints
- 8.2.3 Predicates (check constraints)
- 8.2.4 Derived data
- 8.2.5 Stored procedures
- 8.2.6 Triggers
- 8.2.7 Coping with the limitations of the trigger language

8.3 Basic applications
8.4 Integrity management
8.5 Non standard system behavior
8.6 Updatable view
8.7 Redundancy management and derived data
8.8 Data modification journaling
8.9 Alerters
8.10 Type-subtype implementation
- 8.10.1 About database design
- 8.10.2 The Type-subtype concept
- 8.10.3 Is-a relations in SQL3
- 8.10.4 Is-a relations in SQL2
- 8.10.5 The insert operations
- 8.10.6 TThe update operations
- 8.10.7 The delete operations
- 8.10.8 Automating type-subtype manager production

8.11 Other applications
- 8.11.1 Repair rules
- 8.11.2 Access control
- 8.11.3 Temporal databases

8.12 Building a business application
- 8.12.1 About the case
- 8.12.2 About the case study

8.13 The static structure of the database
8.13.1 The base tables
8.13.2 The transaction tables
8.13.3 The initial state of the database
8.13.4 The transactions
8.14 Implementation of the classical application architecture
8.15 Implementation as an active database
8.16 Extending the data structures
8.17 The user interface
8.18 The transactions
8.18.1 Registering a customer order
8.18.2 Executing a supplier order
8.18.3 Registering the payment of an invoice
8.18.4 Event architecture of the active database
8.19 Verification of an active database
8.19.1 Static analysis: checking trigger circuits
8.19.2 Dynamic analysis: tracing the execution of the active database
About the concept of *process level* in trigger tracing
Event-based tracing
Operation-based tracing
8.19.3 Replaying transactions
8.20 Limits of this case study
8.21 Conclusions, history and extensions
8.21.1 From deductive databases to triggers
8.21.2 Recursive programming
8.21.3 Modeling and design
8.21.4 Finally, what may active databases be good for?
8.21.5 Reference
8.22 The implementation
Appendix A - The initial state of the database
Appendix B - RETAIL application: task analysis
Appendix C - Circuit detection in directed graphs

8.1 Introduction

The main goal of a database is to store data that describe the objects and the activities of a part of the real world, generally called its *application domain*. In addition to the basic constructs through which this static view of the data is implemented (mainly tables, columns and keys), relational DBMS also include more sophisticated mechanisms that allow a database to better translate the facts and the rules of the application domain. Check constraints, generated columns, views (notably updatable ones), stored SQL procedures and triggers are the most important of them. The latter are particularly interesting in that they are the corner stone of so-called *active databases*, that are designed to automatically react to external stimuli.

This study is intended to show how these data structures can be used to built smarter databases. It starts with a short reminder of the SQL data structures, including some of the most advanced constructs. Then, it presents some usual applications of active databases. Finally, it explores, through a simple but complete business application, the power of active databases as compared with traditional application development.

8.2 The SQL data structures

We know that the SQL language offers a rich set of constructs allowing us to create databases that constitute reliable models of data oriented problems and of their solutions. These constructs range from simple static structures, such as tables and columns, to more sophisticated objects like predicates, views, stored procedures and triggers. They all form the basic toolkit to create and manage active databases.

Let us briefly recall and describe them.

8.2.1 Static structures

Let us call *static structures* of a database the constructs that allow us to store data, that is, the basic *data containers*. We recall their main constituents and their properties:

– A *database* comprises tables.

– A *table* is structured into one or more columns. Each *column* is assigned a *data type*, such as numeric, character string, date, blob and some others.

– Each *data row* is stored in a table. For each column of this table, each row comprises one value of the column data type (or *null*, as will be reminded below).

– Among the columns of each table, a subset can be declared its *primary key*. No two rows of the table may include the same values of the primary key. The role of this key generally is to uniquely identify the rows of the table.

– Among the columns of each table, subsets can be declared *foreign keys*. Each value of a foreign key references a row in another table (or in its table itself).

8.2.2 Basic constraints

Three basic constraints can be specified on the static structures:

– *Mandatory column*: if a column is declared *not null*, then each row must comprise a value for this column. Symbolically, we will say that it cannot take the *null* value.

– *Uniqueness constraint*: the rows of a table are distinct. This property can be ensured for a subset of the columns. The *primary key* is one way to assert a uniqueness property[1]. If other subsets enjoy this property as well, they will be declared through *unique keys*.

– *Referential constraint*: this constraint applies to a subset of column of a source table *wrt* a target table. It states that, for any row of the source table, the values of these columns also are those of a row of the target table. The preferred way to ensure this constraint is to declare this subset of columns a *foreign key.*[2]

8.2.3 Predicates (check constraints)

The declaration of a table can include predicates, in the form of *check* constraints. A *check* constraint comprises an SQL condition that must be satisfied (more precisely *not evaluated to False*, which is a bit different) by each row of its table. In most DBMS, this condition may only apply to values of the current row (or system registers, such as *current_date*), without reference to other rows or to other tables.

8.2.4 Derived data

Tables and columns constitute the basic data structures. From them, we can define new tables (called *views*) and columns (called *generated columns*) the contents of which derive through SQL or computation formulae.

– *Views*: A view is a virtual table of which only the definition, in the form of a `select` query, is stored in the database. When a view is queried, its contents is computed through this query, or at least the relevant part of this contents. Under certain conditions the data of a view can be modified. i.e., adding, deleting and modifying rows. In this case, the DBMS propagate the requested modifications on the real data. Generally, a view can be used as a real table.

1. Though primary and unique keys are the preferred ways to ensure uniqueness, they are not the only techniques. Uniqueness can also be verified by triggers.
2. In the same way, there are other techniques to ensure referential constraints, for example through triggers, as in earlier versions of RDBMS.

– *Generated columns*: A *generated* - or *computed* - *column* is a column the values of which are computed by a formula from the values of other columns of its row (or system registers, such as *current_date*). These values can be *virtual*, in which case they are computed each time they are requested, or *stored* (or *persistent*) in the database.

8.2.5 Stored procedures

An SQL stored procedure is a named sequence of SQL statements that is stored in the database. It translates a data manipulation function shared by a community of users. A stored procedure can be called by applications programs, by triggers and by other stored procedures. If the concept of stored procedure is not available in the DBMS, it can be simulated by triggers or UDF.

8.2.6 Triggers

SQL triggers are a concrete implementation of *Event-Condition-Action* rules (aka ECA). They read: when this *Event* occurs and if this *Condition* is satisfied, then execute this *Action*. An SQL trigger is a named procedure (the Action), also stored in the database as a sequence of SQL statements, that will be automatically executed as soon as some events occur (Event), provided the Condition is met. These events comprise data modification actions: `insert`, `update` and `delete`, though some DBMS also allow other events such as opening and closing the database, DDL operations, temporal events or abstract events created by the programs.

The complete definition of a trigger, in addition to its name, generally comprises these components:

– the table or the view it is attached to

– the event(s) that trigger it: *insert*, *delete*, *update* (of any or certain columns)

– when it is executed with respect to the event (*before*, *after* or *instead of* the action). An *instead of* trigger is generally defined on a view, explaining what to do when a modification action is asked for execution.

– its *scope* or *granularity*: once for the action (*for each statement*) or for each row affected (*for each row*)

– an optional *condition* that must be met in order for the body to be executed

– its *body*, which is a sequence of SQL statements. However, many DBMS provide a richer language including *variables* and control statements such as *if-then-else* and *loops*. An important statement is the one that *cancels* or *aborts* the source event and *raises* an exception that can be caught by the application program.

– the *execution order* when more than one trigger have been defined for the same event on the same table.

Many DBMS provide a subset only of these components.[3] Triggers probably are the most powerful feature of relational DBMS. However, they are particularly delicate to use and to debug. Indeed, they rely on three different programming paradigms, namely *event-based*, *logic-based* (through the `when` and the `where` clauses) and *procedural* (within the body part). In addition, they have strong links with the transaction management policy of the DBMS.

As we will learn, the trigger mechanism is at the heart of active databases.

8.2.7 Coping with the limitations of the trigger language

The SQL-3 standard defines the body of a trigger as a *sequence of SQL statements*.[4] However, in most DBMS, the body of a trigger is a procedure written in a general-purpose language comprising, besides SQL statements, standard constructs such as *variables* and *if-then-else* control statements. We show below that we can easily compensate for the lack of such constructs. We also address the way a trigger can call an external procedure.

Simulating variables

Usually, a value used in several statements of the trigger is stored in a local variable. This is particularly recommended if the acquisition cost of this value is high, for instance, if it is computed by an aggregation query.

If the concept of local variable is missing, a technical table containing a single row, where each column implements a variable can be used instead, as illustrated by table **V** in Script 8.23.

```
create table V(A integer, B real, C char(2));
create trigger ...
begin
   ...
   insert into V(A,B)
      select count(*), avg(Account) from CUSTOMER;
   update V set C = 'C1';
   ...
   update CUSTOMER
   set CAT = (select C from V);
   where Account > (select B from V);
   ...
   delete from V;
end;
```

Script 8.23 - Single-row table V acts as a series of local variables

3. This is the case of SQLite: *instead of* triggers on views only, no *for each statement* clause, the trigger *body* is a pure sequence and trigger *execution order* is undefined.
4. SQLite strictly complies with this definition

Simulating *if-then-else* statements

If the trigger language of the DBMS does not include *if-then-else* alternative control statement, then Script 8.24, in which <C>, <C1> and <C2> denote arbitrary SQL conditions, is invalid.

```
create trigger TRG_DEL_T
before delete on T
begin
   if (<C>)
      then delete from S where <C1>;
      else update S set B = null where <C2>;
   endif;
end;
```

Script 8.24 - Standard if-then-else alternative

There are two methods to transform this trigger into pure sequential code. The first one consists in adding respectively <C> and not <C> to each branch of the alternative (Script 8.25). If the evaluation of <C> is expensive, its result can be stored in a local variable, that will be used in next statements as a substitute for <C>.

According to the second method, each branch is translated into a distinct trigger the `when` clause of which decides whether it will fire (Script 8.26).

It is important to note that the validity of these transformations depends on whether the result of the execution of the first branch affects the evaluation of the second instance of condition <C>. For example, in Script 8.25, if <C> is true, the first statement is executed. However, this execution may make condition <C> false, in which case both branches of the alternative will be executed. This point will be examined in more detail in Section 8.18.1.

```
create trigger TRG_DEL_T
before delete on T
begin
   delete from S where <C1> and <C>;
   update S set B = null where <C2> and not <C>;
end;
```

Script 8.25 - Transformation of if-then-else alternative - Method 1

Calling an external procedure

Most DBMS allow the body of a trigger to ask the execution of a procedure written in any proprietary or general programming language. Such a statement will generally look like this one:

```
create trigger TRG_DEL_T1
before delete on T
when <C>
begin
   delete from S where <C1>;
end;

create trigger TRG_DEL_T2
before delete on T
when not <C>
begin
   update S set B = null where <C2>;
end;
```

Script 8.26 - Transformation of if-then-else alternative - Method 2

```
execute procedure sendMail(host,login,pw,...);
```

as in Postgresql, or, simply, as in Oracle PL/SQL:

```
sendMail(host,login,pw,...);
```

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

```
sendMail(host,login,pw,...);
```

we will write:

```
select sendMail(host,login,pw,...);
```

In the same way, instead of:

```
if (<condition>) select sendMail(host,login,pw,...);
```

we will write:

```
select sendMail(host,login,pw,...) where <condition>;
```

If we find this syntax a bit awkward, we can improve it as follows:

```
set sendMail = select sendMail;
```

Then, wherever in the body of a trigger we want to write `<something>` in the output window, we simply write:

```
$sendMail$(host,login,pw,...);
```

8.3 Basic applications

There are many applications of triggers suggested and described in the literature. We will illustrate some of them in Sections 8.4 to 8.11.

8.4 Integrity management

While basic integrity constraints (see 8.2.2) are automatically controlled by the database engine, the management of more advanced constraints must be explicitly expressed into predicates, triggers or stored procedures. Constraints on the values of a row generally can be translated into a `check` condition. In the example below (Script 8.27) which defines the structure of table CUSTORDER (customer order), we state that the *order date* must be less or equal to the current date and that the *quantity* must be a positive number.

```
create table CUSTORDER(
       OrdID   char(12) not null primary key,
       DateOrd date     not null,
       CustID  char(12) not null references CUSTOMER,
       ItemID  char(16) not null references ITEM,
       Qty     integer  not null,
       Status  varchar(20) not null,
       constraint CH_DateOrd check(DateOrd <= current_date),
       constraint CH_Qty check(Qty > 0));
```

Script 8.27 - Controlling the values of columns DateOrd and Qty

Both constraints could have been controlled by a single check condition. In this example, each constraint is named and is controlled by its own check condition to

allow a more precise information to be sent to the application program. If we submit the following statement:

```
insert into CUSTORDER
values ('2020-S33','2040-02-15','D-568',15);
```

... the operation is aborted and this message is sent to the program:

```
CHECK constraint failed: CH_DateOrd
```

A constraint that makes use of values external to the current row (i.e., values of other rows in the same table or in other tables) will be expressed by one or several triggers.

Let us consider that a customer order can be recorded only if the requested quantity is not higher than the quantity available of the item referenced by column ItemID. We assume that this quantity is stored in column Qavail of table ITEM, the primary key of which is (ItemID).

Usually, a trigger that controls a constraint will check whether *the constraint is violated*, in which case exception is raised (Script 8.28).

```
create trigger TRG_ORDER_QTY
before insert or update of Qty,ItemID on CUSTORDER
for each row
when new.Qty > (select Qavail from ITEM
                where  ItemID = new.ItemID)
begin
   raise('Quantity error in order '||old.OrdID);
end;
```

Script 8.28 - Trigger controlling the values of column OrdQty (CUSTORDER side)

This trigger controls the value of Qty against three events affecting the data of CUSTORDER table: inserting a new row, modifying the current value of Qty of an existing row and changing the current item referenced by an existing row.

To fully control the evolution of the data, we must also ensure that any modification that decreases the value of column Qavail of an ITEM row does not invalidate the value of Qty of all the dependent CUSTORDER rows. Symbolically, the constraint Qavail <= sum(Qty) must be maintained in any condition. This is left as an exercise.

The basic integrity constraints, predicates and the kind of trigger shown above all contribute to define the **valid states** of the database. Let's say that checking whether such constraints have been satisfied, we only need to examine the snapshot of the data after the modification.

To control the **valid transitions** between states, we need to compare two snapshots, *before* and *after* the modification. Column Status of table CUSTORDER records the evolution of an order through its successive processes: recording, validation, execution, payment, archiving, etc. So, the successive values of Status of a

definite order must reflect the sequence of these processes: 'recorded' → 'validated' → ... → 'executed' → 'paid' → 'archived'.

Checking that this order is respected can only be performed by triggers, in which both previous and new states are simultaneously available through row aliases **old** and **new**. The trigger shown in Script 8.29 checks that Status can be set to 'archived' only if the previous value was 'paid'. old.Status denotes the value of Status before the update and new.Status the new value it would take, should the update succeed. Completing this code to address the other valid value transitions is straightforward.

```
create trigger TRG_ORDER_STATUS
before update of Status on CUSTORDER
for each row
when new.Status = 'archived' and old.Status <> 'paid'
begin
   raise('Invalid Status update in '||old.OrdID);
end;
```

Script 8.29 - Checking valid Status value transitions

8.5 Non standard system behavior

The way the SQL engine reacts to data modification operations may not meet the requirements of the users. Let us consider the foreign key construct, which is the preferred way to enforce referential integrity. When an operation is likely to violate referential integrity, the DBMS will either refuse it (*no action*, *restrict*), or execute the operation then perform additional actions in order to make the data correct again (*cascade*, *set null*, *set default*).

We suppose that our database comprises, among others, the two tables:

- `CUSTORDER(OrdID,...)`, that records the data of customer orders
- `DETAIL(OrdID,ItemID,...)`, each row of which records the reference of one of the item ordered by order OrdID.

Quite naturally, column OrdID of DETAIL is declared a foreign key to table CUSTORDER. We have found it convenient to assign this foreign key a cascade mode for delete and update operations. DETAIL rows can be deleted freely. However, we wouldn't like to see the last DETAIL row of a customer order being deleted. In such a case, we could either prevent this last row to be deleted (a kind of reverse no action mode) or execute an additional action, namely deleting the CUSTORDER row (a kind of reverse cascade mode). This can be controlled through triggers. Let us adopt the second rule: *when the last detail of an order disappears, this order also disappears*.

This rule is implemented by the trigger of Script 8.30.

```
create trigger TRG_LAST_DETAIL
after delete on DETAIL
for each row
when (select count() from DETAIL where OrdID = old.OrdID) = 0
begin
   delete from CUSTORDER where OrdID = old.OrdID;
end;
```

Script 8.30 - Deleting a CUSTORDER row when its last dependent DETAIL row has been deleted

8.6 Updatable view

In most DBMS, updating data through a view is allowed only in very limited cases. For instance, the query that defines the view cannot include joins, subqueries, aggregate functions but must include all the not null columns (therefore also the primary key of the source table). This means that, in general, we **can** query the data through the view but we **must** update them on the source table, which is particularly awkward!

Thanks to appropriate triggers we can design SQL views that allow us to query **and** update the data.

To show how we can build such views, we consider that the database contains the following tables:

- `CUSTOMER(CustID,Name,Address,...)`, that records the data of the customer,
- `CUSTORDER(OrdID,CustID,...)`, that records the data of orders placed by the customer

We would like to view the data of customer orders augmented with the name and the address of the parent customer. This is what the definition of Script 8.31 gives us.

```
create view CUSTORDER_FULL(OrdID,CustID,Name,Address,...)
as  select CustID,O.CustID,Name,Address
    from   CUSTORDER O, CUSTOMER C
    where  O.CustID = C.CustID;
```

Script 8.31 - A view that is not updatable (so far)

Typically, being based on a join, this view is not updatable. The SQL engine does not understand what we mean when we ask it to execute `insert`, `delete` and

`update` queries on this view. So, we need to explain our own interpretation of these operations. This is why we define the triggers of Script 8.32.

Now, a query such as

```
update CUSTORDER_FULL set Name='O''Neil' where OrdID='3017';
```

becomes perfectly valid, so that the programmers can ignore the difference between base tables and views!

```
create trigger TRG_CORD_FULL_INSERT
instead of delete on CUSTORDER_FULL
begin
   insert into CUSTORDER values (new.OrdID,new.CustID,...);
end;
create trigger TRG_CORD_FULL_DELETE
instead of delete on CUSTORDER_FULL
begin
   delete from CUSTORDER where OrdID = old.OrdID;
end;
create trigger TRG_CORD_FULL_UPDATE
instead of update of OrdID,Name,Address on CUSTORDER_FULL
begin
   update CUSTOMER
   set    Name = new.Name
   where  CustID = new.CustID and new.Name <> old.Name;
   update CUSTOMER
   set    Address = new.Address
   where  CustID = new.CustID and new.Address <> old.Address;
   update CUSTORDER
   set    OrdID = new.OrdID
   where  new.OrdID <> old.OrdID;
   ...
end;
```

Script 8.32 - The triggers that make view `CUSTORDER_FULL` updatable

8.7 Redundancy management and derived data

According to a popular proverb among database people, *each real world fact must be represented once and only once*. So, theoretically, a normalized database includes no redundant data. Practically, things may be different. It may be useful to record some information in more than one place, or to record data derived from other data, and this for various reasons.

Among them:

– *Expressiveness and readability* of the data structures. Some essential data (from the point of view of users or even programmers) may be missing from the database schema because they can be calculate from more elementary values. Considering table CUSTORDER described in Script 8.27, a naive user may be troubled by the absence of a column giving the *amount* of each order. This amount is easy to compute for each row: we just multiply the value of Qty by the unit price, supposedly stored in column UnitPrice of the referenced ITEM row. So, adding a new column that stores this amount value provides a more natural representation of customer orders, but at the cost of increased storage space and processing time.

– *Efficiency.* Processing a customer order requires the extraction of additional data from the CUSTOMER table (notably their name and address) and from the ITEM table (the item unit price for example). By permanently maintaining a copy of these data in each CUSTORDER row, we avoid two costly joins. This is even more critical when the database is distributed among several sites. Some data can be replicated in several sites to avoid internet transmission cost and latency.

– *Robustness against data corruption*. It is usual to copy critical data in several places. This makes it easier to recover them when a copy has been accidentally or intentionally destroyed.

To illustrate the principles of redundancy management, we will develop further the example of the amount of customer orders mentioned above.

First, we augment the schema of table CUSTORDER as follows (we ignore the columns that are of no use in this development):

```
create table CUSTORDER(
       OrdID     char(12) not null primary key,
       ...,
       ItemID    char(16) not null references ITEM
                                    on update cascade
                                    on delete cascade,
       Qty       integer  not null,
       Amount    integer  not null,
       ...);
create table ITEM(
       ItemID    char(16) not null primary key,
       ...,
       UnitPrice integer not null,
       ...);
```

Script 8.33 - New computed column Amount is added to table CUSTORDER

The value of column Amount of CUSTORDER row **ord** is computed by this formula:

```
ord.Amount = ord.Qty * (select UnitPrice
                        from   ITEM
                        where  ItemID = ord.ItemID)
```

Though Amount is a computed column, it cannot be declared a **generated** column since its formula involves external columns. Therefore, we must rely on triggers to make its values comply with the formula.[5]

Let us see this formula as an *equality relation* between two quantities. We have to identify which events may alter this equality. The formula uses the data of two tables, CUSTORDER and ITEM. In table CUSTORDER, three columns are involved, ItemID, Qty and Amount. In table ITEM, two columns are involved, ItemID and UnitPrice. The events of interest are those which modify the state of these objects. For each of them, we decide the action to perform.

- **Events of table CUSTORDER**
  - *insert row*: compute the value of Amount.
  - *update of ItemID*: change of referenced item; recompute the value of Amount.
  - *update of Qty*: recompute the value of Amount.
  - *update of Amount*: forbidden.
  - *delete row*: no effect.

- **Events of table ITEM**
  - *insert row*: no effect (no customer order yet).
  - *update of ItemID*: no action; automatically managed according to the cascade update mode of the foreign key.
  - *update of UnitPrice*: recompute the Amount value of all the dependent CUSTORDER rows.
  - *delete row*: no action; automatically managed according to the cascade delete mode of the foreign key.

Now, we have enough information to build the triggers that preserve the equality relation.

Managing CUSTORDER table

The `insert` event and the `update` of Qty and ItemID events can all be coped with by a single trigger (Script 8.34).[6]

5. An alternative structure could be thought of: we add in CUSTORDER new column UnitPrice, the value of which is that of column UnitPrice of the row referenced by ord.ItemID. This redundancy is simple to manage (e.g., there is no need to control the value of Qty). Now, Amount can be declared a *generated* column.
6. In SQLite, a trigger is fired by one event only. So, this trigger must be split into two distinct triggers.

```
create trigger TRG_ORDER_MODIF
after insert, update of Qty,ItemID on CUSTORDER
for each row
begin
   update CUSTORDER
   set Amount = new.Qty*(select UnitPrice from ITEM
                         where  ItemID = new.ItemID)
   where OrdID = new.OrdID;
end;
```

Script 8.34 - Recalculation of derived column Amount due to events on table CUSTORDER

Prohibiting the explicit modification of Amount can be done in several ways. The most obvious is to *revoke* the right of updating this column from all users:

```
revoke update (Amount) on CUSTORDER from public;
```

Another technique uses a trigger that cancels any operation attempting to change the value of Amount. The problem with this technique is that it will also cancel other valid modifications requested by the same update statement.

Finally, we suggest a third technique that accepts the modification but silently recovers the correct value through a trigger similar to that of Script 8.34. This leads to the modified code of Script 8.35.

```
create trigger TRG_ORDER_MODIF
after insert, update of Qty,ItemID,Amount on CUSTORDER
for each row
begin
   update CUSTORDER
   set Amount = new.Qty*(select UnitPrice from ITEM
                         where  ItemID = new.ItemID)
   where OrdID = new.OrdID;
end;
```

Script 8.35 - Extending trigger 8.34 to prevent users from updating the *Amount* column

Managing ITEM table

We only have to cope with update actions on UnitPrice (Script 8.36).

```
create trigger TRG_ITEM_MODIF
after update of UnitPrice on CUSTORDER
for each row
begin
   update CUSTORDER
   set Amount = Qty*(select new.UnitPrice from ITEM
                     where  ItemID = new.ItemID);
   where OrdID = new.OrdID;
end;
```

Script 8.36 - Propagating UnitPrice modification in table ITEM to the Amount column in CUSTORDER table

8.8 Data modification journaling

We want to keep a precise record of data modification operations on a database. The description of the operations can be written in a text file or in a table of the database itself. We chose the latter approach and we create table JOURNAL, in which the history of `insert`, `update` and `delete` operations affecting each table is recorded.

The content of journal entries depends on the objective we assign to the journal. If the goal is to compute activity statistics, only minimal information will be needed, such as a timestamp, the table name and the nature of the operation. We could want to build a *forward journal* from which the operations can be **replayed** later if necessary. In this case, we must record the complete description of the SQL operations. Or, we could build a *backward journal*, that can be used to **undo** some of the operations that have been successfully executed.

In this exercise, we describe the management of a forward journal. A journal entry will comprise the following information:

- **LogID** : sequence number of the journal entry,
- **OpTime**: date (`aaaa-mm-jj`) and time (`hh:mm:ss.mmm`) of the operation,
- **Operation**: nature of the operation ('insert', 'update', 'delete'),
- **TableName**: name of the table,
- **RowID**: primary key of the row,
- **Arguments**: detail of the operation, e.g., old/new values.

The structure of table JOURNAL is shown in Script 8.37.

```
create table JOURNAL(
    LogID     integer not null primary key autoincrement,
    OpTime    datetime not null,
    Operation char(32) not null,
    TableName varchar(64) not null,
    RowID     varchar(128) not null,
    Arguments varchar(1024));
```

Script 8.37 - Structure of the journal table

Three triggers are suggested to control `insert` (Script 8.38), `update` (Script 8.39) and `delete` operations (Script 8.40). Function `current_timestamp_full()` is a UDF[7] (user-defined function) that returns, in ISO format, the current timestamp with a precision of millisecond.

The arguments of the triggers controlling `insert` and `update` operations collect the new column values in a character string in a format that makes it easy to extract their individual values. Here we have chosen the CSV format, which is more concise than XML and JSON.

```
create trigger TRG_ORDER_LOG_INS
after insert on CUSTORDER
for each row
begin
  insert into JOURNAL(OpTime,Operation,TableName,RowID,Argument)
         values(current_timestamp_full(),
                'insert',
                'CUSTORDER',
                new.OrdID,
                '"'||new.DateOrd||'"'
               ||','||'"'||new.CustID||'"'
               ||','||'"'||new.ItemID||'"'
               ||','||cast(new.Qty as char)
               ||','||'"'||new.Status||'"') $;$
end;
```

Script 8.38 - Generating the JOURNAL entries for the **insert** operations

Let us execute this sequence of statements, applied to an empty table:

```
insert into CUSTORDER
       values('123','2020-02-16','B512','PA45',12,'recorded');
insert into CUSTORDER
       values('184','2020-02-17','H054','PA60',5,'recorded');
delete from CUSTORDER where OrdID = '123';
insert into CUSTORDER
       values('270','2020-02-19','F400','PH222',2,'recorded');
```

7. This function is a member of UDF library **SQLiteUDFlib.py** of the SQLfast distribution.

```
update CUSTORDER set Status = 'validated' where OrdID = '184';
```

```
create trigger TRG_ORDER_LOG_UPD
after update on CUSTORDER
for each row
begin
  insert into JOURNAL(OpTime,Operation,TableName,RowID,Argument)
         values(current_timestamp_full(),
                'update',
                'CUSTORDER',
                old.OrdID,
                '"'||case when new.DateOrd = old.DateOrd then ''
                          else new.DateOrd end||'"'
             ||',"'||case when new.CustID = old.CustID then ''
                          else new.CustID end||'"'
             ||',"'||case when new.ItemID = old.ItemID then ''
                          else new.ItemID end||'"'
             ||','|| case when new.Qty = old.Qty then ''
                          else cast(new.Qty as char) end
             ||',"'||case when new.Status = old.Status then ''
                          else new.Status end||'"');
end;
```

Script 8.39 - Generating the JOURNAL entries for the **update** operations

```
create trigger TRG_ORDER_LOG_DEL
after delete on CUSTORDER
for each row
begin
  insert into JOURNAL(OpTime,Operation,TableName,RowID)
         values(current_timestamp_full(),
               'delete',
               'CUSTORDER',
               old.OrdID) $;$
end;
```

Script 8.40 - Generating the JOURNAL entries for the **delete** operations

It will create the following state of table CUSTOMER.

OrdID	DateOrd	CustID	ItemID	Qty	Status
184	2020-02-17	H054	PA60	5	validated
270	2020-02-19	F400	PH222	2	recorded

As to the the content of table JOURNAL, it will look like this one (shown in two parts to make it readable):

```
+-------+-------------------------+-----------+-----------+----
| LogID | OpTime                  | Operation | TableName | ...
+-------+-------------------------+-----------+-----------+----
| 1     | 2020-02-08 17:54:10.971 | insert    | CUSTORDER | ...
| 2     | 2020-02-08 17:54:11.058 | insert    | CUSTORDER | ...
| 3     | 2020-02-08 17:54:11.532 | delete    | CUSTORDER | ...
| 4     | 2020-02-08 17:54:11.604 | insert    | CUSTORDER | ...
| 5     | 2020-02-08 17:54:12.113 | update    | CUSTORDER | ...
+-------+-------------------------+-----------+-----------+----
+-------+-------+-----------------------------------------+
| LogID | RowID | Arguments                               |
+-------+-------+-----------------------------------------+
| 1     | 123   | "2020-02-16","B512","PA45",12,"recorded" |
| 2     | 184   | "2020-02-17","H054","PA60",5,"recorded"  |
| 3     | 123   | --                                      |
| 4     | 270   | "2020-02-19","F400","PH222",2,"recorded" |
| 5     | 184   | "","","","","validated"                 |
+-------+-------+-----------------------------------------+
```

We observe that this table provides sufficient information to rebuild the source SQL DML queries.[8]

A complete application is available in script **Scripts\Case-Studies\Case_Active_DB/Data-modification-logging.sql**.

8.9 Alerters

An *alerter* is a mechanism intended to inform some agent (e.g., a person or an external process) about specific events that have occurred. It is somehow similar to *journaling*, except that the information is sent in real time outside the database.

In the context of a database application, sending an email is a popular way to transmit this information and, quite naturally, the *sending agent* is a **trigger** associated with the table in which the events occur.

Let us illustrate the concept of alerter with a table called STOCK, (Script 8.41) in which we store the *quantity on hand* of a set of products (column **QonHand**).

Whenever a quantity **q** of a product is consumed, **q** is subtracted from the quantity on hand of this product. When the latter quantity becomes too low, the employee responsible for the resupply of this product receives a warning email. What *too low* exactly means is specified by column **Qresupply**, that tells below what quantity a resupply must be carried out.

For instance, row `('P001',100,20)` indicates that product `P001` has a quantity on hand of `100` units and a resupply level of `20`. When the value of QonHand falls below that of Qresupply, it is time to replenish the stock.

8. Actually, this is only true for modification operations that apply to *individual rows*. For multirow queries, such as this one: "`delete from CUSTORDER where ItemID = 'PA45'`", that is likely to affect several rows, the journal will include a distinct entry for each of these rows.

```
createOrReplaceDB PRODUCTS.db;
create table STOCK(
       ProdID    char(4) not null primary key,
       QonHand   integer not null,
       Qresupply integer not null);
insert into STOCK values ('P001',50,20),
                         ('P002',40,10),
                         ('P003',90,40);
```

Script 8.41 - Creating the test database

Script 8.42 shows the code of the trigger that implements the alerter. It fires when the value of QonHand is modified in such a way that it *crosses* that of Qsupply, a condition that can be translated as follows[9]:

```
old.QonHand >= new.Qresupply and new.QonHand < new.Qresupply
```

The action of the trigger is quite simple: it sends an email to the supply manager, asking her to replenish the stock of the product that just gets out of stock.

The **set** statement builds the body of the mail and the **exec** statement sends the mail to an SMTP server through the sendMail procedure. For obvious privacy reasons, the critical sending parameters are acquired through an **ask** statement.

If we execute the following `update` queries,

```
update STOCK set QonHand = QonHand - 35 where ProdID = 'P001';
wait 1000;
update STOCK set QonHand = QonHand - 25 where ProdID = 'P002';
wait 1000;
update STOCK set QonHand = QonHand - 60 where ProdID = 'P003';
```

... the supply manager will receive two messages:

```
Quantity on hand of item #P001
has dropped from 50 to 15
on 2020-04-13 11:31:33


Quantity on hand of item #P003
has dropped from 90 to 30
on 2020-04-13 11:31:35
```

9. This condition also addresses the simultaneous modification of column Qsupply.

```
ask host,login,pw = [Initialize mail parameters]
                    Host:|Login:|Pw:;
set from = alert@acme.com;
set to   = supply.manager@acme.com;
set subj = Replenishment required;

openDB PRODUCTS.db;

create trigger TRG_STOCK_ALERT
after update of QonHand on STOCK
for each row
when old.QonHand >= new.Qresupply
and  new.QonHand < new.Qresupply
begin
  set body = 'Quantity on hand of <b>item #'
           ||old.ProdID
           ||'</b><br>has dropped from <b>'
           ||cast(old.QonHand as char)||'</b> to <b>'
           ||cast(new.QonHand as char)
           ||'</b><br>on '
           ||datetime(current_timestamp,'localtime');

  exec sendMail('$host$','$login$','$pw$',
                '$from$','$to$','','html',
                '§subj§',body,'');

closeDB;
```

Script 8.42 - The trigger that implements the alerter

About the *sendMail* procedure

The **sendMail** procedure is a general-purpose procedure that sends an e-mail through an SMTP server. It requires the following parameters:

```
sendMail(
     Host,          # SMTP server address
     Login,         # (server) user id
     Pw,            # (server) password
     From,          # sender address
     To,            # recipient address list
     Cc,            # carbon-copy address list
     Format,        # message format: 'plain' or 'html'
     Subject,       # subject text
     Body,          # body text
     Attach)        # list of attached file names
```

In the SQLfast version, calling this procedure in the body of the trigger is slightly different due to the limitation of the allowed syntax (see Section 8.2.7). The modified version is shown in Script 8.43.

A more informative alerter is available in the scripts of the case study. It not only reacts to the low level of a product but also to any replenishment operation.

```
create trigger TRG_STOCK_ALERT
after update of QonHand on STOCK
for each row
when old.QonHand > new.Qresupply
and  new.QonHand <= new.Qresupply
begin
  select raise(IGNORE)
  where  sendMail('$host$','$login$','$pw$',
         '$from$','$to$','',
         'html','§subj§',
         'Quantity on hand of item <b>#'
        ||old.ProdID
        ||'</b><br>has dropped from <b>'
        ||cast(old.QonHand as char)||'</b> to <b>'
        ||cast(new.QonHand as char)
        ||'</b><br>on '
        ||datetime(current_timestamp,'localtime'),'')
       = 'x'; end;
closeDB;
```

Script 8.43 - The SQLite version of the alerter [Alerter-by-mail-v1.sql]

8.10 Type-subtype implementation

This application shows how complex semantic structures, namely supertype-subtype hierarchies, can be implemented through active database techniques.

8.10.1 About database design

Designing a complex database generally starts with an important phase, *conceptual modeling*, during which the application domain[10] is analyzed. The result is called the *conceptual schema* of the database. This schema identifies in the application domain the categories - or types - of entities (*customers* and *orders*), their attributes (*name* of customer, *date* of order) and their relationships (customers *place* orders). So, the main components of a conceptual schema are the entity types, their attributes and their relationship types.

10. Reminder: the application domain is that part of the real world we are interested in and about which we intend to collect, memorize and exploit information. Also called the *universe of discourse*.

This way of describing things[11] is a trade off between two conflicting requirements: to produce a natural and intuitive description of the application domain (the *user view*) and to easily translate it into database structures (the *developer view*). Indeed, generating an SQL schema from a conceptual schema seems to be fairly easy: each entity type is represented by a table, each attribute by a column, each relationship type by a foreign key and, finally, each entity by a row.

8.10.2 The Type-subtype concept

However, the real world we are interested in is not always that simple. It may include many other important fact types that we would like to represent in the database but that are much less straightforward to translate into SQL data structures. Among them, the fact that *an entity may belong to more than one type*, which would imply, in its SQL translation, that a row may belong to more than one table, which is, at first glance, impossible.

For instance, it is generally agreed that *professors* and *students* are *persons*. So, Stonebraker, who is a *professor*, will be classified as a member of type PROFESSOR, and, as such, is an entity of type PERSON as well. In other words, the set of PROFESSOR entities is a subset of the set of PERSON entities.

Obviously, the conceptual schema must report on these facts. It does so in this way:

- We declare entity type PERSON with all its properties (attributes, relationship types, constraints, and the like).
- Then, we declare entity type PROFESSOR a *subtype* of PERSON. PERSON is therefore the *supertype* of PROFESSOR. All the properties of PERSON also are those of PROFESSOR. This mechanism is called *inheritance*: attributes PersID and Name are *inherited* attributes of PROFESSOR. In addition, entity type PROFESSOR may have proper attributes, here Specialty.
- And of course the same for STUDENT.

Synthetically, one says that there is an *is-a* relation between (CUSTOMER, SUPPLIER) and PERSON, an expression that tells that each professor *is-a* person.

These entity types and their *is-a* relations are illustrated in Figure 8.1.[12] Symbol **D** in the triangle indicates that the subtypes are *disjoint*, that is, they cannot share common entities.

Such patterns are at the basis of most programming languages but are generally less frequent in relational databases for reasons we will discuss below.

11. In this example, we refer to the Entity-relationship model, the most popular database conceptual modeling language.
12. The graphical conventions are those of the GER, a general-purpose entity-relationship language intended to represent database schemas according to various data models. This schema has been drawn in the DBMain Case tool.

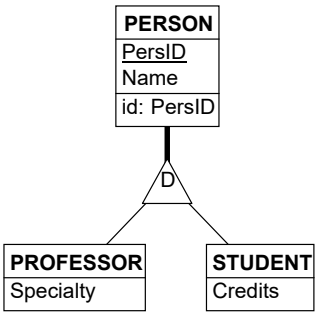


Figure 8.1 - A conceptual schema showing an **is-a** relation

8.10.3Is-a relations in SQL3

Now, we can formulate the central question of this study: how can we translate *is-a* relations into database structures? The answer depends on the level of the *SQL standard* the DBMS implements.

Besides standard table definition (as in Script 8.27 for example), SQL3 defines the concept of *table type*, from which we can create any number of *typed tables*. In the example below, we create type TPERSON, then two tables of this type. Each of these tables comprises columns PersID and Name as they are defined in type TPERSON:

```
create type TPERSON(PersID char(5),Name varchar(32));

create table PROFESSOR of TPERSON;
create table STUDENT of TPERSON;

insert into STUDENT values('P345','Ullman');
```

An interesting property of table type organization is that a *type* can be declared a *subtype* of another one, the latter being the *supertype* of the former. In the example that follows, TTEACHER and TSTUDENT are declared subtypes of TPERSON through clause `under`. This relation also exists between the tables we create from these types: PROFESSOR and STUDENT are *subtables* of PERSON, which in turn is their *supertable*.[13] Since TTEACHER *inherits* from TPERSON, columns PersID and Name need not be declared in this subtype.

```
create type TPERSON(PersID char(5),Name varchar(32));
create type TTEACHER(Specialty varchar(24) under TPERSON;
create type TSTUDENT(Credits integer) under TPERSON
```

13. This example is simplified, the definitions of subtype structures being fairly complex. It is merely intended to give the reader the taste of SQL subtypes, no more.

```
create table PERSON of TPERSON;
create table STUDENT of TSTUDENT under PERSON;
create table PROFESSOR of TTEACHER under PERSON;

insert into PERSON values ('E0123','Smith');
insert into STUDENT values ('S0234','Stonebraker',45);
```

By definition, table PERSON contains a set of PERSON rows. In addition, all the rows of STUDENT and those of PROFESSOR also are rows of PERSON. So, the following query is valid, though it will only return column PersID and Name of the rows of the subtables:

```
select * from PERSON where Name = 'Stonebraker';
```

The SQL3 specifications discussed above are an attempt to offer database designers an easy way to translate the *is-a* relations of conceptual schemas, and, on the other hand, to bridge the database world with the object-oriented paradigm of most programming languages.[14]

8.10.4Is-a relations in SQL2

The SQL3 specifications are both complex and limited.[15] In addition their implementations in major DBMS often are proprietary and therefore not compatible.[16]

Some DBMS, including SQLite, even ignore table types. As a result, despite their undoubted usefulness, these structures are not popular and are rarely used in practice.

So, why not develop home-made subtable-supertable structures? In this way, we can profit from their modeling advantages without suffering from their drawbacks.

Let us try to develop the PERSON/PROFESSOR/STUDENT example described above with standard database structures. First, we must choose an appropriate implementation of the data that underlie this structure. There are several simple solutions. Among them, the following are the most popular:

1. **Supertable representation**: a single PERSON table gathers all the rows of the three tables. Its schema is the following:

```
create table PERSON(PersID not null,Name not null,
                    Subtype,Specialty,Credit)
```

14. Purely object-oriented database systems have been developed in the 1980s. However, they never met with the hoped-for success in the industrial world, and therefore gradually disappeared in the 1990s.
15. Two examples: (1) two subtables of the same supertable are disjoint (a PROFESSOR row may not be a STUDENT row) and (2) a type may not have more than one supertype (PostgreSQL is an exception).
16. SQL standards are just *recommendations*.

Nullable column Subtype indicates whether the row represents a professor, a student or none of them.[17] Subtables are reconstructed by selecting rows based on column Subtype.

2. **Subtable representation**: three tables represents the *partition* of PERSON, i.e., POPULATION_ONLY (people who are neither professor nor student), PROFESSOR (people who are professor only) and STUDENT (people who are student only). Supertable PERSON is reconstructed through a `union` operator. This representation is not recommended to implement non disjoint subtables.
3. **Table representation**: the supertable is implemented by table PERSON as defined in the structure. PersID is its primary key. Each subtable is represented by a table comprising primary key PersID plus its proper columns. PersID is also a foreign key that references PERSON. Subtables are reconstructed through `join` operators.

Let us choose the last implementation, which will prove the most flexible. Script 8.44 creates the three base technical tables. Since they (hopefully) will never be directly referenced by the users (neither by the programmers), we prefix their name with an underscore symbol.

```
create table _PERSON (
        PersID  char(5) not null primary key,
        Name    varchar(32) not null);
create table _PROFESSOR (
        PersID  char(5) not null primary key
                references _PERSON
                on update cascade
                on delete cascade,
        Specialty varchar(24) not null);
create table _STUDENT (
        PersID  char(5) not null primary key
                references _PERSON
                on update cascade
                on delete cascade,
        Credits integer not null);
```

Script 8.44 - Representing each supertable and subtable by a standard table

Then, we create three views, named PERSON, PROFESSOR and STUDENT, that will be the only external interface for anyone interested in these data. We decide that view PERSON provides all the available data of all its members, be they professor, student or none of them. It is created by left outer joins. As to views PROFESSOR

17. Or both of them (non disjoint subtables), though this case may be more complex to understand, to manage and to process.

and STUDENT, they show all the data respectively of professors and students. They are created by inner joins.

```
create view PERSON (PersID,Name,Specialty,Credits)
    as  select P.PersID,Name,Specialty,Credits
        from _PERSON P
             left join _PROFESSOR F using (PersID)
             left join _STUDENT S   using (PersID);
create view PROFESSOR (PersID,Name,Specialty)
    as  select P.PersID,Name,Specialty
        from  _PERSON P, _PROFESSOR F
        where  P.PersID = F.PersID;
create view STUDENT (PersID,Name,Credits)
    as  select P.PersID,Name,Credits
        from  _PERSON P, _STUDENT S
        where  P.PersID = S.PersID;
```

Script 8.45 - Representing each supertable and subtable by a standard table

We populate these tables with a sample of initial rows (Script 8.46).

```
insert into _PERSON    values ('P0123','Smith');
insert into _PERSON    values ('P0234','Stonebraker');
insert into _PROFESSOR values ('P0234','DB Theory');
insert into _PERSON    values ('P0345','Ullman');
insert into _STUDENT   values ('P0345',45);
insert into _PERSON    values ('P0456','Van Bolle');
insert into _PROFESSOR values ('P0456','Advanced SQL');
insert into _STUDENT   values ('P0456',30);
```

Script 8.46 - Introducing the data of four persons (in technical tables)

The content of the views (see below) shows an interesting variety of persons: `'P0123'` is neither a professor nor a student, `'P0234'` is a professor only, `'P0345'` is a student only and `'P0456'` is both a professor and a student.

View PERSON

PersID	Name	Specialty	Credits
P0123	Smith	--	--
P0234	Stonebraker	DB Theory	--
P0345	Ullman	--	45
P0456	Van Bolle	Advanced SQL	30

View PROFESSOR

PersID	Name	Specialty
P0234	Stonebraker	DB Theory
P0456	Van Bolle	Advanced SQL

View STUDENT

PersID	Name	Credits
P0345	Ullman	45
P0456	Van Bolle	30

Querying data through these views is quite natural. However, we cannot say the same for data creation. Indeed, we must insert data in the three technical table that should remain hidden, a state of affair we cannot accept.

This is where *active databases* come in.

Our new objective is to allow data modification operations to be performed on these views instead of on the technical tables. In other words, we must make these views *updatable*. In most DBMS, modifying data through a view is very restricted (or even forbidden, as in SQLite). In particular, the view cannot be defined by a join.

Getting around this difficulty is simple: since the DBMS does not understand what `insert`, `delete` and `update` queries mean, we will attach to each view a set of triggers that explain how to execute these operations.

8.10.5The insert operations

We first state how we would like to proceed to insert data. Script 8.47 shows a first correct approach. For a person who is neither a professor nor a student, the data are inserted in view PERSON (e.g., 'P0123'). The data of a person who is either a professor or a student (but not both) are inserted into views PROFESSOR or STUDENT respectively (e.g., 'P0234' and 'P0334').

The last case is a bit more complex: the data of a person who is both a professor and a student are inserted in both subtables (e.g., `'P0456'`). So, two `insert` queries for one person, a ratio which is far from natural!

```
insert into PERSON(PersID,Name) values ('P0123','Smith');
insert into PROFESSOR values ('P0234','Stonebraker','DB Theory');
insert into STUDENT   values ('P0345','Ullman',45);
insert into PROFESSOR values ('P0456','Van Bolle','Advanced SQL');
insert into STUDENT   values ('P0456','Van Bolle',30);
```

Script 8.47 - Introducing of the data of four persons (in views) - Method 1

In the second approach (Script 8.48), all the data are inserted into supertable PERSON. If the person is a professor, then column Specialty must have a value. If the person is a student, column Credits must have a value. For a *professor-student*, both columns must be valued.

```
insert into PERSON(PersID,Name) values ('P0123','Smith');
insert into PERSON(PersID,Name,Specialty)
       values ('P0234','Stonebraker','DB Theory');
insert into PERSON(PersID,Name,Credits)
       values ('P0345','Ullman',45);
insert into PERSON(PersID,Name,Specialty,Credits)
       values ('P0456','Van Bolle','Advanced SQL',30);
```

Script 8.48 - Introducing of the data of four persons (in views) - Method 2

The insert triggers

The `insert` triggers coded in Script 8.49 are straightforward and allow for both approaches.

When the data are inserted in view PERSON (as in Script 8.48), trigger TRG_PER_INSERT fires. It first inserts a fragment in table _PERSON. Then, if Specialty is not null, it inserts a fragment in table _PROFESSOR. Similarly, if Credits is not null, it inserts a fragment in table _STUDENT. The subtype(s) indicator derives from the presence of a value of a *not null* column of the subtable(s), namely, here, Specialty for PROFESSOR and Credits for STUDENT).

When the data are inserted in view PROFESSOR (as in Script 8.47), trigger TRG_PRO_INSERT fires. It inserts a fragment in table _PERSON if it didn't already exist (if value new.PersID is not already present in this table). Then, it inserts the remaining fragment in table _PROFESSOR. Trigger TRG_STU_INSERT (not shown) behaves in the same way when data are inserted in view STUDENT.

8.10.6 The update operations

Update operations are more powerful than they use to be in standard databases. Indeed, not only are they used to change the value of columns but they also allow us to assign, remove or change the status of persons.

- Changing the value of proper attributes of an existing person

  The name of professor 'P0123' changes

```
update PERSON
set   Name = 'Branson'
where PersID = 'P0123';
```

```
create trigger TRG_PER_INSERT
instead of insert on PERSON
for each row
begin
   insert into _PERSON
      values (new.PersID,new.Name);
   insert into _PROFESSOR
      select new.PersID,new.Specialty
      where  new.Specialty is not null;
   insert into _STUDENT
      select new.PersID,new.Credits
      where new.Credits is not null;
end;

create trigger TRG_PRO_INSERT
instead of insert on PROFESSOR
for each row
when new.Specialty is not null
begin
   insert into _PERSON
      select new.PersID,new.Name
      where  new.PersID not in (select PersID from _PERSON);
   insert into _PROFESSOR
      values (new.PersID,new.Specialty);
end;
```

Script 8.49 - The triggers that control **data insertions** in the user views(trigger TRG_STU_INSERT not shown)

- Changing the value of attributes of an existing professor
  The speciality of professor 'P0234' changes

```
update PROFESSOR
set   Specialty = 'DB Theory and Practice'
where PersID = 'P0234';
```

or

```
update PERSON
set   Specialty = 'DB Theory and Practice'
where PersID = 'P0234';
```

- Assigning a status to an existing person
  Person 'P0123' becomes a professor

```
update PERSON
set   Specialty = 'Business models'
where PersID = 'P0123';
```

- Removing a status of an existing person
  Person 'P0234' is no longer a professor

```
update PERSON
set   Specialty = null
where PersID = 'P0234';
```

or

```
delete from PROFESSOR
where PersID = 'P0234';
```

The update triggers

The update triggers are shown in Script 8.50.

Trigger TRG_PER_UPDATE1 manages the modifications of the proper columns of the supertable. For performance reasons, we have assigned a distinct update query to each of these columns instead of one big query that updates all the columns, whether their value has changed or not. If the primary key is updated, the modification is automatically propagated to the foreign keys in _PROFESSOR and _STUDENT thanks to their `on update cascade` clauses.

Trigger TRG_PER_UPDATE2 manages the modification of the proper columns of view PROFESSOR imported into the supertable view (here, column Specialty). If the new value of Specialty is not *null*, the effect of the operation depends on whether the person concerned already is a professor or not. In the first case, column Specialty is updated in the dependent _PROFESSOR row. In the second case, a new _PROFESSOR row is created. Assigning a *null* value to Specialty means that this person is no longer a professor. His _PROFESSOR row must therefore be deleted. A similar trigger must be devoted to STUDENT updates.

Trigger TRG_PRO_UPDATE manages the modification of the proper columns of view PROFESSOR. There are two cases. If the new value is not *null*, the column is updated. If this value is null, the person loose his status and the corresponding _PROFESSOR row is deleted.

Similar triggers are built to manage STUDENT updates.

```
create trigger TRG_PER_UPDATE1
instead of update of PersID,Name on PERSON
for each row
begin
   update _PERSON
   set   PersID = new.PersID
   where new.PersID <> old.PersID;
   update _PERSON
   set   Name = new.Name
   where PersID = new.PersID
   and   new.Name <> old.Name;
end;

create trigger TRG_PER_UPDATE2
instead of update of Specialty on PERSON
for each row
begin
   update _PROFESSOR
   set   Specialty = new.Specialty
   where PersID = new.PersID
   and   new.Specialty is not null;
   insert into _PROFESSOR
      select new.PersID,new.Specialty
      where  new.PersID not in (select PersID from _PROFESSOR);
   delete from _PROFESSOR
   where  PersID = new.PersID
   and    new.Specialty is null;
end;

create trigger TRG_PRO_UPDATE
instead of update of Specialty on PROFESSOR
for each row
begin
   update _PROFESSOR
   set   Specialty = new.Specialty
   where PersID = new.PersID
   and   new.Specialty is not null;
   delete from _PROFESSOR
   where  PersID = new.PersID
   and    new.Specialty is null;
end;
```

Script 8.50 - The triggers that control data **update** operations on the user views

8.10.7 The delete operations

We can delete a person (with his status) or merely one of his statutes.

- Deleting a person
  Person 'P0234' is deleted

```
delete from PERSON where PersID = 'P0345';
```

- Removing a status of an existing person
  Person 'P0234' is no longer a professor

```
delete from PROFESSOR where PersID = 'P0234';
```

Delete triggers

The delete triggers are particularly straightforward (Script 8.51).

When a _PERSON row is deleted, the dependent rows in the subtables are automatically deleted, due to the `on delete cascade` clauses of the foreign keys.

```
create trigger TRG_PER_DELETE
instead of delete on PERSON
for each row
begin
   delete from _PERSON
   where  PersID = old.PersID;
end;

create trigger TRG_PRO_DELETE
instead of delete on PROFESSOR
for each row
begin
   delete from _PROFESSOR
   where  PersID = old.PersID;
end;
```

Script 8.51 - The triggers that control **delete** operations on the user views

8.10.8 Automating type-subtype manager production

Maintaining the consistency of a hierarchy of *is-a* relations requires a fairly large set of triggers. However, if we examine carefully these triggers, we find that their structure is quite simple. Once we know:

– the schema of the technical tables (Script 8.44)

– the *is-a* relations, for instance stored in a temporary table ISA(Supertable,Subtable)

writing them is straightforward. In other words, a simple script based on the SQLfast dictionary (`createDictionary`) and the content of table ISA, can automatically generate the complete set of triggers managing the *is-a* hierarchy.

Developing such a script is left to the reader's initiative.

8.11 Other applications

Before discussing and developing the prototype business application that closes this study, we mention three more of the many applications of active databases.

Repair rules

Instead of rejecting `insert` and `update` queries that attempt to introduce erroneous data in the database, these operations are accepted but the data they convey are sanitized or quarantined according to specific repair rules. Some examples:

- *value clipping*: if a value must be greater than or equal to **v**, but actually is lower than this value, it is replaced by **v**
- *dummy rows*: attempting to insert a CUSTORDER row that references a customer who doesn't exist (yet) should be rejected. Instead, a dummy CUSTOMER row is inserted with the CustID value comprised in the invalid row. Later, this row will be completed with additional information on this customer.
- *alternative reference*: a CUSTORDER row to insert references a product that is no longer available. This reference is automatically replaced by that of a similar product.

In all these cases, it is recommended to report in a log table these anomalies and the fixes that have been applied.

Access control

Most SQL implementations provide information about the execution context of queries. This makes it possible to control whether a query can be executed or should be rejected. Let us consider that only employee HR_008 is authorized to update table SALARY, and only on business days from 9 a.m. to 12 a.m. This rule can be enforced by the following trigger:

```
create trigger TRG_SALARY_SECURITY
before update on SALARY
for each row
when not (current_user = 'HR_008'
          and day_of_week(current_date)  between 2 and 6
          and hour_of_time(current_time) between 9 and 12)
begin
   raise('Illegal update of SALARY table');
end;
```

Temporal databases

A database is said *temporal* - or *historical* - when it stores the current and past states of its content.[18] When rows are inserted, updated or deleted, the current state is saved and a new one is added. Each state has a time interval associated with it, indicating the period during which the report is/was valid. These data manipulations may be complex but can be managed automatically by appropriate triggers. Temporal databases are studied in detail in two case studies of this series, *Temporal databases - Part 1* and *Part 2*.

8.12 Building a business application

This application is of a nature different from those we have developed so far. While the latter provide some sort of ancillary services for the benefit of higher level application, this one is an active database that implements a full scale business application (except the graphical user interface of course).

8.12.1 About the case

We choose a business application that is sufficiently simple to be described in less than half a page, but sufficiently rich to require the implementation of some non-trivial rules.

This application is to support some basic business processes of a small retail shop that sells building materials to private customers. We identify three actors in this scenario: the *customer*, the *retail shop employee* and the *supplier*(s).

The responsibilities of the customer are simple: placing orders and executing payments.

An employee of the shop receives orders from customers and encodes them *in the computer*. A customer order specifies a quantity of a definite item. If this item is physically available in this quantity, the order is executed, an invoice is printed and sent, and the customer is invited to take away the materials ordered. When the payment of the invoice is received, the order is said to be closed. If there is not enough of this item in the inventory, the order is suspended, the employee writes a purchase order and sends it to the supplier which proposes the best offer for this item: i.e., the lowest price, and in case of tie, the shortest delivery time.

From time to time, each supplier examines the purchase orders it has received for a definite item and decides to execute them. On the retail shop side, this replenishment allows pending orders to be executed and invoices to be sent.

From time to time too, customers pay their invoices, which closes their orders.

18. Actually, true temporal database also record future states.

Since we concentrate our discussion on the retail shop management, we assume that the inventories of suppliers are infinite so that their items never become out of stock. We also consider that the items as viewed by the retail shop also are those viewed by the suppliers.

8.12.2 About the case study

We intend to develop this application according to two different paradigms that can be abstractly described as *3-tier* and *2-tier architectures*.[19]

In a standard **3-tier architecture**, the application is distributed in three components, that usually reside on three distinct hardware/software platforms: the *database*, the *business logic* and the *presentation* (or *client*). See Figure 8.2, left.

The *database* comprises all the persistent data required to allow the business to work. The *business logic* is made up of the application program(s) that collect, check, transform, process, compute, store, retrieve the data exchanged with the clients and with the database. The *presentation* component is in charge of managing the user dialogues through a GUI, either proprietary (such as through a specific app in a smartphone) or generic (such as a web browser).

Theoretically, the database platform, generally called the *database server*, requires a specific hardware architecture comprising large, fast external data stores. The application programs will generally reside on a CPU intensive *application server*, with many-core processors and large RAM, while the *presentation* component will run on a mobile device or on a standard PC. The communication channels between the components are expected to be as fast as possible. Actually, one must distinguish the logical architecture (the three components) from the physical one (the platforms). A 3-tier architecture can be distributed on three platforms, as described above, but also on a single machine, in which case the communications are particularly fast. At the opposite, the database can be distributed and replicated among several machines, to improve the availability and security of the data.

In a **2-tier architecture**, two of these components are merged. For instance, the database component and the application programs are integrated. This integration can be understood in two ways. First, the application programs *absorb* the database component; it accesses the data directly, without the intermediary of a database server. Programs accessing SQLite databases are of this kind.[20] The second interpretation is much more interesting: the application programs *are absorbed* by the database, which becomes an **active database**. This variant of application only comprises the *presentation* and *database* components (Figure 8.2, right).

This is the architecture we will explore in this case study. We first study, in an informal and intuitive way, the sequence of actions that must be performed whenever customer orders are encoded in the database, customers execute their payments

19. https://en.wikipedia.org/wiki/Multitier_architecture
20. The SQLite DBMS, that is a function library, is said *serverless*.

and suppliers replenish out of stock items, irrespective of the technique that could be applied to implement these actions.

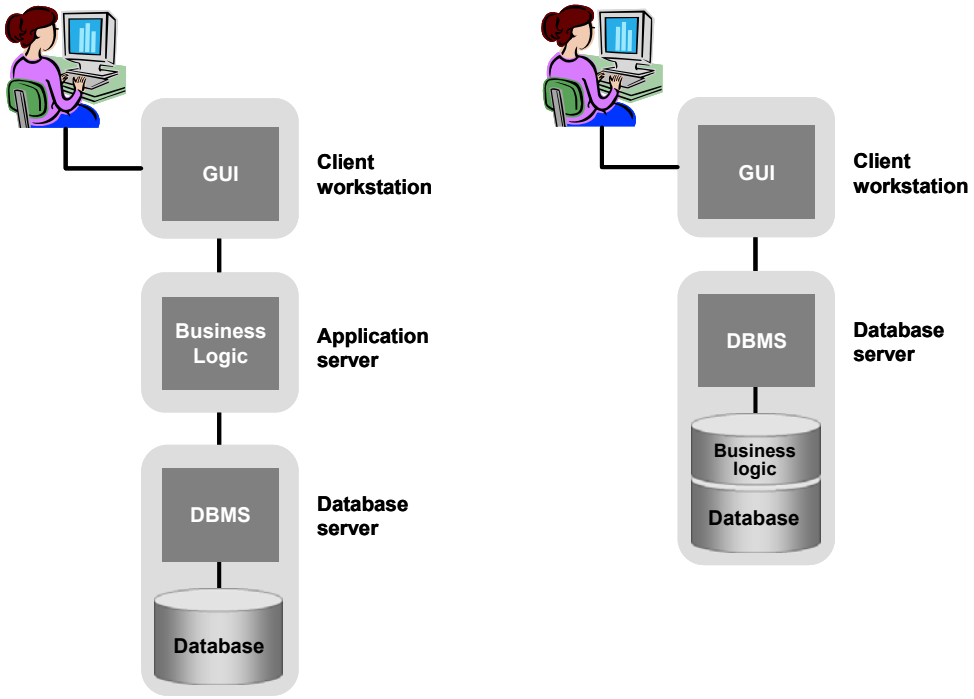


Figure 8.2 - Symbolic representation of 3-tier (left) and 2-tier (right) architecture

8.13 The static structure of the database

The short description developed above suggests that the application domain (the part of the world we are interested in) comprises seven kinds of core entities: the *customers*, the *items*, the *suppliers*, the *offers* (the items the suppliers propose to sell, with their price and delivery time), the *customer orders*, the *customer invoices* and the *supplier orders*.

The first four categories are *basic entities*, that define the intrinsic static structure of the shop. These entities evolve slowly. For example, the life span of a customer is measured in years. Although new items may appear at any time, we do not expect more than 10% of them to be introduced in the inventory each year.

The last three entity categories are *transactional entities* that describe the operations that take place in the application domain. They evolve rapidly, most of them living only two or three days.

To record data about these entities, the database comprises four *base tables* and three *transaction tables*.

8.13.1 The base tables

The *base tables* describe stable entities: the customers (CUSTOMER), the items sold by the retail shop (ITEM), the suppliers (SUPPLIER) and the offers (table OFFER).

Table CUSTOMER

Table CUSTOMER stores identification and contact data of customers (columns CustID, Name, Address and City) plus the current balance of each customer account (column Account). Once an invoice has been sent, this value is decreased by the amount of this invoice. The balance is negative if some invoices still are to be paid.[21]

```
create table CUSTOMER (
        CustID  char(10) not null,
        Name    char(32) not null,
        Address char(60) not null,
        City    char(30) not null,
        Account decimal(9,2) not null,
        primary key (CustID));
```

Table ITEM

The rows of table ITEM describe the items of the retail shop that the customers may buy.

```
create table ITEM (
       ItemID      char(15) not null,
       Description char(60) not null,
       Price       decimal(4,2) not null,
       SuppID      char(10),
       SuppPrice   decimal(4,2),
       QonHand     integer not null,
       Qord        integer default 0,
       Qavail      integer not null,
       Qeco        integer not null,
       constraint PKITEM primary key (ItemID));
```

In addition to its Id (column ItemID) and description (column Description), each item is characterized by the following properties.

21. In this small case study, we do not consider purchase *reimbursement*. Therefore, the balance will never be positive.

- Price: the current customer unit price of the item.
- SupplD: Id of the reference supplier of the item. The reference supplier is selected as the one that provides the best offer for the item (lower price and lower shipment time). This choice is re-evaluated each time the item is replenished.
- SuppPrice: the unit price requested by the reference supplier to replenish the inventory.
- QonHand: the *quantity on hand* is the quantity that **physically exists** in the inventory of the retail shop. If a customer order requests a quantity **qty** not greater than QonHand, it is immediately executed. Otherwise, the order is *pending*, waiting for the replenishment the reference supplier will execute. In both cases, QonHand is decreased by **qty**. This means that QonHand may be negative.
- Qord: the *quantity on order* is the total quantity that has been ordered to the reference supplier *but not shipped yet*. When a customer order, the quantity qty of which cannot be satisfied by the quantity on hand (qty > QonHand) nor by the future replenishment (qty > Qavail), then an order is issued for a certain quantity **mQeco** (explained below) and sent to the reference supplier. Qord is then augmented by mQeco.
- Qavail: the *available quantity* is the sum of QonHand and Qord. This is the quantity that can be consumed to execute future customer orders without the need to place a new replenishment order with the reference vendor. If a customer order requests more than the quantity on hand but no more than the available quantity (QonHand < Qty ≤ Qavail), it will be executed as soon as the pending supplier orders are executed.
- Qeco: this is a simplified variant of the *economic order quantity*, that is, the optimal quantity[22] to be ordered from the reference supplier to replenish the item inventory. Actually, the quantity **mQeco** requested from the supplier is a *multiple* of Qeco computed as follows:

  mQeco = m x Qeco, m being the smallest integer such that

  Qavail + mQeco > 0.

Table SUPPLIER

Table SUPPLIER describes the suppliers of the retail shop. It plays no role and is mainly symbolic.

22. In real inventory management, this quantity is computed so that the total cost of replenishment is minimal. See https://en.wikipedia.org/wiki/Economic_order_quantity for more detail.

```
create table SUPPLIER (
       SuppID  char(10) not null,
       Name    char(32) not null,
       City    char(30) not null,
       primary key (SuppID));
```

Table OFFER

Table OFFER describes, for each supplier (SuppID) and for each item this supplier can supply (ItemID) its unit price (Price) and its delivery time in days (Delay).

```
create table OFFER (
       SuppID  char(10) not null,
       ItemID  char(15) not null,
       Price   decimal(4,2) not null,
       Delay   integer not null,
       primary key (SuppID,ItemID),
       foreign key (SuppID) references SUPPLIER,
       foreign key (ItemID) references ITEM);
```

8.13.2 The transaction tables

The *transaction tables* describe the documents processed by the employees of the retail shop: the customer orders (CUSTORDER), the invoices sent to the customers (CUSTINVOICE) and the orders sent to the suppliers (SUPPORDER).

The entity states

Before translating these entities into database structures, we examine the timeline of the life of transactional entities, from their initial recording to their closure. Their lives comprise a sequence of states.

The first state of a **customer order** is *'recorded'*. If the item requested is available in sufficient quantity (Qty ≤ QonHand), an invoice is created and the state of the order is *'invoiced'*. When the customer has made the corresponding payment, the order is in the *'closed'* state. However, if there is no sufficient quantity (Qty > QonHand), the order is in the *'pending'* state, followed, later by *'invoiced'* and *'closed'* states.

The first state of a **customer invoice** is *'recorded'*. Then, if the order is in *'invoiced'* state, the invoice is set in *'sent'* state. If the order is in *'pending'* state, the state of the invoice also is *'pending'*. When the customer has made the payment, the invoice is in the *'paid'* state.

The sequence of states of a **supplier order** is simpler: *'recorded'*, *'assigned'* when the supplier with the best offer has been identified, and *'closed'* when the order has been executed.

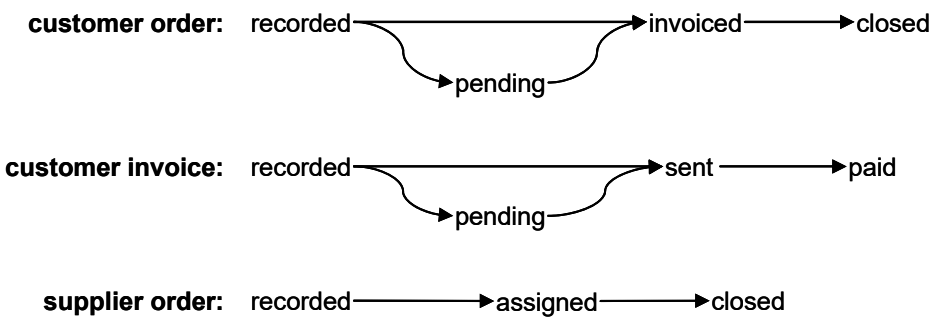


Figure 8.3 - Diagram of state transitions of transactional entities

Table CUSTORDER

Table CUSTORDER contains the data of the customer orders received so far. In addition to the Id (column OrdID) and date received (column DateOrd) of each order, its row specifies:

– CustID: the reference of the customer who placed the order.

– ItemID: the reference of the item ordered.

– Price: the unit price of the item *at the time the order has been registered*. This column is nullable; it will be updated after the row has been inserted.

– Qty: the quantity requested of the item.

– State: the current state of the order, among the following: *'recorded'*, *'invoiced'*, *'closed'*, *'pending'*.

```
create table CUSTORDER (
       OrdID   integer  not null,
       DateOrd date     not null,
       CustID  char(10) not null,
       ItemID  char(15) not null,
       Price   decimal(4,2),
       Qty     integer  not null,
       State   char(20) not null,
       primary key (OrdID),
       foreign key (CustID) references CUSTOMER,
       foreign key (ItemID) references ITEM);
```

Table CUSTINVOICE

Table CUSTINVOICE contains the data of the invoices generated for each customer order as soon as the latter has been processed. Since each invoice is attached to an order, it shares most of its data with the latter.

These data include the Id of the invoice (column InvID), the date it has been generated (column DateInv). In addition, they comprise, for each invoice:

- OrdID: the reference of the customer order with which it is associated.
- CustID: the reference of the customer who placed the order (and to whom the invoice will be [or has been] sent)
- ItemID: the reference of the item ordered.
- Price: the unit price of the item (same as in the CUSTORDER row).
- Qty: the quantity requested of the item.
- State: the current state of the invoice, among the following: *'recorded'*, *'sent'*, *'paid'*, *'pending'*.

```
create table CUSTINVOICE (
       InvID   integer  not null,
       DateInv date not null,
       OrdID   integer  not null,
       CustID  char(10) not null,
       ItemID  char(15) not null,
       Price   decimal(4,2) not null,
       Qty     integer  not null,
       Amount  integer  not null,
       State   char(20) not null,
       primary key (InvID),
       foreign key (OrdID) references CUSTORDER);
```

Table SUPPORDER

Each row of table SUPPORDER describes an order issued by the retail shop to replenish an item that is out of stock. This order is sent to the reference supplier of this item. A supplier order is sent for each customer order that cannot be executed due to insufficient quantity available (Qavail).
In addition to the Id (column OrdID) and date sent (column DateOrd) of each order, its row specify:

- CustID: the reference of the customer who placed the order.
- ItemID: the reference of the item ordered.
- SuppID, Price: the supplier Id and the unit price of the item as specified by the best offer.
- Qty: the quantity requested of the item.
- State: the current state of the order, among the following: *'assigned'*, *'closed'*.

```
create table SUPPORDER (
       OrdID   integer not null,
       DateOrd date not null,
       ItemID  char(15) not null,
       SuppID  char(10) not null,
       Price   decimal(4,2) not null,
       Qty     integer  not null,
       State   varchar(20) not null,
       primary key (OrdID),
       foreign key (SuppID,ItemID) references OFFER);
```

Script 8.52 - Schema of the last of the seven tables of the database

8.13.3The initial state of the database

New, we will observe how the data evolve when we execute a series of representative transactions according to the scenario described in the next sections. To make things concrete, we consider that the base tables contain the initial data shown in Appendix A. The subset of these data that will actually be used by the transactions are depicted in Figure 8.4.

```
CUSTOMER CustID Name    Address               City      Account
======== ====== ======= ==================== ========= =======
         ...
         C400   NEUMANN 454, Kirchenstraße   Berlin    0
         ...
         K111   JANSEN  5B, Grote Halstraat  Amsterdam 0
         ...

ITEM ItemID ...  Price SuppID SuppPrice QonHand Qord Qavail Qeco
==== ====== ==== ===== ====== ========= ======= ==== ====== ====
     ...
     PA45   ...  48    --     --        80      0    80     80
     ...

SUPPLIER SuppID Name         City
======== ====== ============ =========
         ...
         F-725  BricoMat     Paris
         ...

OFFER SuppID ItemID Price Delay
===== ====== ====== ===== =====
      ...
      D-109  PA45   35    3
      E-034  PA45   37    1
      F-725  PA45   35    1
      N-601  PA45   39    4
      ...    ...    ...   ...
```

Figure 8.4 - Initial state of OFFER table (limited to item PA45)

8.13.4 The transactions

We first introduce and process some customer orders. Then, we will let the supplier ship the items requested by the retail shop. Finally, the customers will pay the invoices they have been sent.

A first customer order

This first customer order is placed by German customer C400 (*NEUMANN*). It requests 60 units of item PA45 (*Steel nails of 45 mm*). This order is encoded as row **1** in table CUSTORDER (Figure 8.5).

```
CUSTORDER OrdID DateOrd    CustID ItemID Price Qty State
========= ===== ========== ====== ====== ===== === ========
          1     2020-03-27 C400   PA45   48    60  invoiced
```

Figure 8.5 - Customer C400 has ordered 60 units of item PA45

The customer price (48) of the item has been inserted into this row. Since the quantity on hand of this item is 80 (Figure 8.4), this order **can be executed immediately**. Therefore:

- an invoice has been created in the database (Figure 8.6),
- this invoice has been printed and sent (Figure 8.7); so, its state is set to 'sent' (Figure 8.6),
- as a natural consequence, the state of the order is set to 'invoiced' (Figure 8.5),

```
CUSTINVOICE InvID DateInv    OrdID CustID ItemID Price Qty Amount State
=========== ===== ========== ===== ====== ====== ===== === ====== =====
            1     2020-03-27 1     C400   PA45   48    60  2880   sent
```

Figure 8.6 - The invoice as it has been created in the database

```
INVOICE no 1
--------------------
 CUSTOMER: C400
     Date: 2020-03-27
     Item: PA45
      Qty: 60
    Price: 48
--------------------
    Total: 2,880
```

Figure 8.7 - The invoice has been printed (simplified layout) and sent to the customer

In table ITEM, the row describing item PA45 has been updated: the quantity of the order (Qty = 60) is subtracted from the *quantity on hand* of the item (80 - 60 = 20), and from its *quantity available* (Figure 8.8).

ITEM	ItemID	...	Price	SuppID	SuppPrice	QonHand	Qord	Qavail	Qeco
	...	...	...	...	...	...	...	...	...
	PA45	...	48	--	--	20	0	20	80
	...	...	...	...	...	...	...	...	...

Figure 8.8 - The new state of table ITEM

Finally, we have to notify that the customer's account is decreased by the amount of the invoice, i.e., 2,880 (Figure 8.9).

CUSTOMER	CustID	Name	Address	City	Account
	...	...	...	...	...
	C400	NEUMANN	454, Kirchenstraße	Berlin	-2880
	...	...	...	...	...

Figure 8.9 - Since the order has been executed and the invoice has been sent, the account level of the customer is updated

Another customer order

We decide to process a second order, from customer K111 (*JANSEN*), that also specifies item PA45 of which it requests 35 units. Since there are only 20 units left of PA45, this order **cannot be executed**. It is created with a *pending* state. An invoice is also created with the same *pending* state (Figure 8.10).

CUSTORDER	OrdID	DateOrd	CustID	ItemID	Price	Qty	State
	1	2020-03-27	C400	PA45	48	60	invoiced
	2	2020-03-27	**K111**	**PA45**	**48**	**35**	**pending**

CUSTINVOICE	InvID	DateInv	OrdID	CustID	ItemID	Price	Qty	Amount	State
	1	2020-03-21	1	C400	PA45	48	60	2880	sent
	2	**2020-03-27**	**2**	**K111**	**PA45**	**48**	**35**	**1680**	**pending**

Figure 8.10 - Customer order n° 2 cannot be executed and is left *pending* as well as its associated invoice

The quantity on hand of item PA45 becomes negative (20 - 35 = -15), as well as its quantity available, so, we decide to replenish the stock of item PA45 by sending an order to a supplier. The quantity requested (Qord) is the *economic order quantity* of this item (Qeco = 80), which will be sufficient to replenish the stock. The question is, which supplier will be chosen? First, of course, the suppliers offering the lowest price are selected. There are two contenders, namely D-109 and F-725, both with a price of 35 (Figure 8.4). From them, we choose the offer that guarantees the shortest delivery time, F-725, to which the order is assigned. This *best offer* is translated into

the data of Figure 8.11. A supplier order is created. Its components are shown in Figure 8.12.

ITEM	ItemID	...	Price	SuppID	SuppPrice	QonHand	Qord	Qavail	Qeco
	...	...	...	...	...	...	...	...	...
	PA45	...	48	F-725	35	-15	80	65	80
	...	...	...	...	...	...	...	...	...

Figure 8.11 - 80 units of P45 have been ordered according to the best offer.

SUPPORDER	OrdID	DateOrd	ItemID	SuppID	Price	Qty	State
	1	2020-03-27	PA45	F-725	35	80	assigned

Figure 8.12 - The supplier order has been assigned to supplier F-725

About next customer orders

If the next customer order specifies a quantity **qty** of PA45 not greater than 65 units, it could be executed once the supplier order is executed (theoretically the next working day according to the data of Figure 8.4). In this case, there is no need to issue another supplier order. On the contrary, if this quantity is greater than 65 a new supplier order of 80 units, so that Qord is increased to 160.

On the supplier side

Now, let us go and see what happens on the supplier side once the first supplier order has been received (that depicted in Figure 8.12). This order asks supplier F-725 to replenish item PA45. When this request is satisfied, 80 units of this item are sent to the retail shop to replenish the PA45 stock.

The ITEM row is changed as follows (Figure 8.13):

– the quantity on hand is now QonHand = -15 + 80 = 65 units

– the quantity ordered is reset to 0

– the available quantity does not change (65).

ITEM	ItemID	...	Price	SuppID	SuppPrice	QonHand	Qord	Qavail	Qeco
	...	...	...	...	...	...	...	...	...
	PA45	...	48	F-725	35	65	0	65	80
	...	...	...	...	...	...	...	...	...

Figure 8.13 - Item PA45 has been replenished

The supplier order has been successfully processed and can be closed (Figure 8.14).

SUPPORDER	OrdID	DateOrd	ItemID	SuppID	Price	Qty	State
	1	2020-03-27	PA45	F-725	35	80	**closed**

Figure 8.14 - The supplier order is closed

Item PA45 has been replenished, what next?

Now, we go back to the retail shop, where the second customer order can be executed, with the following consequences (Figure 8.15):

- its *pending* invoice is sent to the customer,
- the customer order is declared invoiced ,
- the Account value of customer K111 is changed to -1680.

CUSTINVOICE	InvID	DateInv	OrdID	CustID	ItemID	Price	Qty	Amount	State
	1	2020-03-27	1	C400	PA45	48	60	2880	sent
	2	**2020-03-27**	**2**	**K111**	**PA45**	**48**	**35**	**1680**	**sent**

CUSTORDER	OrdID	DateOrd	CustID	ItemID	Price	Qty	State
	1	2020-03-27	C400	PA45	48	60	invoiced
	2	2020-03-27	K111	PA45	48	35	**invoiced**

CUSTOMER	CustID	Name	Address	City	Account
	...	...	...	...	...
	C400	NEUMANN	454, Kirchenstraße	Berlin	-2880
	...	...	...	...	...
	K111	JANSEN	5B, Grote Halstraat	Amsterdam	**-1680**
	...	...	...	...	...

Figure 8.15 - Processing pending customer orders and invoices after item replenishment

Finally, customer C400 pays his invoice

The last operation to consider is the responsibility of the customer: paying for the materials she has been delivered. Let us examine the case of customer C400, who ordered, and took away, 60 units of item PA45, and for which he has to pay an amount of 2,880. When this payment has been executed, the employee of the retail shop encodes it *in the computer*. This operation triggers three data modifications (Figure 8.16):

- the invoice state is set to 'paid'
- the customer order is closed

– and the account level of the customer is increased (becomes less negative) by the amount of the payment.

```
CUSTINVOICE InvID DateInv    OrdID CustID ItemID Price Qty Amount State
=========== ===== ========== ===== ====== ====== ===== === ====== =====
            1     2020-03-27 1     C400   PA45   48    60  2880   paid
            2     2020-03-27 2     K111   PA45   48    35  1680   sent

 CUSTORDER OrdID DateOrd    CustID ItemID Price Qty State
 ========= ===== ========== ====== ====== ===== === ========
           1     2020-03-27 C400   PA45   48    60  closed
           2     2020-03-27 K111   PA45   48    35  invoiced

 CUSTOMER CustID Name    Address               City      Account
 ======== ====== ======= ==================== ========= =======
          ...    ...     ...                  ...       ...
          C400   NEUMANN 454, Kirchenstraße   Berlin    0
          ...    ...     ...                  ...       ...
```

Figure 8.16 - Registering the payment of the first invoice

This scenario is great for intuitively understanding the principles of the mechanisms we will have to implement. However, to translate them into code, we would need a more precise definition of the operations. In order not to make this presentation too cluttered, we have moved these definitions to Appendix B.

8.14 Implementation of the classical application architecture

We can ignore the standard ancillary functions for registering, modifying and deleting data of basic tables (CUSTOMER, ITEM, SUPPLIER and OFFER). On the one hand, they do not depend on the application architecture and, on the other hand, their code is quite simple. So, we can concentrate the discussion on the components and algorithms of the standard 3-tier architecture.

The GUI comprises three modules that collect the data required to register a *customer order*, a *customer payment* and the execution of *supplier orders* to replenish the item stock of the retail shop. These modules are implemented by three SQLfast scripts (Figure 8.17):

– Register-CUSTORDER-GUI.sql (Script 8.53 and Figure 8.18),

– Register-PAYMENT-GUI.sql,

– Replenish-ITEM-GUI.sql.

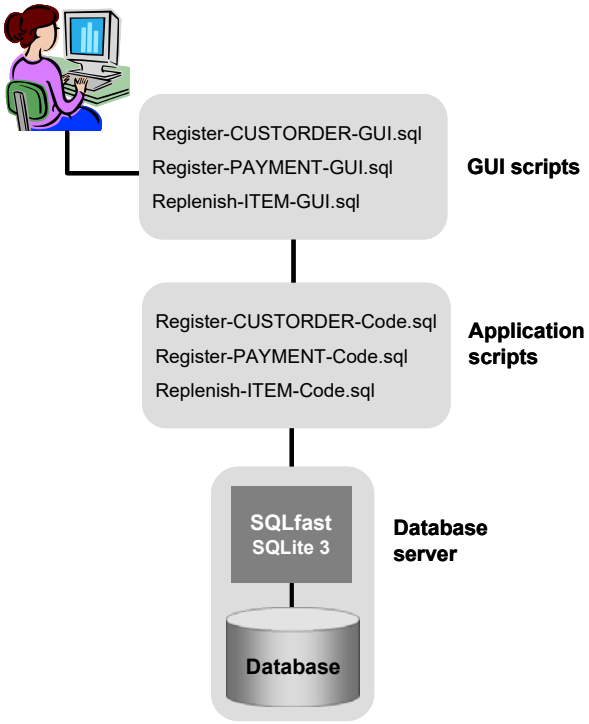


Figure 8.17 - The standard implementation of the 3-tier architecture

```
extract oid = select coalesce(max(OrdID),0)+1 from CUSTORDER;
set dat = $date$;
ask-u oid,dat,cus,itm,qty = [/bRegister a new customer order]
         Order ID:
        |Current date:
        |Customer ID:[select Name||' ('||CustID||')',CustID
                      from   CUSTOMER order by CustID]
        |Item ID:[select Description||' ('||ItemID||')',ItemID
                  from   ITEM order by ItemID]
        |Quantity:;
execSQL Register-CUSTORDER-Code.sql;
```

Script 8.53 - Dialogue box to register a new customer order (simplified version)

The first script computes the next order Id (or 0 for the very first order), extracts the current date, execute the dialogue box, then call application script RETAIL-Register-CUSTORDER-Code.sql that implements in a straightforward way the business logic developed in Section 8.24.2 of Appendix B. Considering the emphasis of this document on active database implementation, the code of Register-CUSTORDER-

Code.sql is not shown here but can be examined in directory SQLfast\Scripts\Case-Studies\Case_Active_DB\Standard-Version.

The other GUI modules and application scripts are coded in a similar way.

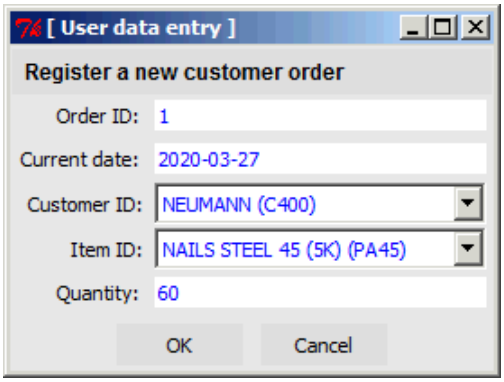


Figure 8.18 - Collecting the data to register a new customer order

8.15 Implementation as an active database

The basic philosophy of the standard architecture is to decompose a complex task into simpler tasks, that typically translates into a hierarchy of procedures in which parent procedures call their children procedures for execution.

The principle of active databases is quite different. It consists in creating events that send some kind of messages to target objects materialized as rows in the tables.[23] For example, when the payment of an invoice has been received, a message is sent to this invoice by changing its state from 'sent' to 'paid'. As soon as the invoice receives this message (by catching the State update event), it closes its order and updates the account of its customer. In other words, each object is responsible for the consequences of its evolution.

The architecture specific to our case study is shown in Figure 8.19. The GUI comprises three scripts quite similar to those of the standard architecture. The only difference is that, instead of calling complex application scripts, they just send messages to database objects by executing elementary data modification queries.

8.16 Extending the data structures

The main components of our active database are the **triggers** that are responsible for applying the rules of the business logic elaborated in Appendix B.[24] Our discus-

23. Somewhat similar to object-oriented and agent-based approaches.

sion will of course focus on their development. However, there is some opportunities to alleviate their work by automating some checking and computing operations through `default`, `check` and `generated` clauses. Some of them are shown in Script 8.54.

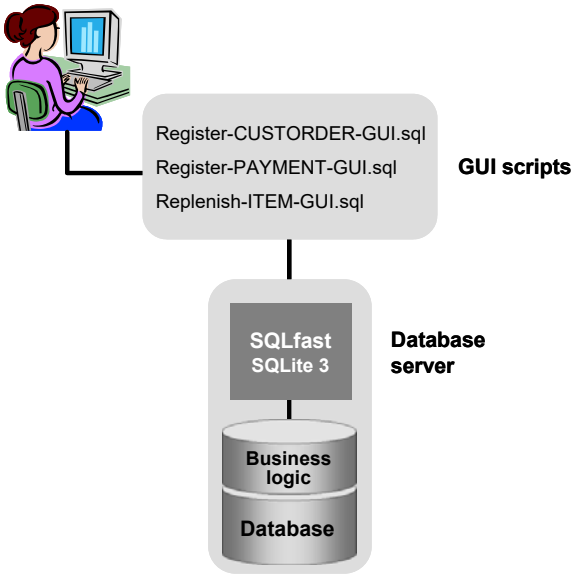


Figure 8.19 - The 2-tier architecture implementation as an active database

8.17 The user interface

The three transaction operations, namely *registering a customer order*, *registering the payment of a customer invoice* and *replenishing the stock of an item* by a supplier are carried out through the same dialogue boxes that collect data from the user. Then, they use them to update the database:

- *Registering a customer order*: a new row is inserted in CUSTORDER table from the data collected from the user (Script 8.55).
- *Registering a payment*: the user provides the Id of an invoice, the state of which is set to 'paid' (Script 8.56).
- *Replenishing the stock of an item*: the user identify all the *assigned* (that is, not *closed* yet) supplier orders of an item to a supplier. The state of this order is set to 'closed' to indicate that the quantity ordered has been shipped (Script 8.57).

24. *Stored procedures* can also be used to encapsulate some critical rules. However, since the DBMS on which our case study will be built does not offer these constructs, they will be ignored.

```
create table CUSTOMER (
        ...,
        Account decimal(9,2) not null default 0.0,
        ...);
create table ITEM (
       ...,
       Qavail  integer not null
               generated always as (QonHand+Qord),
       ...);
create table CUSTORDER (
       ...,
       State   char(20) not null default 'recorded',
       ...,
       check(Qty > 0));
create table CUSTINVOICE (
       ...,
       Amount  integer  not null
               generated always as (Qty*Price),
       State   char(20) not null default 'recorded',
       ...);
create table SUPPORDER (
       ...,
       State   varchar(20) not null default 'recorded',
       ...);
```

Script 8.54 - Augmenting table structures with *default*, *check* and *generated* clauses

```
ask-u oid,dat,cus,itm,qty = <see Script 8.53>;
insert into CUSTORDER(OrdID,DateOrd,CustID,ItemID,Qty)
       values ($oid$,'$dat$','$cus$','$itm$',$qty$);
commitDB;
```

Script 8.55 - Dialogue box to register a new customer order (simplified version)

```
ask inv = [/bRegister a payment]
    Invoice:[select InvID||': '||CustID||', '||DateInv,InvID
             from   CUSTINVOICE
             where  State = 'sent' order by InvID];
update CUSTINVOICE
set    State = 'paid'
where  InvID = $inv$;
commitDB;
```

Script 8.56 - Dialogue box to register the payment of an invoice (simplified version)

```
ask itm = [/bSelect an item to replenish]
           Item:[select distinct ItemID from SUPPORDER
                 where  State = 'assigned' order by ItemID];
ask sup = [/bSelect a supplier of "$itm$"]
           Supplier:[select distinct SU.SuppID||' ('||SU.Name
                                    ||')' ,SU.SuppID
                     from   SUPPORDER SO,SUPPLIER SU
                     where  SO.SuppID = SU.SuppID
                     and    ItemID = '$itm$'
                     and    State = 'assigned'
                     order by SU.SuppID];
update SUPPORDER
set    State = 'closed'
where  SuppID = '$sup$' and ItemID = '$itm$'
and    State = 'assigned';
commitDB;
```

Script 8.57 - Dialogue box to execute supplier orders (simplified version)

The actions of the user interface merely consist in sending messages to objects (i.e., table rows). They are symbolized in Figure 8.20.

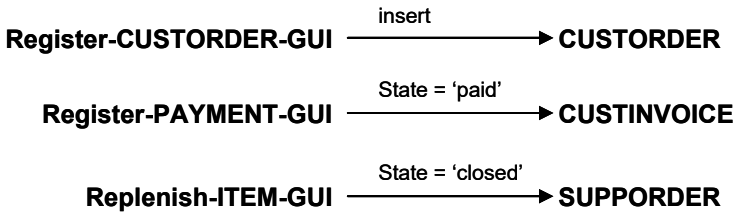


Figure 8.20 - The GUI scripts send messages through data modification events

Now, the real work is performed by triggers that react to these messages by carrying out the data processing operations implementing the business tasks.

8.18 The transactions

Now we examine how the logic of the three main transactions are translated into a trigger system.

8.18.1 Registering a customer order

When a CUSTORDER row has been inserted in the database, the first question is, does the stock contain enough of the item to execute this order (**Case 1** in Appendix B, Section 8.24.2), or not (**Case 2** in the same section)?

We find it clearer to create a trigger for each branch of this alternative (Script 8.58).

```
create trigger TRG_CORD_INS1
after insert on CUSTORDER
when new.Qty <= (select QonHand from ITEM
                 where  ItemID = new.ItemID)
begin ... end;

create trigger TRG_CORD_INS2
after insert on CUSTORDER
when new.Qty > (select QonHand from ITEM
                where  ItemID = new.ItemID)
begin ... end;
```

Script 8.58 - Controlling CUSTORDER insert operations - First attempt

Beware of side effects!

As a general rule, the conditions of the `when` clauses of these triggers (let us call them **c1** and **c2**) must be *exclusive* and *covering*:

- exclusive: **c1** and **c2** = False
- covering: **c1** or **c2** = True

In other words, when a new CUSTORDER row is inserted, one and only one of these triggers will fire. Unfortunately, as they are expressed in Script 8.58, these triggers may not always execute as expected. The analysis of Section 8.24.2 shows that, in **Case 1** and **Case 2**, the value of QonHand of the referenced ITEM row is decreased by that of Qty of the order. Indeed, we read, in both cases:

- update referenced ITEM row **it**:
  subtract co.Qty from it.QonHand

Let us consider that an `insert` occurs when condition Qty ≤ QonHand is met. The database engine selects and fires one of the `after insert` triggers that meet this condition, that is, TRG_CORD_INS1. During the execution of its body, this trigger decreased the value of QonHand of the referenced ITEM row. When this execution is finished, the database engine examines one of the other `after insert` triggers, i.e., TRG_CORD_INS2, and evaluates its condition.

This is where the aforementioned problem may arise: if, initially, Qty ≤ QonHand < 2 * Qty, then the first trigger assigns to QonHand a new value such that QonHand <

Qty, which means that the **`when` condition of the second trigger is also met**. Therefore, both triggers will fire!

The pattern of the problem is easy to identify: the body of TRG_CORD_INS1 modifies the values used in the `when` clause of TRG_CORD_INS2, in such a way that conditions **c1** and **c2** no longer satisfy the *exclusive* condition.

This problem can be solved in two ways. First we can encapsulate all the different cases in a single (large and complex) trigger. Secondly, we refine the `when` conditions in such a way that they remain exclusive, whichever the order in which they are executed. Let us choose the last solution.

We examine the operations of **Case 1** and **Case 2** and we observe that both create an invoice. So we modify the `when` condition as follows: each trigger will fire if the `when` condition of Script 8.58 is met and *no invoice has been created yet*. This means that, whatever the order in which the triggers are considered for execution by the SQL engine, only the first one will fire.

Now, the triggers of Script 8.59 work as expected.

Important remark

The behavior of the trigger system of a database depends on the execution model implemented by the DBMS. Theoretically, when an event occurs, the SQL engine identifies all the candidate triggers that this event may fire. From them, only those that satisfy the `when` condition will actually fire. This leads to two questions: in which order will these triggers fire and when will their `when` condition be evaluated?

Most DBMS allow the database designer to specify this order, in a deterministic way, through various means. Some others, SQLite being one of them, leave the order undefined so that the designer must resort to some adhoc trick to force a definite execution order. This is what we have done by checking the absence of a dependent CUSTINVOICE row.

The question of the `when` condition is more delicate. According to a first model, all the conditions of the candidate triggers are evaluated before any of them fires. In this model, there is no side effect and the *exclusive* property of conditions Case 1 and Case 2 is guaranteed by construction. In another model, the condition of the first candidate trigger is evaluated, and if *True*, its body is executed. After that, the SQL engine consider another trigger, evaluates its condition, and, if *True*, executes its body. This process goes on until all the candidate triggers have been examined. So, the value of the condition of a trigger depends on the execution of the triggers executed before it, which leads to the undesirable side effect described in this section. The execution model of SQLite is of this kind.

```
create trigger TRG_CORD_INS1
after insert on CUSTORDER
when new.Qty <= (select QonHand from ITEM
                 where  ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where  OrdID = new.OrdID)
begin ... end;

create trigger TRG_CORD_INS2
after insert on CUSTORDER
when new.Qty > (select QonHand from ITEM
                where  ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where  OrdID = new.OrdID)
begin ... end;
```

Script 8.59 - Controlling CUSTORDER insert operations - Final version

Local variables

Triggers of CUSTORDER tables will use four local variables. They are simulated by technical table V_CORDER:

```
create table V_CORDER(Price,Qavail,mQeco,InvID);
```

Controlling CUSTORDER row in Case 1

Script 8.60 shows the code of **TRG_CUSTORDER_INS1**, the trigger responsible for the propagation of data modification due to the insertion of a new CUSTORDER according to **Case 1.**, that is, when the quantity on hand is sufficient to immediately execute the order. It uses two local variables: Price, the customer unit price of the item, and InvID, the Id of the next invoice. It comprises five queries:

1. The first query (`insert`) stores in V_CORDER the current values of the local variables.
2. The next query (`update`) subtract the quantity ordered from the quantity on hand.
3. Then, the CUSTORDER row just inserted is completed (`update`) with the unit price of the item.
4. The fourth query creates the invoice (`insert`) and initializes its state to 'send'. This latter setting is a message that instructs the target row to execute a series of actions deriving from its new state. These actions are specified in Script 8.64.
5. Finally, the local variables are reset.

```
create trigger TRG_CUSTORDER_INS1
after insert on CUSTORDER
for each row
when new.Qty <= (select QonHand from ITEM
                 where ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where OrdID = new.OrdID)
begin
   insert into V_CORDER(Price,InvID)
       select Price,
              (select coalesce(max(InvID),0)+1 from CUSTINVOICE)
       from   ITEM where ItemID = new.ItemID;
   update ITEM
   set   QonHand = QonHand - new.Qty
   where ItemID = new.ItemID;
   update CUSTORDER
   set    Price = (select Price from V_CORDER)
   where  OrdID = new.OrdID;
   insert into CUSTINVOICE(InvID,DateInv,OrdID,CustID,ItemID,
                           Price,Qty,State)
       values ((select InvID from V_CORDER),new.DateOrd,
                new.OrdID,new.CustID,new.ItemID,
               (select Price from V_CORDER),new.Qty,'sent');
   delete from V_CORDER;
end;
```

Script 8.60 - Trigger that controls the creation of a customer order when the quantity on hand is sufficient

Controlling CUSTORDER row in Case 2

The task of trigger **TRG_CUSTORDER_INS2** is to cope with the cases of insufficient quantity on hand, according to the rules of **Case 2.** It uses four local variables: Price, the customer unit price of the item, Qavail the quantity available without issuing a new supplier order, mQeco, the quantity that will be needed to replenish the stock of the item (the smallest multiple of Qeco such that Qavail will become positive), and InvID, the Id of the next invoice. It comprises the following queries (Script 8.61):

1. The first query (`insert`) stores in V_CORDER the current values of the local variables.
2. The next query (`update`) subtract the quantity ordered by the customer from the quantity on hand (QonHand) of the ITEM row.
3. A block of three queries is executed *if the quantity available is not sufficient*.[25]
   - The quantity to order from the supplier (mQeco) is added to the quantity on order (Qord) of the ITEM row.

25. For the current SQLfast version, that uses the SQLite DBMS, this conditional block must be transformed according to the rules of Section 8.2.7.

- A best offer for the item is identified if none has been selected yet (`SuppID is null`) and notified (`update`) in the ITEM row.
- A supplier order is created by inserting a row in SUPPORDER table.

```
create trigger TRG_CUSTORDER_INS2
after insert on CUSTORDER
for each row
when new.Qty >  (select QonHand from ITEM
                 where ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where OrdID = new.OrdID)
begin
   insert into V_CORDER(Price,Qavail,mQeco,InvID)
       select Price,Qavail,
              Qeco*(truncate(1.*(new.Qty-Qavail)/Qeco)+1),
              (select coalesce(max(InvID),0)+1 from CUSTINVOICE)
       from   ITEM where ItemID = new.ItemID;
   update ITEM
   set   QonHand = QonHand - new.Qty
   where ItemID = new.ItemID;
   if (new.Qty > (select Qavail from V_CORDER));
      update ITEM
      set   Qord   = Qord + (select mQeco from V_CORDER)
      where ItemID = new.ItemID;
      update ITEM
      set   (SuppID,PriceSupp) = (select SuppID,Price from OFFER
                                  where  ItemID = new.ItemID
                                  order by Price,Delay limit 1)
      where ItemID = new.ItemID and SuppID is null;
      insert into SUPPORDER(OrdID,DateOrd,ItemID,SuppID,
                            Price,Qty,State)
          select (select coalesce(max(OrdID),0)+1 from SUPPORDER),
                  new.DateOrd,new.ItemID,SuppID,PriceSupp,
                 (select mQeco from V_CORDER),'assigned'
          from   ITEM
          where  ItemID = new.ItemID;
   endif;
   update CUSTORDER
   set    Price = (select Price from V_CORDER),
          State = 'pending'
   where  OrdID = new.OrdID;
   insert into CUSTINVOICE(InvID,DateInv,OrdID,CustID,ItemID,
                           Price,Qty,State)
       values ((select InvID from V_CORDER),new.DateOrd,
                new.OrdID,new.CustID,new.ItemID,
               (select Price from V_CORDER),new.Qty,'pending');
   delete from V_CORDER;
end;
```

Script 8.61 - Trigger that controls the creation of a customer order when the quantity on hand is **not** sufficient

4. Then, the CUSTORDER row just inserted is completed (`update`) with the unit price of the item. In addition, its state is set to 'pending'.
5. An invoice is created by inserting a row in table CUSTINVOICE. Its state is also that of the customer order: 'pending'.
6. Finally, the local variables are reset.

8.18.2Executing a supplier order

Script 8.57 shows that the execution of all the orders of a definite item sent by the retail shop to a supplier is asked for by sending closing messages to the corresponding SUPPORDER rows, i.e., by changing their state from 'assigned' to 'closed'. The reaction consists in adjusting the quantities of the referenced ITEM row: the value of Qord of the order is added to the quantity of hand (QonHand) of the item and subtracted from its quantity on order (Qord) (Script 8.62).

```
create trigger TRG_SUPPORDER_UPD
after update of State on SUPPORDER
for each row
when new.State = 'closed'
begin
   update ITEM
   set   QonHand = QonHand + Qord,
         Qord    = Qord - new.Qord
   where ItemID = new.ItemID;
end;
```

Script 8.62 - Execution of a supplier order that (partially) replenish the stock of an item

When the last supplier order has been executed, the value of Qord of the ITEM row falls to 0. Setting the quantity on order to zero is a message to the ITEM row telling it that its pending invoices can now be processed. This is implemented in the trigger of Script 8.63. It comprises two queries:

1. The first one changes the state of pending invoices to 'sent'.
2. The second one re-evaluates the best offer for this item.

So, by changing the state of pending invoices, this trigger sends them a message to tell them that they can now be sent to their customers. The reaction is coded in trigger CUSTINVOICE_UPD1 of Script 8.64.

This trigger comprises three operations:

1. Calling procedure writeFile that prints (in a text file) the invoice to be sent to the customer. This procedure uses two arguments, the name of the text file and the character string to write in it. In the latter argument, symbol `'@n'` denotes a *new line* and function thousandSep formats a numeric character string by inserting thousands separators ('1234567' becomes '1,234,567').

```
create trigger TRG_ITEM_UPD
after update of Qord on ITEM
for each row
when new.Qord = 0
begin
   update CUSTINVOICE
   set   State = 'sent'
   where ItemID = new.ItemID
   and   State = 'pending';
   update ITEM
   set   (SuppID,PriceSupp) = (select SuppID,Price from OFFER
                               where  ItemID = new.ItemID
                               order by Price,Delay limit 1)
   where ItemID = new.ItemID;
end;
```

Script 8.63 - When the stock of an item is fully replenished, its pending customer orders and invoices can be processed

```
create trigger TRG_CUSTINVOICE_UPD1
after insert, update of State on CUSTINVOICE
for each row
when new.State = 'sent'
begin
   writeFile('D:/SQLfast/Files/INVOICE-'||new.InvID||'.txt',
             'INVOICE no '||new.InvID
           ||'@n--------------------'
           ||'@n CUSTOMER: '||new.CustID
           ||'@n     Date: '||new.DateInv
           ||'@n     Item: '||new.ItemID
           ||'@n      Qty: '||new.Qty
           ||'@n    Price: '||new.Price
           ||'@n--------------------'
           ||'@n    Total: '
           ||thousandSep(cast(new.Qty*new.Price as char),',');
   update CUSTORDER set State = 'invoiced'
   where OrdID = new.OrdID;
   update CUSTOMER set Account = Account - new.Amount
   where CustID = new.CustID;
end;
```

Script 8.64 - Sending the invoice to the customer of an order

2. Setting the state of the (pending) order of the invoice to 'invoiced'.
3. Subtracting the amount of the invoice from the account balance of the customer.

8.18.3Registering the payment of an invoice

According to Script 8.56, the registration of the payment of an invoice is triggered by the sending of a message to this invoice. This message is created by changing its status to 'paid'. When fired, trigger CUSTINVOICE_UPD2 closes the order of this invoice then updates the account balance of the customer (Script 8.65).

```
create trigger TRG_CUSTINVOICE_UPD2
after update of State on CUSTINVOICE
for each row
when new.State = 'paid'
begin
   update CUSTORDER
   set   State = 'closed'
   where OrdID = (select OrdID from CUSTINVOICE
                  where InvID = new.InvID);
   update CUSTOMER
   set   Account = Account + new.Amount
   where CustID = new.CustID;
end;
```

Script 8.65 - This invoice has been paid

8.18.4Event architecture of the active database

The graph of Figure 8.21 extends that of Figure 8.20. It shows the flow of messages that control the cascade of operations triggered by the three user interactions: *placing a customer order*, *paying an invoice* and *replenishing an item stock*. We observe that some tables appear more than once in this graph. Indeed, the messages sent by a table depend on the messages it receives. So, merging table nodes would reduce the information content of the graph.

8.19 Verification of an active database

The set of triggers in a database can make up a complex system in that the activation of one trigger can fire other triggers. So, it is not unusual that the first executions of an active database either crash or produce unexpected results.

Hence the need for specific techniques to analyze a trigger system to identify as early as possible flaws in its architecture. There are two main families of analysis techniques: static and dynamic. Static analysis techniques rely on the visual or automated examination of the source code of the triggers. Dynamic techniques extract and analyze information on the behavior of the triggers at run time.

We will examine informally some of these techniques.

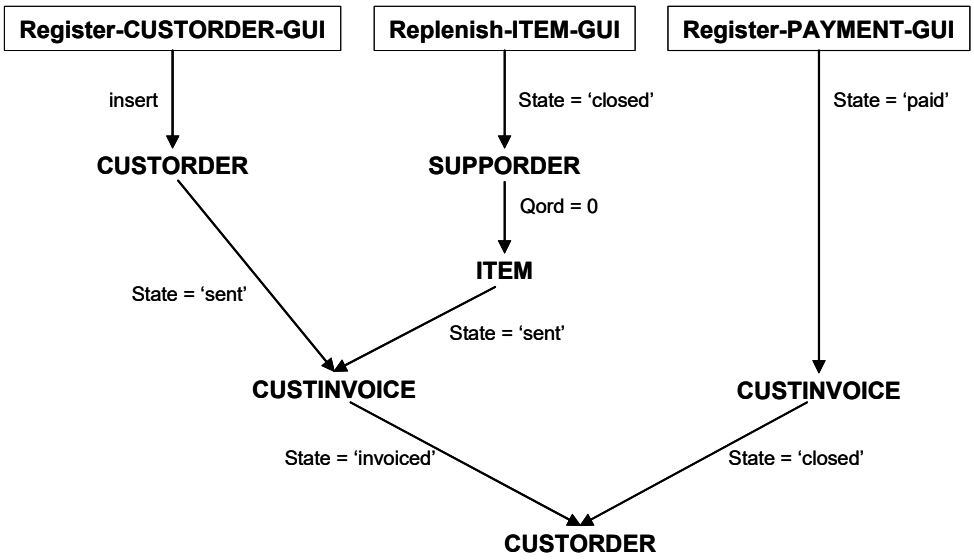


Figure 8.21 - The graph of the messages controlling the business application

8.19.1 Static analysis: checking trigger circuits

One of the major concerns of active database development is whether the execution of the trigger system being built will *terminate* whatever the initial event that triggers it. *Infinite execution* may occur when the code includes some form of *recursivity*, either in the explicit form of recursive triggers or when a trigger T executes data modifications that start a chain of modifications that ultimately fires T itself.

Let us consider directed graph **F** defined as follows. Each node represents a trigger and each arc (**tp**,**tq**) indicates tells that the body of **tp** includes modification queries that *may* fire trigger **tq**. If graph **F** is *acyclic*, that is, it *includes no circuit*, then the all the possible executions of the trigger system are ensured to terminate in a finite number of steps (lesser or equal to the number of triggers). If, on the contrary, **F** includes one or more circuits, then there is a risk of infinite execution.

Hence the importance of checking the acyclicity of **F**. If the test fails, we must identify the circuits of **F** for in-depth examination. These processes are carried out by two algorithms that are developed and illustrated in Appendix C.

The trigger system of the RETAIL.db database is depicted in Figure 8.22. A simple visual inspection shows that it is acyclic, so that there is no risk of infinite execution, whatever the initial data.

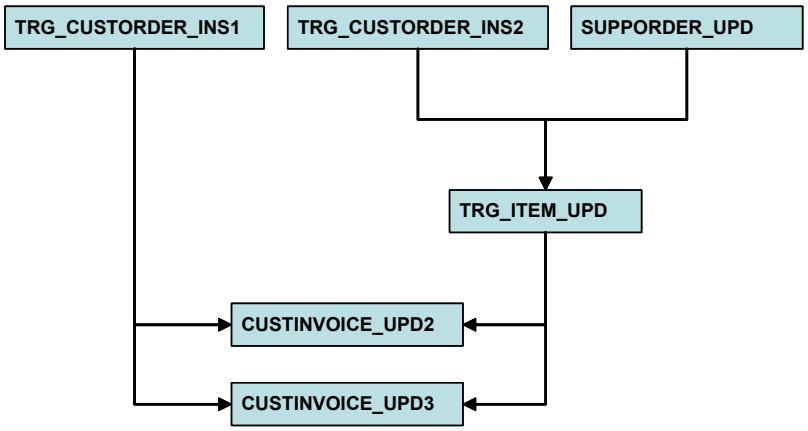


Figure 8.22 - The trigger graph of the RETAIL database

8.19.2 Dynamic analysis: tracing the execution of the active database

Trigger tracing techniques consists in extracting, storing and analyzing information on the execution of a trigger system. This information constitutes the trace of this execution. By examining the traces produced for a representative set of initial data, we can derive some important properties of these triggers. In particular, such traces can show us when and where the triggers exhibit unexpected behavior.

An in-depth description of tracing techniques in the SQLfast environment has been presented in Chapter 23 (*Aid to SQLfast script development*) of the SQLfast manual. In this chapter, Section 23.7 is devoted to a short introduction to *tracing trigger execution*. The reader is referred to this chapter before proceeding with this reading.

We will describe two complementary categories of traces: *event-based* and *operation-based*. Both rely on UDF functions provided by the SQLfast language that can be invoked from within trigger body. They are specific to SQLite but their generalization to other DBMS is straightforward.[26]

About the concept of *process level* in trigger tracing

Before describing these types of trace, we must refine the concept of *process level* introduced in Chapter 23 of the SQLfast manual and used in the tracing of SQLfast code. The process level specifies the depth of a process[27] in the call hierarchy at run time. If process **P**, with level **k** calls script **Q**, thus creating a process of **Q**, then the level of the latter process is **k**+1. The main script is at level 0.

26. All the more since big DBMS already include tracing features that are missing in SQLite.

The process level associated with each trace of a statement is stored in column Tlevel of table TRACES. To trace trigger executions, we also need to define some form of process level. We consider each trigger activation as a process. When an event created by a statement of a process (of a script or of a trigger) with level **k** is caught by a trigger, the activation of the latter is said to have level **k**+1.

In script execution, the process level if provided by system variable processLevel. However, this variable is not available from within a trigger execution, so that we will simulate it as a global variable that must be controlled *manually*. Since it must be available by all the triggers, it is translated into elementary table TLEVEL comprising a single column, Level, and a single row (see Section 8.2.7). Its definition and management are specified as shown in Script 8.66.

```
create table TLEVEL as select 0 as Level;
...
create trigger TRG_CUSTORDER_INS1
...
begin
   update TLEVEL set Level = Level + 1;
      ...
   update TLEVEL set Level = Level - 1;
end;
```

Script 8.66 - Implementation of the process level in triggers

Event-based tracing

Event-based tracing reports on the events that fire triggers. For each event, it generates a short description of the SQL data modification query and the start and end of the execution of the body of the triggers that fired. This is done by the instrumentation of the code of the body with procedure writeMessage(<message>) that writes character string <message> in the output window.[28] A summary of the event is crafted from states old and new of the current row. The expected result, produced by the execution of trigger TRG_CUSTORDER_INS1, is shown in Figure 8.23.

It must be noticed that, in addition to the six functional triggers TRG_CUSTORDER_INS1, TRG_CUSTORDER_INS2, TRG_ITEM_UPD (renamed TRG_ITEM_UPD2), TRG_CUSTINVOICE_UPD1, TRG_CUSTINVOICE_UPD2 and TRG_SUPPORDER_UPD (see Figure 8.22), we have created six auxiliary triggers (TRG_CUSTOMER_UPD, TRG_CUSTORDER_UPD, etc.) the sole objective of which is

27. A process is not the same thing as a script. It is the execution of a script. In a recursive execution, in which, for example, script A calls script B, which itself calls script A, this execution includes two processes of A and one process of B.
28. Remember that this procedure is implemented as a UDF function. Due to SQLite's limitations regarding trigger body syntax, the call to this function is made in the form $message$.

to write *we were here* in the trace. These additional triggers catch the events created by the functional triggers but that do not fire any other functional trigger.

For each trigger activation, the trace comprises four sections:

- the text of the event (`insert into CUSTORDER ...`),
- a label marking the start of the body, together with the identification of the trigger (> `Enter TRG_CUSTOMER_INS1`),
- for each query of the body that fires a trigger, and for each row affected by this query, the trace of this trigger; this trace is indented *wrt* its parent trace to make the parent-child hierarchy explicit,
- a label marking the end of the body (< `Exit TRG_CUSTOMER_INS1`).

The trace of Figure 8.23 shows that the recording of a customer order, initiated by event "`insert into CUSTORDER ...`)", fires trigger `TRG_CUSTORDER_INS1`. The execution of its body creates four events that fire successively triggers `TRG_ITEM_UPD1`, `TRG_CUSTORDER_UPD`, `TRG_CUSTINVOICE_INS` and `TRG_CUST INVOICE_UPD2`. The execution of the latter then fires triggers `TRG_CUST INVOICE_UPD2` and `TRG_CUSTOMER_UPD`.

```
EVENT: insert into CUSTORDER (1,'2020-08-30','C400',
                              'PA45',--,60,'recorded')
  > Enter TRG_CUSTORDER_INS1
    EVENT: update ITEM set QonHand = 20
      > Enter TRG_ITEM_UPD1
      < Exit TRG_ITEM_UPD1
    EVENT: update CUSTORDER set Price = 48
      > Enter TRG_CUSTORDER_UPD
      < Exit TRG_CUSTORDER_UPD
    EVENT: insert into CUSTINVOICE (1,'2020-08-30',1,'C400',
                                    'PA45',48,60,2880,'recorded')
      > Enter TRG_CUSTINVOICE_INS
      < Exit TRG_CUSTINVOICE_INS
    EVENT: update CUSTINVOICE set State = 'sent'
      > Enter TRG_CUSTINVOICE_UPD2
        EVENT: update CUSTORDER set State = 'invoiced'
          > Enter TRG_CUSTORDER_UPD
          < Exit TRG_CUSTORDER_UPD
        EVENT: update CUSTOMER set Account = -2880
          > Enter TRG_CUSTOMER_UPD
          < Exit TRG_CUSTOMER_UPD
      < Exit TRG_CUSTINVOICE_UPD2
  < Exit TRG_CUSTORDER_INS1
```

Figure 8.23 - Event trace of the creation of a customer order when the quantity ordered is available

The trace has been generated by the specific instrumentation of trigger TRG_CUSTORDER_INS1 (in blue in Script 8.60). The indentation of child traces within the trace of their parent is produced by the expression "`(select repeat(' ',Level*4) from TLEVEL)`"[29] that creates a space string from the process level stored in TLEVEL.

```
create trigger TRG_CUSTORDER_INS1
...
begin
   update TLEVEL set Level = Level + 1;
   -- Tracing ------------------------------------------
   writeMessage((select repeat(' ',Level*4) from TLEVEL)
                ||'EVENT: insert into CUSTORDER ('
                ||new.OrdID||','
                ||''''||new.DateOrd||''''||','
                ||''''||new.CustID||''''||','
                ||''''||new.ItemID||''''||','
                ||'--'||','
                ||new.Qty||','
                ||''''||new.State||''''||')');
   writeMessage((select repeat(' ',Level*4) from TLEVEL)
                ||'  > Enter TRG_CUSTORDER_INS1');
   -- ------------------------------------------------
   ...
   -- Tracing ------------------------------------------
   writeMessage((select repeat(' ',Level*4) from TLEVEL)
                ||'  < Exit TRG_CUSTORDER_INS1');
   -- ------------------------------------------------
   update TLEVEL set Level = Level - 1;
end;
```

Script 8.67 - Instrumentation of trigger TRG_CUSTORDER_INS1 for event tracing

Operation-based tracing

The operation-based tracing of triggers is an extension of the tracing of scripts described in Chapter 23 of the SQLfast manual. It consists in inserting in table TRACES of database TRACING.db the information on the execution of each query of the trigger bodies.

In the same way as we did to implement event-based tracing, we will instrument the body of the triggers we want to control through procedure `writeTrace(idx,tim,pro,lev,nat,raw,prep)`.[30] If we insert this procedure just before each SQL query, we will store a description of this query similar to those of pure SQLfast statements. In Script 8.68, procedure **writeTrace** will store in table TRACES the state of query "`update ITEM ... where ItemID = new.ItemID`" of trigger `TRG_CUST_ORDER_INS1` before its execution. We observe that:

– the value of column Tindex is set *manually*,

29. Function `repeat(s,n)` produces a character string formed by **n** occurrences of string **s**.
30. That will appear as `$trace$(...)` in SQLfast scripts.

– the name of the procedure (column Tproc) is built as the concatenation of the database name and the trigger name; this combination is unique,

– the process level (Tlevel) is drawn from column Level of table TLEVEL,

– the nature (Tnature) of the query is 0 (native elementary),

– the prepared version of the query (PrepStat) is built from the column values of the new state of the current row.

When processed by procedure writeTrace, the value of Tlevel will be augmented by the value of system variable procLevel. In this way, the activation of a trigger will appear in the trace as the calling of a procedure. In a procedure at level **k**, a trigger fired by a data modification query will appear at level **k**+1. So, the trace of the triggers will be seamlessly integrated in that of the SQLfast statements.

```
create trigger TRG_CUSTORDER_INS1
after insert on CUSTORDER
for each row
when new.Qty <= (select QonHand from ITEM
                 where ItemID = new.ItemID)
and  not exists (select * from CUSTINVOICE
                 where OrdID = new.OrdID)
begin
   update TLEVEL set Level = Level + 1;
   insert into V_CORDER(Price,InvID) ...;
   writeTrace(5,current_timestamp_full(),
            'RETAIL.db/TRG_CUSTORDER_INS1',
            (select Level from TLEVEL),0,
            'update ITEM set QonHand = QonHand - new.Qty
             where ItemID = new.ItemID',
            'update ITEM set QonHand = QonHand - ' || new.Qty||'
             where ItemID = ''' ||new.ItemID|| '''',1);
   update ITEM
   set    QonHand = QonHand - new.Qty
   where  ItemID = new.ItemID;
   update CUSTORDER ...;
   insert into CUSTINVOICE ...;
   delete from V_CORDER;
   update TLEVEL set Level = Level - 1;
end;
```

Script 8.68 - Writing the trace of query "`update ITEM`" in trigger `TRG_CUSTORDER_INS1`

Instrumenting triggers for event-based or operation-based tracing may be felt particularly convoluted. This is true. However, the good news is that the structure of the tracing statements appears to be quite systematic, in such a way that they can be automatically generated in the trigger code from the information provided by dictio-

nary tables SYS_TRIGGER_FULL and SYS_TRIGGER_COMP_FULL. The code of Script 8.69 is a skeleton of such a generator. Completing it is fairly easy and is left as an exercise.

```
set db = RETAIL.db;
openDB Dictionary-of-$db$;
for trid,taid,tanam,trnam,trwh,trev,trcol,trcon
   = [select TrigID,TableID,TableName,TrigName,TrigWhen,
             TrigEvent,TrigCol,TrigCond
      from   SYS_TRIGGER_FULL order by TrigID];
   if ('$trcol$' = '')
      write-b trigger $trnam$ $trwh$ $trev$ on $tanam$;
   if ('$trcol$' <> '')
      write-b trigger $trnam$ $trwh$ $trev$ of $trcol$
              on $tanam$;
   write begin;
   for seq,exp = [select StatSeq,StatExpr
                  from SYS_TRIGGER_COMP_FULL
                  where  TrigID = $trid$ order by StatSeq];
      if (startswith(lower('§exp§'),'select raise')) next;
      write @S2 writeTrace($seq$,...,$db$/$trnam$,1,0,...);;
      write @S2 $exp$;;
   endfor;
   write end;;
endfor;
```

Script 8.69 - Automating the instrumentation of the triggers of the RETAIL.db database

8.19.3 Replaying transactions

Let us consider a sequence of transactions such as those of the scenario of Section 8.13.4: registering two customer orders, then replenishing the stock of an item and finally registering the payment of a customer. Let us also consider that we want to execute them again. There can be several reasons for this:

- a bug has been found and fixed, then we want the four transactions being carried out again,
- we want to show newly hired employees how the application works,
- if the RETAIL application we have developed actually is just a playable prototype for a future, full scale, application, this scenario (and its effect on the database) is a part of the specifications[31] of the latter,

31. The specifications of an application is a document that defines precisely the goals of the application, without anticipating the way it will be implemented [short and incomplete definition!]

– we find this scenario to be an excellent test case, to be used to verify that the future versions of the application work as expected.

What we want is called *replaying* these transactions. In this context, the architecture of the RETAIL application in the form of an active database presents an interesting property. Since starting each interaction between a user and the application translates into a single data modification query, collecting then running these queries allow us to *replay* a series of transactions.

For example, our scenario results from the execution of the GUI scripts detailed in Section 8.17. These scripts have translated the user requests into the four queries recalled in Script 8.70. So, if we execute the latter script, we replay this scenario exactly, provided the database is reset in the same initial state of course.

```
insert into CUSTORDER(OrdID,DateOrd,CustID,ItemID,Qty)
       values (1,'2020-03-27','C400','PA45',60);
insert into CUSTORDER(OrdID,DateOrd,CustID,ItemID,Qty)
       values (2,'2020-03-27','K111','PA45',35);
update SUPPORDER set State = 'closed'
where  SuppID = 'F-725' and ItemID = 'PA45'
and    State = 'assigned';
update CUSTINVOICE set State = 'paid' where InvID = 1;
```

Script 8.70 - Replaying the scenario of Section 8.13.4

The nice aspect of this exercise is that such replay scripts can be automatically generated. By enabling the tracing of the SQLfast scripts, limited to SQL queries (for this we check button Trace SQL statements only), these queries are stored into the TRACES table. We just extract them by the simple query of Script 8.71 to build the replay script.

```
select PrepStat from TRACES where Tproc like '%-GUI';
```

Script 8.71 - Extracting the SQL queries for replaying a scenario

8.20 Limits of this case study

It is worth mentioning some of the simplifying assumptions which our active database is based on.

– The suppliers are idealized and abstract entities: their stock of items are infinite, they are absolutely reliable and they never require to be paid. In addition,

their item catalogue is the same as that of the retail shop (notably same Id and same description).

– Then, the customers are quite liable and trusted people: they always come and take their items away, they never cancel their orders and they always pay their invoices in due time.
– The shop and supplier employees never make reading, encoding, typing or operation errors.
– The IT infrastructure and the application programs never crash.
– The best offer selection policy guarantees that only one supplier at a time is asked to replenish the stock of an item.

Coping with these limitations to build a really usable application would require hundreds of code pages. This is left as an exercise!

8.21 Conclusions, history and extensions

In this case study, we have tried to illustrate two outstanding properties of active databases in general and triggers in particular: their power and the wide range of their applications.

It is interesting to note that triggers have been available for several decades in the early versions of most DBMS. One of their first use was to implement referential integrity maintenance well before foreign keys were available. This use is now deprecated unless one wants to implement non standard foreign key behavior.

From deductive databases to triggers

The concept of ECA (Event-Condition-Action) rule, or active rule, comes from the domain of *artificial intelligence* as it was perceived in the eighties.[32] The objective was to describe in a declarative way the dynamic evolution of knowledge. Another source of the concept is *logic programming*, where knowledge is represented by a collection of raw facts plus a set of derivation rules, possibly recursive. These concepts have been introduced in database models and technology to form the family of *deductive databases*. Some prototype implementations have been developed in the eighties but due to complex logic problems and low performance, none have really met industrial success. Most have disappeared in the nineties.[33]

32. In these years, statistical analysis of large data sets still was considered a domain distinct from that of A.I.

33. Datalog is another attempt to couple databases and logic programming. It is an *extended subset* of the Prolog language (https://en.wikipedia.org/wiki/Datalog). Many Datalog implementations still are available, either as standalone interpreters or as language extensions (e.g. PyDatalog, a Python module).

In some sense, active databases (mainly based on SQL views and triggers) are the remains of the historical deductive database dream. Instead of relying on a built-in generic deductive engine (similar to that of Prolog interpreters) the database designer now must build, from scratch, for each database, an adhoc deductive engine through which new data and operations are derived from base data and queries. Of course, implementing this engine with triggers often proves a non trivial task for large applications.

Recursive programming

Managing and exploiting *recursive data structures* can be done with recursive queries but also through recursive triggers. This is a point that we have not addressed in this study though it would deserve a full chapter in its own. Several application examples are developed in the SQLfast manual, Chapter 19 (*Recursive programming*).

Modeling and design

Developing large business applications has long been supported by a hierarchy of models and design processes.[34] To develop a database, the standard approach starts by building an abstract, technology independent conceptual schema, which is then translated into a specific technology, such as a relational or NoSQL DBMS. Solid models and design processes also exist to help develop the programs and the GUI of business applications.

Despite many attempts in the nineties,[35] there are few such comprehensive guidelines to create active databases.

We have also shown that a trigger system (the set of triggers attached to a database) is more difficult to model and to design than standard 3-tiers systems. Obviously, there is no single approach to design them.

- *Integrity maintenance* and *redundancy control* can be modeled by a set of equations for which we identify the events that may disturb them. For each of them, we specify the reaction that restores the equation.
- Building a *temporal database* will require quite different reasoning based on the consistency of a temporal sequence of database states (see the case studies on *Temporal Databases*).
- For *business applications* implementation where business rules are moved from the programs to the database, we could think of an approach inspired by

34. Look for example at the UML ecosystem that offers more than a dozen specific models to describe the different aspects of complex, interactive, data intensive, applications.

35. In particularly the IDEA approach to information system design, the modeling language of which, named Chimera included ECA rules. See Ceri S. and Fraternali, P. *The Story of the IDEA Methodology*, 1996, Springer [https://link.springer.com/content/pdf/10.1007/3-540-63107-0_1.pdf].

object-oriented modeling methods. We identify in the database representative objects classes (implemented as tables or aggregates of tables), with which we associate attributes and methods. These methods are activated by sending messages to objects through events.

Finally, what may active databases be good for?

Active databases generally are known for implementing non trivial integrity properties, that is, those that cannot be controlled by the standard uniqueness, referential and not null constraints nor by check predicates.

We have shown that the triggers of a databases can also be used to *silently* perform useful ancillary functions that simplify the task of application programmers. Redundancy management, updatable views, logging data changes and managing temporal data are some of the most useful applications.

All these examples share a common goal, showing that, far beyond being a mere data store, a database can be a very powerful knowledge management system.[36]

However, implementing full size business applications as active databases, that is, moving business rules from application programs to the database, is a quite different matter!

The experiment described in Section 8.15 suggests that such an enterprise is realistic, at least for small size applications. However, whether this architecture will prove better (and according to which criteria) than the standard one is, at least, questionable.

There are a number of arguments against trigger based architecture.

– Developing and understanding the code of a trigger system may prove more complex than that of a standard architecture. Indeed, there exist since long a rich variety of models and methods intended to help developers analyze, specify, verify and implement complex application programs. (Almost) none of them is able to usefully support the development of trigger-based technology. More in this in the next section.

– Active databases technology relies on strongly intricated programming paradigms, namely procedural, logic and event-based, in addition to the complexity of the RDBMS data structures. Many programmers, otherwise proficient in classical development languages (e.g., Java, Swing and ODBC) are not familiar with these concepts.

– Trigger body languages generally lack the resources that usually are available in standard programming languages. For instance, there is no easy way to execute a GUI dialogue, to catch an exception, to control the current transaction or use some API.

36. In addition, gathering and storing in a unique, secure, place (the database schema) a set of rules that are common to many applications contribute to the integrity of the data.

- There is no unique syntax nor execution model for triggers. Though the trigger concept is understood by most DBMS, its implementation may considerably vary, in particular concerning the way conflict situation are resolved. We have already discussed this issue in Section8.18.1. This means that a non trivial trigger system cannot be ported to another DBMS without an in-depth revision.

Reference

Paton, N., W. and Díaz, O. Active Database Systems, *ACM Computing Surveys*, Vol. 31, No. 1, March 1999, pp. 63-103 [*https://www.academia.edu/7023990/Active_database_systems*]

One of the most comprehensive study on active databases. Includes a very large bibliography.

8.22 The implementation

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Active_DB**.

They can be run from main script **ActiveDB-Main.sql**, that displays the selection box of Figure 8.24.

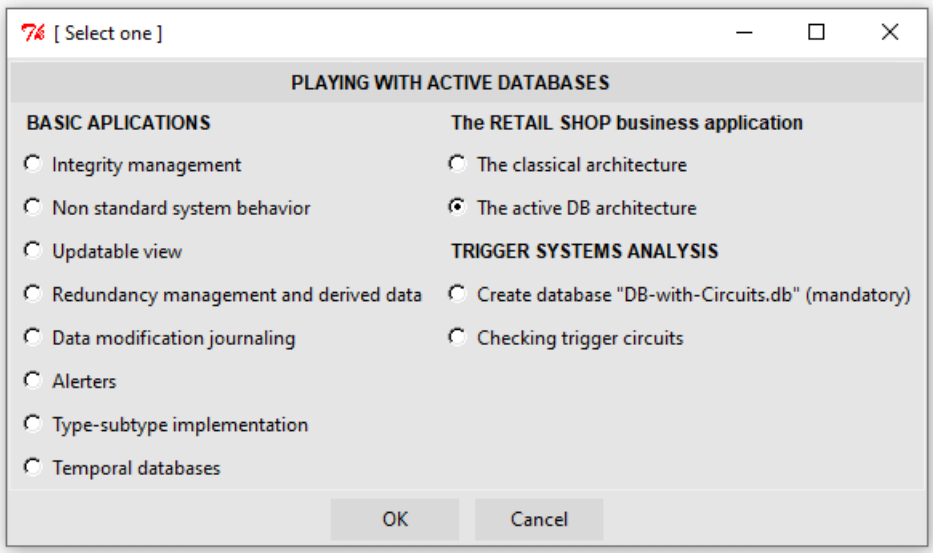


Figure 8.24 - Selection of an Active DB appplication

The last action, **Checking trigger circuits**, evaluates the triggers of two databases, namely RETAIL.db, created by the RETAIL Shop application (active DB version) and DB-with-Circuits.db, created by action **Create database "DB-with-Circuits.db"**.

Action **The active DB architecture** opens the selection box of Figure 8.25. It shows four categories of operations:

- **INITITIALIZATION**: creating the RETAIL.db database, ditto with event tracing, ditto with query tracing, load initial data in the base tables.
- **RETAIL OPERATIONS**. The operations to execute by the retail shop employee: creating a customer, creating an item, updating an item, recording customer order data and recording the payment of a customer.
- **SUPPLIER OPERATIONS**. The operations to execute by the employee(s) of the supplier(s): creating a supplier, adding an offer, updating an offer, and replenishing an item at the retail shop.
- **SHOW DATA**. Show the content of a table. Print a selected invoice.

These are the operations of interest to observe the behavior of the application. Other operations (e.g., updating and deleting elements) can be easily added.

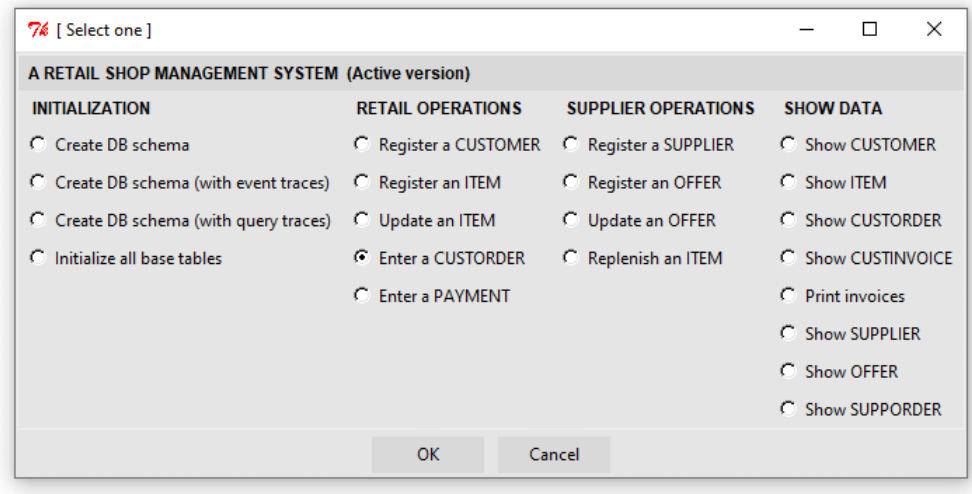


Figure 8.25 - The main menu of the retail shop application (active DB version)

Grouping the retail shop and the supplier(s) operations in a single selection box is quite artificial, as is a single work station were shared by the employees of all the actors.

We suggest a more natural organization, in which the retail shop and each supplier are assigned their own application. Its implementation is easy:

- Split the main script functions into two versions: ActiveDB-Shop-Main.sql and ActiveDB-Supplier-Main.sql.
- In the SQLfast directory, create as many copies of file SQLfast.exe as there are suppliers (named, for instance, SQLfast_D-109.exe, SQLfast_E-034.exe, etc.)
- The original file will be run by the shop employee, who then executes the first main script. Each supplier employee runs its own copy of SQLfast.exe and execute the supplier main script. These applications run in parallel and access the same database.

This is a good way to experiment with parallel execution of scripts!

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to let them develop their own applications.

8.23 Appendix A - The initial state of the database

The initial content of base tables is depicted in Figures 8.26 to 8.29. Observations:

- In table CUSTOMER, no unpaid invoice, so, Account = 0.
- In table ITEM, no quantity on order (Qord = 0, therefore QonHand = Qavail) and the inventory has just been replenished (QonHand = Qeco). No best offer selected yet.
- Table OFFER is fairly large (29 rows!). Since we will concentrate on item PA45, we have shown the offers of this item only.

CUSTOMER	CustID	Name	Address	City	Account
	B062	TAYLOR	139, Elm Park	Atlanta	0
	B112	WEBER	127, Hauptstraße	Berlin	0
	B332	MORETTI	1604, via Cavour	Roma	0
	B512	ROBERTS	108, Baker Street	London	0
	C003	WRIGHT	27, Portobello Road	London	0
	C123	SMITH	95B/2, Park Street	Atlanta	0
	C400	NEUMANN	454, Kirchenstraße	Berlin	0
	D063	SMITH	1083, King's Road	London	0
	F010	SCHMIDT	56, Bahnhofplatz	Berlin	0
	F011	JONES	10, Downing Street	London	0
	F400	LOPEZ	54B, Calle Carretas	Madrid	0
	K111	JANSEN	5B, Grote Halstraat	Amsterdam	0
	K729	EDWARDS	5, Abbey Road	London	0
	L422	GARCIA	40, W. Lake Avenue	Atlanta	0
	S127	JOHNSON	2164, Fifth Avenue	Atlanta	0
	S712	DUBOIS	58, avenue Montaigne	Paris	0

Figure 8.26 - Initial state of the CUSTOMER table

ITEM	ItemID	...	Price	SuppID	SuppPrice	QonHand	Qord	Qavail	Qeco
	CS262	...	70	--	--	40	0	40	40
	CS264	...	85	--	--	125	0	125	125
	CS464	...	125	--	--	100	0	100	100
	PA45	...	48	--	--	80	0	80	80
	PA60	...	55	--	--	60	0	60	60
	PH222	...	105	--	--	160	0	160	160
	PS222	...	94	--	--	110	0	110	110

Figure 8.27 - Initial state of ITEM table

SUPPLIER	SuppID	Name	City
	D-109	Wood&Steel	Berlin
	E-034	Golden Key	London
	F-725	BricoMat	Paris
	N-601	Materia-2000	Amsterdam
	U-542	Smith&Son	Atlanta

Figure 8.28 - Initial state of SUPPLIER table

OFFER	SuppID	ItemID	Price	Delay
	D-109	CS262	45	3
	D-109	CS264	68	2
	D-109	CS464	102	4
	D-109	PA45	35	3
	D-109	PA60	36	2
	D-109	PH222	165	5
	D-109	PS222	60	3
	E-034	CS262	40	3
	E-034	CS264	73	2
	E-034	CS464	90	1
	E-034	PA45	37	1
	E-034	PA60	38	2
	E-034	PH222	160	1
	E-034	PS222	72	3
	F-725	CS262	51	1
	F-725	CS264	62	4
	F-725	PA45	35	1
	F-725	PA60	38	3
	F-725	PH222	168	4
	N-601	CS262	42	1
	N-601	CS264	75	4
	N-601	CS464	85	5
	N-601	PA45	39	4
	N-601	PA60	30	3
	N-601	PH222	155	4
	U-542	CS262	40	6
	U-542	CS264	62	2
	U-542	CS464	88	3
	U-542	PS222	85	2

Figure 8.29 - Initial state of OFFER table

The transaction tables are empty.

8.24 Appendix B - RETAIL application: task analysis

Before coding our application, whatever architecture is chosen, we must identify the tasks to implement and precisely specify their effect on the database.

We will consider a limited set of representative tasks to be performed by the employees of the retail shop and of the suppliers: registering the data of new basic entities, registering a customer order, executing a supplier order and registering a customer payment. Other data management tasks, such as updating the contents of base tables, would be quite simple to analyze and implement.

8.24.1 Register basic entities

Registering a customer, an item, a supplier and an offer pose no particular problems. We just have to take care of the initial value of some critical attributes:

– The Amount column of the new **CUSTOMER** row is set to 0.

– For each new **ITEM** row, columns SuppID and SuppPrice, that define the best offer, are left undefined (i.e., *null*). Columns QonHand and Qavail are set to the same non negative value and Qord is set to 0. The values of Price (customer unit price) and of Qeco (the economic order quantity) are specified by the retail shop manager.

8.24.2 Register a customer order

The first operation consists in inserting a new row in table CUSTORDER. Columns OrdID, DateOrd, CustID and ItemID are extracted from the registering dialogue box. The value of Price is null and that of State is 'recorded'.

Let **co** be the CUSTORDER row just inserted, **it** the ITEM row referenced by co.ItemID and **cu** the CUSTOMER row referenced by co.CustID.

We consider two main cases, depending on the value of co.Qty compared to that of QonHand in the referenced ITEM row.

– **Case 1**. co.Qty ≤ it.QonHand: there is enough *quantity on hand*, **the order can be executed.**

- update ITEM row **it**:
  subtract co.Qty from it.QonHand
  (Qavail, being a generated column, is automatically updated)
- update CUSTORDER row **co**:
  set co.Price = it.Price
  set co.State = 'invoiced'
- insert row **ci** in table CUSTINVOICE:

set ci.(DateInv, OrdID, CustID, ItemID, Price, Qty, Amount, State) =
(co.DateOrd, co.OrdID, co.CustID, co.ItemID, co.Price, co.Qty * co.Price,
'sent')

- generate an invoice document from the data of row **ci**
- update CUSTOMER row **cu**:
  subtract ci.Amount from cu.Account

– **Case 2**. co.Qty > it.QonHand: there is not enough *quantity on hand*, **the order cannot be executed.**
Considering whether the *quantity on order* can suffice to satisfy the customer order leads us to two subcases.
  – **Case 2.1**. co.Qty ≤ it.Qavail : there is enough *quantity available*, **the order can be executed** as soon as the quantity on order (it.Qord) is delivered by the reference supplier.
    - update referenced ITEM row **it**:
      subtract co.Qty from it.QonHand
      subtract co.Qty from it.Qavail
    - update CUSTORDER row **co**:
      set co.Price = it.Price
      set co.State = 'pending'..
    - insert row **ci** in table CUSTINVOICE:
      set ci.(DateInv, OrdID, CustID, ItemID, Price, Qty, Amount, State) =
      (co.DateOrd, co.OrdID, co.CustID, co.ItemID, co.Price, co.Qty * co.Price,
      'pending')
  – **Case 2.2**. co.Qty > it.Qavail : there is not enough *quantity available*, **a new order must be sent to the reference supplier**.
    - compute mQeco, the total quantity to order:
      mQeco = smallest multiple of it.Qeco such that new value of it.Qavail is non negative
    - update referenced ITEM row **it**:
      subtract co.Qty from it.QonHand
      add mQeco to it.Qord
      subtract co.Qty from it.QonHand then add mQeco
    - if order **co** is the first one for item **it**, select best offer and update it.SuppID and it.PriceSupp
    - insert row **so** in table into SUPPORDER:
      set so.(OrdID, DateOrd, ItemID, SuppID, Price, Qty, State) =
      (<next value of SUPPORDER.OrdID>, co.DateOrd, it.ItemID, it.SuppID,

it.SuppPrice, mQeco, 'assigned')

- update CUSTORDER row **co**:
  set co.Price = it.Price
  set co.State = 'pending'.
- insert row **ci** in table CUSTINVOICE:
  set ci.(DateInv, OrdID, CustID, ItemID, Price, Qty, Amount, State) =
  (co.DateOrd, co.OrdID, co.CustID, co.ItemID, co.Price, co.Qty * co.Price,
  'pending')

8.24.3 Register a customer payment

Through the dialogue box, the employee of the retail shop specifies an invoice Id. Let **ci** be the corresponding CUSTINVOICE row, **co** the CUSTORDER row referenced by ci.OrdID and **cu** the CUSTOMER row referenced by ci.CustID.

- update CUSTINVOICE row **ci**:
  set ci.State = 'paid'
- update CUSTORDER row **co** referenced by ci:
  set co.State = 'closed'
- update CUSTOMER row **cu** referenced by ci:
  add ci.Amount to cu.Account

8.24.4 Execute a supplier order

An employee of a supplier decides to execute supplier orders from the retail shop. Through the dialogue box, the employee selects a couple of Id's (sup, itm) that identifies the set of SuppOrder rows where SupplierID = **sup**, ItemID = **itm** and State = 'assigned' (which means *still awaiting processing*). It is important to note that the reference supplier does not change between two replenishments. Consequently, all the (supplier) orders for an item are sent to the same supplier.[37]

let **it** be the ITEM row where ItemID = **itm**

let **totQord** the sum of Qord of the SUPPORDER rows
where SupplierID = **sup**, ItemID = **itm** and State = 'assigned'

- update ITEM row **it**:
  add totQord to it.QonHand'
  set it.Qord = 0

37. Allowing orders to be sent to different suppliers before the next replenishment would make the rules more complex.

- recalculate the best offer (result: OFFER row denoted by **bo**) and update ITEM row **it**:

  set it.SuppID = bo.SuppID

  set it.SuppPrice = bo.Price

- update SUPPORDER rows **so**
  where SupplierID = **sup**, ItemID = **itm** and State = 'assigned':

  set so.State = 'closed'

- update CUSTORDER rows **co** that reference **it** and where State = 'pending':

  set co.State = 'invoiced'

- for **ci** in CUSTINVOICE rows that reference **it** and where State = 'pending':
  let **cu** be the CUSTOMER row referenced by ci:
  - update CUSTOMER row **cu**:

    subtract ci.Amount from cu.Account

- generate an invoice document from the data of each CUSTINVOICE row **ci** that references **it** and where State = 'pending'

- update CUSTINVOICE rows **ci** that reference **it** and where State = 'pending'

  set co.State = 'sent'

8.25 Appendix C - Circuit detection in directed graphs

Avoiding infinite executions of a trigger system is not an easy problem. In this section, we will suggest some graph analysis algorithms that can help us identify directed graphs to detect and identify circuits.

Let us consider event **e0** that fires trigger **t0**, the body of which includes data modification queries. If the latter create events that are caught by other triggers, initial event **e0** starts a chain of data modification operations that can be of any length. This is what Figure 8.30 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.[38] Solving it is particularly simple and intuitive. However, finding these circuits may be a bit more difficult.

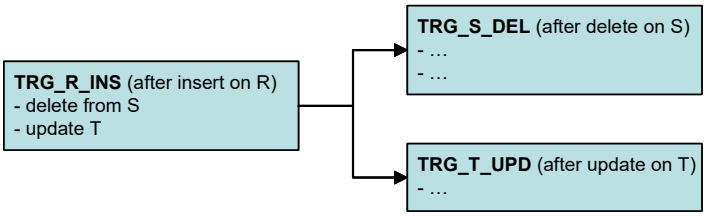


Figure 8.30 - When fired through an *insert into R* query, trigger TRG_R_INS executes data modification queries that fire triggers TRG_S_DEL and TRG_T_UPD

To study the acyclicity of trigger graphs in a practical way, we will consider the small schema of Script 8.72, that creates database DB-with-Circuits.db. This schema comprises the definitions of five tables and five triggers.

To derive the trigger graph from these definition, we will rely on the system tables of the dictionary. Among them, tables SYS_TRIGGER and SYS_TRIGGER_COMP include all the information needed to create this graph. These tables have been described in detail in Chapter 20 of the SQLfast Manual (*Metadata - Walking on the wild side*). As a reminder, table SYS_TRIGGER comprises the information of the header of triggers:

– TrigID: unique trigger id in the schema

38. It is addressed in two other case studies: *The book of which you are the hero* and *From data bulk loading to database book writing.*

- TableID: id of the parent table
- TrigName: name of the trigger
- TrigWhen: when the trigger will fire (`before`, `after`, `instead of`)
- TrigEvent: the event that fires the trigger (insert, delete, update)
- TrigCol: if the event is `'update'`, the list of columns (none for a general update)
- TrigScope: the scope or granularity of the trigger (`statement`, `row`)
- TrigCond: the condition of the `when` clause.

System table SYS_TRIGGER_COMP describes the statements of the body part of the trigger:

- TrigStatID: unique id of the statement in the schema
- TrigID: id of the parent trigger
- StatSeq: sequence number of the statement in the trigger body
- StatTable: the name of the table processed by the statement
- StatOper: the operation performed by the statement (`insert`, `delete`, `update`, others)
- StatCol: in case of `update` operation, the list of columns updated
- StatExpr: the integral text of the statement.

Chapter 20 also suggests that examining and processing the system tables may be more comfortable when their data are permanently stored into an independent database. This dictionary database can be created by script Create-persistent-dictionary.sql, available in the directory of this case study. Its name is formed as follows: 'Dictionary-of-' + <current database name>. Let us create it for our illustration database, with name Dictionary-of-DB-with-Circuits.db.

We observe that this dictionary database comprises additional tables the name of which is postfixed by '_FULL'. They are extensions of other system tables in order to ease their examination and their processing. In particular:

- table SYS_TRIGGER_FULL comprises new column TableName (name of the parent table)
- table SYS_TRIGGER_COMP_FULL comprises two new columns, TrigName (name of the trigger) and TableID (id of the trigger parent table).

Figure 8.31 shows the description of trigger R_INS (left) and of its second statement (right) in these extended system tables. The trigger has been assigned TrigID = 1. The id of the other triggers are shown in the right side of Script 8.72.

```
createOrReplaceDB DB-with-Circuits.db;
create table R (R1 int,R2 int,R3 int);
create table S (S1 int,S2 int,S3 int);
create table T (T1 int,T2 int,T3 int);
create table U (U1 int,U2 int,U3 int);
create table W (W1 int,W2 int,W3 int);
create trigger R_INS after insert on R                    [1]
    begin insert into S values(1,2,3);
          update T set T2=2,T3=3; end;
create trigger S_INS after insert on S                    [2]
    begin delete from U; end;
create trigger T_UPD after update on T                    [3]
    begin update T set T1=1;
          update W set W3=3; end;
create trigger U_DEL after delete on U                    [4]
    begin insert into S values(1,2,3);
          update W set W1=1; end;
create trigger W_UPD after update of W3 on W              [5]
    begin update W set W2=2; end;
closeDB;
```

Script 8.72 - A small database that ***may*** include trigger circuits

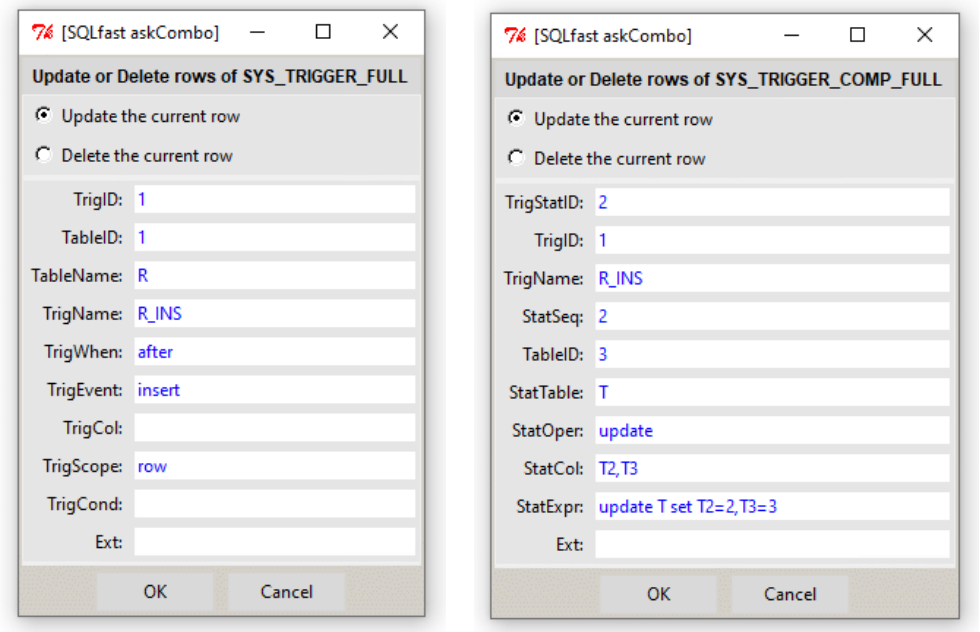


Figure 8.31 - Description of trigger R_INS and of its second body component "update T set T2=2,T3=3"

Before deriving and processing the trigger graph, we will apply a little modification in table SYS_TRIGGER_FULL in order to make the information more regular. When the event of the trigger is a general `update`, that specifies no columns, it implicitly specifies all the columns of the parent table. The code of Script 8.73 complete column TrigCol of all triggers fired by a general `update` event.

```
update SYS_TRIGGER_FULL
set    TrigCol = (select group_concat(ColName,',')
                  from   SYS_COLUMN
                  where  TableID = SYS_TRIGGER_FULL.TableID)
where  TrigEvent = 'update' and TrigCol = '';
```

Script 8.73 - SYS_TRIGGER_FULL is completed with "TrigCol" of general "update"

Now, let us build the trigger graph. It will be stored in table BINARY as binary relation (TrigID1, TrigID2) where TrigID1 denotes a trigger and TrigID2 a trigger that the former is likely to fire. The content of this table is based on a simple syntactic analysis of the code of the triggers. Practically, row (t1,t2) is inserted into table BINARY if one of these conditions is met:

1. (1) the body of trigger t1 comprises a statement `insert into X` and (2) trigger `t2` is fired by any event of the form `insert on X`,
2. (1) the body of trigger t1 comprises a statement `delete from X` and (2) trigger `t2` is fired by any event of the form `delete on X`,
3. (1) the body of trigger t1 comprises a statement `update X` affecting columns StatCol, (2) trigger t2 is fired by any event `update on X`, affecting (explicitly or implicitly) columns TrigCol and (3) column lists TrigCol and StatCol share at least one common column.

According to the last rule, the statement "`update T set T2=2,T3=3`" of trigger R_INS (TrigID = 1) will fire trigger T_UPD (TrigID = 3), the event of which is "`update on T`", equivalent to "`update T1,T2,T3 on T`". Indeed, sets StatCol = {`T2,T3`} and TrigCol = {`T1,T2,T3`} have elements {`T2,T3`} in common. Therefore, (1,3) is a row of table BINARY.

On the contrary, the execution of trigger U_DEL will never fire trigger W_UPD because sets StatCol = {`W1`} and TrigCol = {`W3`} have no common elements.

These rules are translated into the code of Script 8.74. The join associates the rows of table SYS_TRIGGER_COMP_FULL (from which the value of TrigID1 will be extracted) with those of table SYS_TRIGGER_FULL (from which the value of TrigID2 will be extracted) when they share the same table name (`C1.StatTable = T2.TableName`) and event (`C1.StatOper = T2.TrigEvent`). The third condition selects the couples of rows that satisfy the three rules described above. The state of BINARY is shown in Figure 8.32 and its graphical representation in Figure 8.33.

```
create temp table BINARY as
select TrigID1,TrigID2
from (select
             C1.TrigID as TrigID1,C1.StatOper,C1.StatCol,
             T2.TrigID as TrigID2,T2.TrigCol
      from   SYS_TRIGGER_COMP_FULL C1,
             SYS_TRIGGER_FULL T2
      where  C1.StatTable = T2.TableName
      and    C1.StatOper = T2.TrigEvent
      and    case
             when C1.StatOper = 'update'
            then iif(itemInter(C1.StatCol,T2.TrigCol,',') = '',
                      False,True)
             else True
             end);
```

Script 8.74 - Building the BINARY table

TrigID1	TrigID2
1	2
1	3
2	4
3	3
3	5
4	2

Figure 8.32 - Table BINARY derived from the schema of Script 8.72.

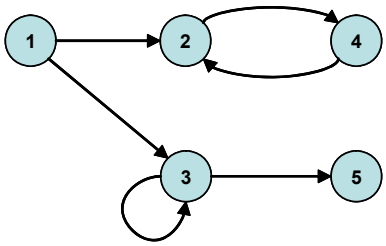


Figure 8.33 - Graphical representation of the rows of table BINARY

A first interesting question is whether the graph includes one or more circuit(s). To belong to a circuit, a node must be the source of some arc(s) **and** the destination of some (not necessary distinct) arc(s). A node that does not meet these conditions cannot belong to a circuit.

Considering the graphical interpretation of Figure 8.33, we observe that arcs (1,2) and (1,3) cannot participate in circuits since node 1 is the destination of no arc (it is not accessible from another node). Similarly, node 5 is the source node of no arc. Obviously, nodes 1 and 5 can be discarded from the graph without modifying its cyclicity property. Finally, we observe that nodes 2, 3 and 4, that are both source and destination of some arcs, cannot be eliminated based on the above rules.

This suggest a simple and intuitive algorithm:[39] we eliminate the rows of BINARY the TrigID2 value of which is not in the values of TrigID1 and the rows the TrigID2 value of which is not in the values of TrigID1. We process is to iterate until there is no elimination possible. The graph is acyclic if table BINARY is empty. This algorithms is translated in Script 8.75.

```
extract N1 = select count(*) from BINARY;
set N2 = 999;
while ($N1$ < $N2$);
   delete from BINARY
   where TrigID2 not in (select TrigID1 from BINARY);
   delete from BINARY
   where TrigID1 not in (select TrigID2 from BINARY);
   set N2 = $N1$;
   extract N1 = select count(*) from BINARY;
endwhile;
if ($N1$ = 0) write The graph is acyclic;
```

Script 8.75 - Checking whether a graph is acyclic

Now, we would like to extract the circuits *hidden* in table BINARY. A circuit is a path in the graph (a suite of consecutive arcs) the end of which is the starting node. The use of a recursive query clearly is a matter of course (Script 8.76). The CTE PATHS(TFrom,TTo,Path) computes all the paths in BINARY the last node of which is not already present (ignoring the first position).

We start by considering that each row of BINARY is an elementary path. Then, to each of these paths, we add a matching row from BINARY, if any. We go on until we try to add a node that has already been inserted into the path, in which case we stop augmenting it. In this comparison, we only consider the nodes *beyond* the starting one. When all the paths have been built in PATHS, the final query selects those for which TFrom = TTo (Figure 8.34)

39. Called the *Marimont* algorithm.

```
create temp view CIRCUIT(Circuit)
as with PATHS(TFrom,TTo,Path) as
     (select TrigID1,TrigID2,'.'||TrigID1||'.'||TrigID2||'.'
      from   BINARY
         union
      select TFrom,TrigID2,Path||TrigID2||'.'
      from   PATHS,BINARY
      where  TTo = TrigID1
      and    Path not like '.%.'||TrigID2||'.%'
     )
select Path from PATHS where TFrom = TTo;
```

Script 8.76 - Computing the circuits of the graph BINARY

```
+---------+
| Circuit |
+---------+
| .3.3.   |
| .2.4.2. |
| .4.2.4. |
+---------+
```

Figure 8.34 - The circuits as shown by view CIRCUITS

Actually, we are not completely done. First, it would nice to discard the heading and trailing dots. Then, we observe that each circuit is represented in view CIRCUITS by as many rows as the circuit comprises nodes. For instance, the circuit formed by nodes 2 and 4 is represented twice, by 2.4.2 and 4.2.4. Since any node of a circuit can be, by convention, selected as its starting points, we suggest to represent each circuit by a single expression, the first node of which has the smallest denotation, lexicographically speaking, that is, here, "2.4.2". This is what Script 8.77 computes. Function trim removes leading and trailing dots, while function itemSort generates a sorted permutation of the node denotations. This function requires four parameters: the list to sort, the sorting direction (0 = ascending, 1 = descending), the uniqueness (0 = duplicates preserved; 1 = duplicates discarded) and the separator. The result is shown in Figure 8.35

```
select trim(min(Circuit),'.') as Circuit
from   CIRCUIT
group by itemSort(trim(Circuit,'.'),0,'1','.');
```

Script 8.77 - Cleaning and reducing the expressions of circuits

```
+---------+
| Circuit |
+---------+
| 2.4.2   |
| 3.3     |
+---------+
```

Figure 8.35 - The final view of the circuits of triggers

Conclusion on trigger (a)cyclic graph analysis

It is necessary to observe that the analysis of the trigger system developed above can prove that its graph is acyclic. However, the fact that circuits have been detected in the schema does not mean that they will lead to infinite execution. The real danger is the existence of circuits *at the data level*. When the triggers in a circuit fire several times, each of their executions must tend to an exit point. Practically, this condition will be evaluated manually. Hence the importance to identify all the triggers participating in a circuit and to prove that each of their executions brings them closer to the termination.

A row such as (t1,t2) can be interpreted as: *trigger t1* ***will*** *fire trigger t2*, only if t2 includes no `when` clause and if data modification queries in the body of t1 include no `where` clause, i.e., if the firing relation does not depend on the state of the database. Otherwise the interpretation of (t1,t2) must be weaken as: *trigger t1* ***may*** *fire trigger t2*.

Let us modify our example schema by adding `when` and `where` clauses (Script 8.78). In trigger R_INS, the second body statement will execute under condition `new.R2 > 0`, the value of which can only be known at execution time. If this statement is executed, then trigger T_UPD will fire under the condition that column T3 is assigned increasing values, a condition that too can only be evaluated at execution time.

From the analysis of an active database, we can tell that the graph of the trigger system is acyclic and therefore that there is no risk of infinite execution. However, if this graph includes circuits, proving (or at least getting reasonable convinced) that there is any (or no) risk of infinite execution would require either a more in-depth syntactic analysis of the code or the tracing of the execution of the concerned triggers for a representative collection of initial data.

When it is impossible to prove that a trigger circuit will never cause infinite execution, it is recommended to include protection mechanisms in the code of the incriminated triggers. See for example the section "***Recursive queries against cyclic data***" in chapter 19 of the SQLfast manual (***Recursive programming***), which proposes several techniques to stop infinite executions.

```
...
create trigger R_INS after insert on R                    [1]
begin insert into S values(1,2,3);
      update T set T2=2,T3=3 where new.R2 > 0; end;
...
create trigger T_UPD after update on T                    [3]
when new.T3 > old.T3
begin update T set T1=1;
      update W set W3=3; end;
```

Script 8.78 - Trigger R_INS **may** fire trigger T_UPD

Case study **9**

Temporal databases - Part 1

Objective. In this study we examine various ways to organize the data describing the evolution of a population of entities. The basic model consists in storing the successive states of each entity, completed by the time period during which the state was observable. We distinguish between the *transaction time*, that refers to the data modification time in the database and the *valid time*, referring to modification events of entities in the real world. This study particularly develops *entity-based*, *attribute-based*, *event-based* and *document-oriented* temporal database models. In these models, data management is ensured by triggers that automate as far as possible entity creation, modification and deletion operations.

The next study will be devoted to temporal database querying and transformation.

Keywords. temporal data type, temporal database, history, entity type, time point, time period, time interval, evolution, event, state, transaction time, valid time, bitemporal data, julian day, document-oriented model, JSON, trigger

9.1 Introduction

A database, as it generally is considered in text books and lectures, informs on the current state of the application domain, that is, that part of the real world it is intended to describe. This is the case of the ORDERS.db database we have used as the support of most chapters of the SQLfast tutorial and of some case studies.

It turns out that things change, sometimes at a fast pace. For instance, new customers are registered, new products are offered, orders are placed, prices are modified, customers move to other cities, orders are cancelled and obsolete products are withdrawn.

These changes are propagated to the database, as soon as possible, in such a way that the data always reflect the current state of the application domain. To know the price of a product, we just consult the database.

The problem with this way of managing data is that we lose the trace of past facts. To compute the total amount of an order we need to know the price of products *at the time the order was placed*, not their current price, which may be higher. We may also need to know the previous address(es) of customers, when they have been registered (and what was their first order), which of them have been removed from our customer base, and when. In others words, we would like to know, not only the current state of the application domain, but also its past states, that is, its *history.*[1]

Keeping the trace of past states is not as obvious as we could think, particularly if we intend to *reason* on these states. Taking a save copy of the database at regular time point as we do for a simple document is not an option. Just adding a date column (e.g., *createdOn*, *lastUpdatedOn*) does not work either. To cope with these new problems in a clean and coherent way, we must add to databases a new dimension: *time*. This is what we call a **temporal database**.

This is the goal of this series of studies to develop some of the most appropriate techniques to represent historical data, to manage them and to exploit them. The domain of *temporal data* (of which historical data are a large subset) is particularly rich and would deserve hundreds of pages to describe and develop most of its aspects. Considering the modest objectives of these case studies, we will content ourselves with developing in a very practical way some of these aspects.

Just to give readers the taste of them and the desire to know more!

9.2 Representation of time

Let us first address the way time will be represented in our case studies. Time representation systems appear to be particularly complex when we consider all their physical, geographical, cultural and historical idiosyncrasies. To keep things simple,

1. We will consider past and current states, but not future states, that may pose specific problems.

and considering the kind of application domains we intend to address, we will adopt this subset of the ISO 8601 standard:[2]

1. *dates*: format YYYY-MM-DD.
   Example: 2019-07-24
2. *time* in day: format HH:MM:SS or HH:MM:SS.mmm, where mmm represents milliseconds[3].
   Examples: 16:32:09 or 16:32:09.085
3. *datetime*: date + time separated by one space or by letter T. Also often named *timestamp*.
   Example: 2019-07-24 16:32:09.085 or 2019-07-24T16:32:09.085

In these formats, we ignore time zones as well as the numerous variations and shorthands allowed by the standard.

A key concept of our model is the ***granularity*** (or *precision*) of the temporal dimension. It is defined by the smallest unit of time we can use to specify an instant. In the examples illustrating the time format here above, the granularities are respectively the *day*, the *second* and the *millisecond*. The appropriate granularity depends on the kind of phenomenon we intend to describe. In business processes, such as those concerned by database ORDERS.db, the *day*, or, at best, the *hour*, should be sufficient. We can imagine that anthropology, astronomy or subatomic physics would require quite different granularities. A **time point** is any instant to which a value (a **timestamp**[4]) is assigned in the current granularity. To simplify the development of temporal databases, we will associate with our application domain a unique granularity.

In some applications, instead of the real time values, we will use an **abstract time** representation through positive integers: 2, 7, 20, 40, etc., as shown in Figure 9.3. This will be useful either to simplify some demonstration (real values as 2017-07-24 16:32:09.085 are quite cumbersome!) or to resolve granularity conflicts, for instance to distinguish and order two events that occur at the same time point.

An **event** is a perceived change that occurs in the application domain or in the database and that is considered worth being documented. The main characteristic of an event, besides the nature of the change, is the time at which it occurred. Later, we will use the term **valid time** when the change occurs in the application domain and **transaction time** when the change applies to the contents of the database. When a change in the application domain is recorded in the database, we could want to include in this recording both the valid and transaction times to the change. The database then comprises **bitemporal** data.

2. See notably https://en.wikipedia.org/wiki/ISO_8601 and https://www.iso.org/iso-8601-date-and-time-format.html
3. Actually, according to the standard, the fractional part can be specified up to seven decimals. However, many DBMS impose a limit of three decimals.
4. This term may raise ambiguity. In the SQL language, timestamp is a temporal data type. Here, it is a value created to identify an instant in a timeline, according to the chosen granularity.

We must also mention the concept of **timeline**, which is the ordered set of all the time points that can be associated to events. In the kind of application domains we will cope with, it is common:

1. to identify the *starting point* of the timeline with a definite time, the events that occurred before this point being ignored,
2. to leave the *ending point* undefined, that is, set to *infinite future*. Since it is not easy to represent infinite in a computer, we will associate with the ending point a definite time, but very far in the future.[5]

In the basic case we will study (database TDB.db), we will choose date 2014-11-18 as the starting point and 9999-12-31 as the ending point.

Finally, three terms that must be (re)defined before starting:

- *period*: a portion of a timeline comprised between two time points. A period is defined by these time points. Can be *closed* (includes its starting and ending points), *open* (excludes its starting and ending points) or *half-open* (one of its extreme points is excluded).
- *interval*: a pair of time points delimiting a series of consecutive time points. An *interval* + a *time point* setting its starting point define a period. In other words, a period is an anchored interval. SQL:1999 defines an *interval* as a directed duration expressed either in *years + months* or in *days + time*.
- *duration*: exact measure of the length of a period. Expressed in non-ambiguous units such as days or smaller. Also called *time distance* or *span*. Example: 41,567.146 seconds

In practice, this vocabulary is used in a quite ambiguous and confusing way. Among the authors in the database domain, *interval* and *period* are often used interchangeably, sometimes in the same document. In both parts of this case study, we will use, whenever no ambiguity may arise, the terms *period* and *interval* to denote any portion of a timeline. When an interval is associated to a state, we will sometimes allow ourself to identify this state to its interval: for instance, *both states overlap* is a shorthand for *the intervals of these states overlap*.

9.3 The concept of current state

The project we will develop concerns a collection of *entities* that we intend to observe and describe during a definite time period. Entities appear, evolve then disappear. We will illustrate the study by a small application domain made up of two sorts of entities, namely *projects* and *employees*:

5. Some authors suggest to represent the ending point by a *null* value. This is consistent but this will make many queries more complex.

1. *projects* have three attributes: their *title*, their *theme* and their *budget*; their title is unique
2. *employees* have five attributes: their *code* (which is unique), their *name*, their *salary*, the *city* they live in and the *project* they work on.

Currently (we mean *now*), our project comprises two projects and five employees. They can be described by rows in relational tables PROJECT and EMPLOYEE of database **TDB.db** [6] (Figures 9.1 and 9.2). Each entity type is represented by a table.[7] In a table, each row describes one entity and each entity is described by one row. Only *active entities* are documented. If an entity does not exist (yet), it has no descriptive row in the table. Similarly, when an entity disappears, there is no row any more to tell that it has, one day, existed. These tables describe the **current state** of the application domain. The rows of these tables are denotations of existing entities and their current attributes.

TITLE	THEME	BUDGET
AGRO-2000	Crop improvement	82000
BIOTECH	Biotechnology	140000

Figure 9.1 - The current state of project entities

CODE	NAME	SALARY	CITY	PROJECT
A237	Antoine	3800	Grenoble	AGRO-2000
C45	Carlier	3100	Lille	BIOTECH
D107	Delecourt	3300	Grenoble	BIOTECH
G96	Godin	3900	Genève	AGRO-2000
M158	Mercier	3000	Paris	BIOTECH

Figure 9.2 - The current state of employee entities

9.4 History of an entity

An entity has a *life*, which is the time period during which it is *active* (i.e., it is *living*). It starts with its birth time and ends with its death time or the current time if it still lives. Before its *life*, the entity does not exist, and after its *life*, it does not exist

6. This database is available in directory databases of the SQLfast distribution. Actually, EMPLOYEE and PROJECT are not base tables but views on the historical data.
7. This does not preclude situations in which entities can be composed of several sub-entities. This is the case of database ORDERS.db, in which an order comprises a header and one or several details. Headers are described in table CUSTORDER and details in table DETAIL. Distributing the data of complex entities among several tables is typical of normalized relational databases, in which redundancies are, as far as possible, eliminated.

any more, though its memory can be maintained. The attributes of an entity are assigned specific values at birth time and will generally change their values over time.

If we intend to document the life of these entities, we will have to record all the events that affect their existence: birth, modification of their attributes and death. These events form the *history* of the entity. Figure 9.3 depicts the history of project BIOTECH. The event times are real dates, but they are denoted by abstract number for reasons that will be discussed later.

We observe that the project started at time 2, then, at time 7, got a new theme and a lower budget, at time 9, still a lower budget, and so on until now, at which point the project still is active.

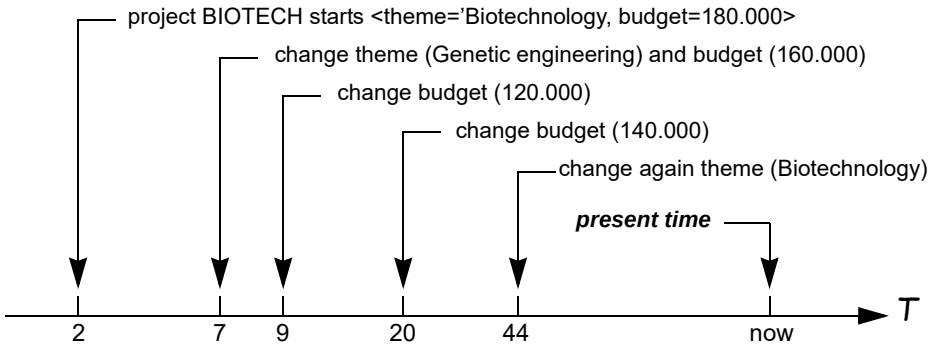


Figure 9.3 - History of project BIOTECH

now is a time point a bit special: it is constantly moving rightward! It is therefore difficult to assign it an actual temporal value. There are several ways to denote it. For instance through a special symbol such as an empty string or *null* (the SQL marker), or `'until changed'` or `'infinite future'` or ∞.

For convenience reason that will appear later, we will choose a time point very far in the future, so far that it will never be confused with an actual time point of the application domain.

Times 9999-12-31 or 9999-12-31 23:59:59 or 9999-12-31 23:59:59.999 (in ISO format) depending on the chosen granularity, could be fair choices, at least to model human activities. Abstract time point 999999 could be less far (less than one million events) but still appropriate, at least in our sample database.

Note. The abstract time points must be interpreted with some caution. For example, considering that the time is measured in days, time point 7 does not necessarily denote the date of time point 2 plus 5 days. It just denotes an outstanding date later than the date of time point 2, and that there *may* be four other outstanding date between them, denoted 3, 4, 5 and 6, be they used or

not in building the history of other entities. This will become clear with the concept of table CLOCK.

From events to states

Instead of recording these events, we can record the sequence of *states* which each entity goes through. A state is defined by the *time period* during which the values of its properties remain unchanged. When the value of an attribute changes, the current state ends (and from then belongs to the past of the entity) and a new current state starts.

Converted into states, the history of project entities is stored in history table H_PROJECT of database TDB.db. A part of it, shown in Figure 9.4, provides the complete history of project BIOTECH. We have represented each state by a row, in which we show the values of the attributes of the project during this state. The state itself is specified by the starting point (column **start**) and the ending point (column **end**) of its time period.[8] Similarly, the history of employee entities will be stored in history table H_EMPLOYEE.

We observe that the last time point of a period (**end**) is also the first point of the next period. This does not mean that this time point is common to both states. Instead, the **end** point of a state is excluded, so that the time period of a state is a semi-open period, noted **[start, end)**. As we will see later, this convention will simplify many algorithms. Moreover, it remains valid, whatever the granularity of the time measure.

TITLE	THEME	BUDGET	start	end
...	...	...	...	...
BIOTECH	Biotechnology	180000	2	7
BIOTECH	Genetic engineering	160000	7	9
BIOTECH	Genetic engineering	120000	9	20
BIOTECH	Genetic engineering	140000	20	44
BIOTECH	Biotechnology	140000	44	999999
...	...	...	...	...

Figure 9.4 - Excerpt of table H_PROJECT showing the history of project BIOTECH

For instance, if point 7 denotes date 2016-02-17 and point 9, date 2016-07-24, then the second state of Figure 9.4 is covered by period **[2016-02-17,2016-07-24)**, or, if the time granularity is *one day*, by **[2016-02-17,2016-07-23]**.

8. Names start and end are reserved words in the SQL standard and in some DBMS. If needed, they will be renamed as, for example, Start_time and End_time, or, shorter, Stime and Etime. SQLite is fairly tolerant on this point!

9.4.1 Temporal primary, unique and foreign keys

Keys are standard features of relational databases. *Primary* and *unique keys* support uniqueness constraints while *foreign keys* ensure referential constraints. The *primary key* of a table is a unique key that has been given a special status. In particular, all its components must be declared `not null`. In the mind of the designer, it represents the *essence* of entities[9].

What do they become when we add a temporal dimension to data?

Let us start with the non temporal tables of Figures 9.1 and 9.2, that describe the current states of entities. Let us call them *entity tables* PROJECT and EMPLOYEE. The primary key of PROJECT is column TITLE (that translates project id's) while that of EMPLOYEE is column CODE. Column PROJECT in EMPLOYEE is a foreign key referencing table PROJECT. Its target is the primary key of PROJECT.

Temporal primary keys

Considering that a state is at least one time unit long, the values of **start** of two successive states of an entity are distinct. This suggests that,

- the *temporal primary key* of H_PROJECT is (TITLE,start)
- the *temporal primary key* of H_EMPLOYEE is (CODE,start).

Temporal unique keys

Similarly, each unique key **UK** in entity table **T** gives *temporal unique key* **(UK,start)** in historical table **H_T**.

We could also declare (TITLE,end) and (CODE,end) *unique keys*, but the way historical data will be managed will automatically ensure their uniqueness.

Temporal foreign keys

There is no big risk in declaring (PROJECT,start) a *temporal foreign key* of H_EMPLOYEE referencing H_PROJECT. However, the concept of *referential integrity* is less straightforward than in non temporal tables.

In non temporal database, we must check that:

for each row **e** in EMPLOYEE, there exists a row **p** in PROJECT such that:

- **p**.TITLE = **e**.PROJET

Translated for a temporal database, this property becomes:

for reach row **he** in H_EMPLOYEE, for each time point **t** of **he**, there exists a row **hp** in H_PROJECT such that **hp**.TITLE = **he**.PROJECT and **hp**.start $\leq$ **t** < **hp**.end

9. The *non-nullability* of primary keys is often expressed as *entity integrity.*

More practically,

for each row **he** in H_EMPLOYEE, there exists two rows **hp1**, **hp2** (not necessarily distinct) in H_PROJECT such that (see Figure 9.5):

- **hp1**.TITLE = **hp2**.TITLE = **he**.PROJET
- **hp1**.start ≤ **he**.start et **he**.end ≤ **hp2**.end

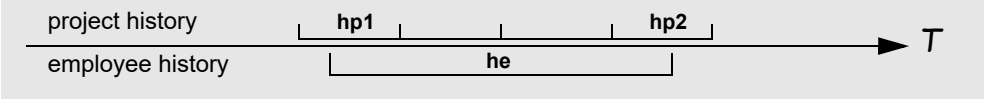


Figure 9.5 - Temporal referential integrity

9.4.2 Normalized history

So, to develop a consistent temporal database, we must enforce some constraints on the *non temporal* schema:

1. Each entity table has a primary key. Its components are **stable**: they cannot be modified. Its components are **not reusable**: the values of the primary key of an entity that has been deleted cannot be assigned to another entity. These properties ensure that all the states of a definite entity can be gathered without any ambiguity.
2. The target of each foreign key is the primary key of the referenced table.
3. The schema of the database does not evolve. Studying schema evolution of historical data is (far) beyond the scope of this studies.

In addition, the history of an entity must satisfy the following properties:

4. There is no gap in its history. This means that each state **s1**, except the last one, is followed by another state **s2** such that **s2.end** = **s2.start**. States **s2** and **s2** are said **consecutive**. A history that includes no gap is called **complete**.
5. Two consecutive states are distinct. They must have at least one different attribute value.
6. The duration of a state is at least one time unit. This means that **start** < **end**.
7. At any time, an entity is in one state only. Two different states may not overlap.

There is another constraint on the events that will create the history:

8. For each entity, the modification events occur *chronologically*. This means that, at any time, the value of **start** of the last state of an entity is higher than those of all its other states. Making it possible to change past states (for in-

stance to correct errors) would require a *bitemporal history*, which would be much more complicated to manage and process.

A history that satisfies these constraints is called **normalized**. Weaker forms of history may be considered and coped with. In a non normalized history,

– gaps may exist,

– consecutive states may be *value-equivalent* (their columns have exactly the same values),

– two states may overlap provided they are value-equivalent.

Such forms usually appear in the result set of some temporal queries. They can be normalized through the reduce and normalize operator. These issues will be studied in the second part of the case study.

The 7th constraint is particularly critical: a history in which some entities have more than one distinct state at some time point is considered **corrupt**. It must be fixed before being processed.

9.5 Transaction and valid times

What kind of event *exactly* do columns **start** and **end** measure? If they specify the time points at which changes occur in the application domain, that is, in the real world, they refer to the **valid time**. If they specify the time at which information on the changes is recorded in the database, they refer to the **transaction time**.

When an employee moves from a city to another one on *September 1st, 2017* (2017-09-01), and if this fact is recorded in the database twenty days later, on 2017-09-21, then 2017-09-01 is the valid time of this change while 2017-09-21 is its transaction time.

Both types of time are important. Let us consider a fact **F** that appears at time **vT** and that has been recorded in the database at time **tT**. If we consider the database as the repository of all the relevant knowledge on the application domain, then we can say that fact **F** is *officially* known from time **tT** and *officially* unknown before. This can have deep implication from the legal point of view. Ideally, we should associate both valid and transaction times to the data, that is, record **bitemporal** data. Unfortunately, the management and processing of bitemporal data is particularly complex, so that we will consider one temporal dimension at a time only.

In the next sections, we will develop simple mechanisms that automate as far as possible the management of temporal data according to each of these time dimensions.

9.6 Managing transaction time historical data

Recording transaction time events enjoys a quite interesting property. Indeed, the transaction time associated with an event typically is provided by the system clock. This means that the user has no control on the values of the timestamps, so that the time management can be fully automated. The user just has to tell: *create this new entity*, *modify these attributes of this existing entity* and *delete this existing entity*. In technical words, the data modification events will be created by issuing `insert`, `update` and `delete` queries on the views describing the current states of the entities. This way, the users can ignore all the time management mechanics.

From the SQL engine perspective, these transaction time values are obtained through register `current_timestamp`, that returns the current system time down to the millisecond. Whether this precision is appropriate depends on the pace of the events. In some cases, it will be far too detailed, for instance in administrative tasks, where a delay of one day generally is the rule. In other cases, this precision will be too coarse, as for a web server that processes several hundreds of requests per second.

Whether the application domain is subject to slow events or to fast events, this choice may entail three important consequences.

1. A high time precision requires, in the ISO format (YYYY-MM-DD HH:MM:SS:.mmm), a string of 23 bytes, that is, 46 additional bytes per state. In a fast evolving system, where an entity may be subject to thousands of modification events, timestamping states will consume a lot of space.[10]

2. Even if we choose the most fine-grained precision, there is no guarantee that two events affecting the same entity will always be assigned different timestamps. In such cases, some modifications of an entity will be recorded with the same timestamp, leading the database to lose some of these updates.

3. To simulate the evolution of systems with slow events, such as administrations, collecting real-time data to build test datasets, as we will do in these studies, would take months and even years, which would be unrealistic.

To solve these problems, we add to historical base tables H_PROJECT and H_EMPLOYEE a new table, called CLOCK (Script 9.1), in which we store the timestamps of events (in column TTime) whenever they occur. A unique integer number is assigned to these time points, acting as time id (column TimeID). Columns **start** and **end** in historical tables will contains such time ids instead of actual timestamps, whatever the granularity of the latter. Column TimeID is the primary key of CLOCK but column TTime is not unique.

10. Actually, this space depends on the internal representation of temporal data. For instance, *datetime* values can be coded as *julian day*, which itself is implemented by a floating point number spanning 8 or 16 bytes. See https://en.wikipedia.org/wiki/Julian_day for more detail. Julian day arithmetics is available as UDF functions in SQLfast.

```
create table CLOCK(
   TimeID integer not null primary key autoincrement,
   TTime  varchar(23) not null);
```

Script 9.1 - Table CLOCK stores the time points of the events

Historical table H_PROJECT and H_EMPLOYEE can now be specified (Script 9.2). Considering the nature of the application domain of this case study, representing time points by dates seems quite sufficient.

```
create table H_PROJECT(
   TITLE    char(20)   not null,
   THEME    char(22)   not null,
   BUDGET   decimal(8) not null,
   start    integer    not null,
   end      integer    not null default 999999,
   primary key (TITLE,start));

create table H_EMPLOYEE(
   CODE     char(5)  not null,
   NAME     char(10) not null,
   SALARY   integer  not null,
   CITY     char(10) not null,
   PROJECT  char(20) not null,
   start    integer  not null,
   end      integer  not null default 999999,
   primary key (CODE,start));
```

Script 9.2 - Historical tables recording the evolution of project and employee entities

Figures 9.6 and 9.7 show the evolution of project and employee entities. Table Clock that translates abstract time points into actual dates is shown in Figure 9.8. The contents of table H_PROJECT shows an interesting fact: a third project, named SURVEYOR, was started on time 12 but was closed on time 36, so that it does not exist any more and does not appear in the current state of the database (Figure 9.1).

TITLE	THEME	BUDGET	start	end
AGRO-2000	Crop improvement	65000	21	36
AGRO-2000	Crop improvement	75000	36	41
AGRO-2000	Crop improvement	82000	41	999999
BIOTECH	Biotechnology	180000	2	7
BIOTECH	Genetic engineering	160000	7	9
BIOTECH	Genetic engineering	120000	9	20
BIOTECH	Genetic engineering	140000	20	44
BIOTECH	Biotechnology	140000	44	999999
SURVEYOR	Satellite monitoring	310000	12	18

SURVEYOR	Satellite monitoring	375000	18	31
SURVEYOR	Satellite monitoring	345000	31	40

Figure 9.6 - History of PROJECT entities

CODE	NAME	SALARY	CITY	PROJECT	start	end
A237	Antoine	4000	Grenoble	AGRO-2000	24	30
A237	Antoine	3800	Grenoble	BIOTECH	30	33
A237	Antoine	3800	Grenoble	AGRO-2000	33	999999
A68	Albert	3200	Toulouse	BIOTECH	8	13
A68	Albert	3700	Toulouse	SURVEYOR	13	26
A68	Albert	3900	Paris	SURVEYOR	26	39
C45	Carlier	3100	Lille	BIOTECH	11	22
C45	Carlier	3400	Lille	AGRO-2000	22	28
C45	Carlier	3100	Lille	BIOTECH	28	999999
D107	Delecourt	3800	Grenoble	SURVEYOR	15	17
D107	Delecourt	4100	Genève	SURVEYOR	17	27
D107	Delecourt	4100	Grenoble	SURVEYOR	27	35
D107	Delecourt	3300	Grenoble	BIOTECH	35	999999
D122	Declercq	3200	Paris	BIOTECH	10	16
D122	Declercq	3800	Paris	SURVEYOR	16	37
D122	Declercq	4100	Toulouse	AGRO-2000	37	42
G96	Godin	3400	Genève	BIOTECH	3	5
G96	Godin	3300	Genève	BIOTECH	5	23
G96	Godin	3900	Genève	AGRO-2000	23	29
G96	Godin	4100	Genève	AGRO-2000	29	38
G96	Godin	3900	Genève	AGRO-2000	38	999999
M158	Mercier	2900	Paris	BIOTECH	4	6
M158	Mercier	3000	Paris	BIOTECH	6	14
M158	Mercier	3600	Paris	SURVEYOR	14	34
M158	Mercier	3000	Paris	BIOTECH	34	999999
N240	Nguyen	3100	Toulouse	BIOTECH	19	25
N240	Nguyen	3700	Grenoble	SURVEYOR	25	32
N240	Nguyen	3700	Genève	AGRO-2000	32	43

Figure 9.7 - History of EMPLOYEE entities

TimeID	TTime
1	2017-07-24
2	2017-07-25
3	2017-07-27
4	2017-07-29
5	2017-07-30
6	2017-08-02
7	2017-08-06
8	2017-08-09
9	2017-08-11
10	2017-08-12
11	2017-08-14
...	...
43	2017-10-26
44	2017-10-27

Figure 9.8 - Table CLOCK records the time points of all the events and assigns them a unique time Id

9.6.1 Current state of entities

Live entities are those that have an *active current state*, with an *infinite future* end time, that is, according to our convention, with TimeID = 999999. So, they are quite easy to select through the SQL views of Script 9.3. The current states of our example database are shown in Figures 9.9 and 9.10.

```
create view PROJECT(TITLE,THEME,BUDGET)
as select TITLE,THEME,BUDGET
   from   H_PROJECT
   where  end = 999999;

create view EMPLOYEE(CODE,NAME,STATUS,CITY,PROJECT)
as select CODE,NAME,STATUS,CITY,PROJECT
   from   H_EMPLOYEE
   where  end = 999999;
```

Script 9.3 - Extracting the current states

TITLE	THEME	BUDGET
AGRO-2000	Crop improvement	82000
BIOTECH	Biotechnology	140000

Figure 9.9 - The current state of PROJECT entities

CODE	NAME	SALARY	CITY	PROJECT
A237	Antoine	3800	Grenoble	AGRO-2000
C45	Carlier	3100	Lille	BIOTECH
D107	Delecourt	3300	Grenoble	BIOTECH
G96	Godin	3900	Genève	AGRO-2000
M158	Mercier	3000	Paris	BIOTECH

Figure 9.10 - The current state of EMPLOYEE entities

The main interest of these views is that users will be able to manage data, that is, *create*, *modify* and *destroy* entities, by merely executing `insert`, `update` and `delete` SQL queries **on these views**, and not on the underlying historical tables.

9.6.2 Creating an entity

The nice property of transaction time management is that all data modification actions can be performed on the current state of entities without worrying about complicated temporal manipulations. Therefore, creating a new entity, cannot be simpler:

```
insert into PROJECT(TITLE,THEME,BUDGET)
    values ('BIOTECH','Biotechnology',180000);

insert into EMPLOYEE(CODE,NAME,STATUS,CITY,PROJECT)
    values ('G96','Godin','F','Genève','BIOTECH');
```

The effect of such a query is to insert in table H_PROJECT (same for H_EMPLOYEE) the following row:

```
'BIOTECH','Biotechnology',180000,2,999999)
```

Value start = 2 is the TimeID value returned by the insertion of the current time in table CLOCK and end=999999 tells that this state is active.

Now, we have to consider what must happen *under the hood*. The implementation of `insert` operations will be coded in `instead of` triggers operating on views PROJECT and EMPLOYEE.

Let us begin with the creation of ***project** entities*, that will be recorded by trigger `TRG_PRO_INSERT`. Its body must execute three actions.

1. Checking that there is no known entity with the same entity Id.

   We verify that there is no state, past or current, in H_PROJECT with the same value of TITLE as the one we intend to insert, otherwise, we raise an exception that cancels the insert operation:

   ```
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   ```

2. Creating a time event and storing it in table CLOCK.

   Theoretically, we would just store in table CLOCK the value of register `current_timestamp` or `current_date`:

   ```
   insert into CLOCK(TTime) values(current_date);
   ```

   This would be appropriate in a real temporal database, with real events captured in real time, but certainly not in our modest case study! So, we will generate artificial, but realistic time points. For instance through the following query:

   ```
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5))from CLOCK;
   ```

   Function `random_i(n1,n2)` returns a random integer in interval [**n1**,**n2**] and function `addToDate(d,n)` returns date **d** augmented by **n** days. This query computes the most recent value of TTime in CLOCK, adds to it a random number of days then inserts the result in CLOCK.[11]

3. Inserting data in H_PROJECT with the time ID just created.

   The **end** column is automatically set to 999999 through it `default` clause.

11. Of course, table CLOCK must be initialized with something like:
`insert into CLOCK(TTime) values ('2017-07-24');`.

```
insert into H_PROJECT(TITLE,THEME,BUDGET,start)
values (new.TITLE,new.THEME,new.BUDGET,
        (select max(TimeID) from CLOCK));
```

We are now ready to code this trigger (Script 9.4).

```
create trigger TRG_PRO_INSERT instead of insert on PROJECT
begin
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)
      values(new.TITLE,new.THEME,new.BUDGET,
             (select max(TimeID) from CLOCK));
end;
```

Script 9.4 - Trigger controlling the **creation** of a new **project** (transaction time)

Managing the creation of employee entities is similar to that of projects, with an additional concern: controlling foreign key PROJECT. Indeed, the value of column PROJECT of the new row must reference an active PROJECT entity. Hence the statement:

```
if not exists(select * from H_PROJECT
              where  TITLE = new.PROJECT
              and    end = 999999)
   raise_exception('target PROJECT does not exist');
```

The complete trigger is shown in Script 9.5.

Two comments

1. Our case study does not include *unique keys*. Their control can be less strict than that of primary key. Verifying that there is no active entity with the same value of the unique key may be sufficient:

   ```
   if exists(select * from H_PROJECT
             where THEME = new.THEME and end = 999999)
   raise_exception('duplicate unique key');
   ```

2. The way triggers are coded is strongly dependent on the DBMS. For example, the body of an SQLite trigger is a pure sequence of SQL queries. The exception itself is specified through function `raise()` in a `select` query:

   ```
   select raise(ABORT,'duplicate entity PK')
   where  exists(select * from H_PROJECT
                 where  TITLE = new.TITLE);
   ```

```
create trigger TRG_EMP_INSERT instead of insert on EMPLOYEE
begin
   if exists(select * from H_EMPLOYEE where CODE = new.CODE)
      raise_exception('duplicate Entity PK');
   if not exists(select * from H_PROJECT
                 where  TITLE = new.PROJECT
                 and    end = 999999)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
      values(new.CODE,new.NAME,new.SALARY,new.CITY,
             new.PROJECT,(select max(TimeID) from CLOCK));
end;
```

Script 9.5 - Trigger controlling the **creation** of a new **employee** (transaction time)

9.6.3 Evolution of an entity

To modify the state of an entity, a simple, non temporal `update` query will suffice:

```
update PROJECT
set THEME = 'Genetic engineering',
    BUDGET = 160000
where TITLE = 'BIOTECH';

update EMPLOYEE
set PROJECT = 'AGRO-2000',
    SALARY = 3900
where CODE = 'G96';
```

Let us suppose that

1. the *update* of project BIOTECH occurs at a time point with TimeID = 7
2. the current state of this project, just before the modification, is the following:

   `('BIOTECH','Biotechnology',180000,2,999999)`

The translation of the modification requires two actions. First we close the current state by modifying its **end** column:

`('BIOTECH','Biotechnology',180000,2,7)`

Then we insert a new current state which starts at time point 7:

`('BIOTECH','Genetic engineering',160000,7,999999)`

Now, the history of project BIOTECH comprises two states. The implementation of `update` operations will be coded in `instead of` triggers acting on views PROJECT and EMPLOYEE.

The trigger controlling the evolution of projects, named `TRG_PRO_UPDATE`, must perform five actions.

1. Verifying that the entity is active.

   No action is required: only active states are visible through view PROJECT. Since no modification is applied to a *non-existent entity*, no exception can be fired in this case.

2. Verifying that at least one attribute is modified.

   Quite straightforward:

   ```
   if new.THEME = old.THEME and new.BUDGET = old.BUDGET
      raise_exception('no attribute modified');
   ```

3. Creating a time event and storing it in table CLOCK.

   Same as for `insert`.

4. Close the current state

   The Id of the time point just created is extracted and assigned to column **end**:

   ```
   update H_PROJECT
   set    end = (select max(TimeID) from CLOCK)
   where  TITLE = old.TITLE and end = 999999;
   ```

5. Insert a new current state.

   Same as for `insert`.

   ```
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)

   values(old.TITLE,new.THEME,new.BUDGET,
          (select max(TimeID) from CLOCK));
   ```

   We notice that the value of column TITLE (the primary key) is left unchanged (old.TITLE), whatever the new value possibly provided by the user.

The complete trigger is given in Script 9.6.

The trigger controlling the evolution of employees is similar to that of projects but it must cope with a new constraint: is a new project is specified for the employee, this project must exist and be active:

```
if new.PROJECT <> old.PROJECT
and not exists(select * from H_PROJECT
               where  TITLE = new.PROJECT
               and    end = 999999)
raise_exception('target PROJECT does not exist');
```

The trigger for EMPLOYEE is shown in Script 9.7.

```
create trigger TRG_PRO_UPDATE instead of update on PROJECT
begin
   if new.THEME = old.THEME and new.BUDGET = old.BUDGET
      raise_exception('no attribute modified');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   update H_PROJECT
   set    end = (select max(TimeID) from CLOCK)
   where  TITLE = old.TITLE and end = 999999;
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)
      values(old.TITLE,new.THEME,new.BUDGET,
             (select max(TimeID) from CLOCK));
end;
```

Script 9.6 - Trigger controlling the **modification** of a **project** (transaction time)

```
create trigger TRG_EMP_UPDATE instead of update on EMPLOYEE
begin
   if     new.NAME = old.NAME and new.SALARY  = old.SALARY
      and new.CITY = old.CITY and new.PROJECT = old.PROJECT
      raise_exception('no attribute modified');
   if new.PROJECT <> old.PROJECT
      and not exists(select * from H_PROJECT
                     where  TITLE = new.PROJECT
                     and    end = 999999)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   update H_EMPLOYEE
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
      values(old.CODE,new.NAME,new.SALARY,new.CITY,
             new.PROJECT,(select max(TimeID) from CLOCK));
end;
```

Script 9.7 - Trigger controlling the **modification** of an **employee** (transaction time)

9.6.4 Deleting an entity

Deleting an entity follows the standard technique, once again applied to view PROJECT and EMPLOYEE:

```
delete from PROJECT where TITLE = 'SURVEYOR';

delete from EMPLOYEE where CODE = 'D122';
```

Let us suppose that the current state of project SURVEYOR is the following:

```
('SURVEYOR','Satellite monitoring',345000,29,999999)
```

To delete this project (actually to *close* it) at time point 36, we just modify its **end** column to give it value 36:

```
('SURVEYOR','Satellite monitoring',345000,29,36)
```

However, the foreign key of EMPLOYEE dictates specific constraints on delete operations on project entities. The current state of PROJECT can be closed only if the referential constraint will be validated at operation completion. If some employees still are working on the project we intend to delete, we must first take care of them: either by deleting them as well or by preventing the project to be deleted, according to the *delete mode* declared for this foreign key. If we choose the `no action` delete mode, then we must first check that there is no employee working of project SURVEYOR any more:

```
if exists(select * from H_EMPLOYEE
          where  PROJECT = old.TITLE and end = 999999)
   raise_exception('dependent EMPLOYEEs still exist');
```

The code of the trigger that deletes PROJECT entities is shown in Script 9.8. The trigger for EMPLOYEE entities is similar (not shown).

It is interesting to notice that the last state of a deleted entity provides two informations: one about the last update of the entity (or on its creation) and the other one about its deletion.

```
create trigger TRG_PRO_DELETE instead of delete on PROJECT
begin
   if exists(select * from H_EMPLOYEE
             where  PROJECT = old.TITLE
             and    end = 999999)
      raise_exception('dependent EMPLOYEEs still exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   update H_PROJECT
   set    end = (select max(TimeID) from CLOCK)
   where  TITLE = old.TITLE and end = 999999;
end;
```

Script 9.8 - Trigger controlling the **deletion** of a **project** (transaction time)

9.7 Managing valid time historical data

The term *valid time* refers to the time as it flows in the application domain, that is, in the *real world*.

Let us study the behavior of temporal data that describe what happens in the real world. We can keep the same data structures as in Section 9.6 to store temporal data except for the nature of columns **start** and **end** which are of type `date` (or `datetime` or `timestamp`, according to the time granularity) instead of `integer`. Now, the values of **start** and **end** are those provided by users. They will need additional checking and control.

To make the process manageable, we will impose a constraint on the temporal sequence of the events affecting each entity:[12]

> the state changes of an entity are recorded in **chronological order** of their occurrence. Indeed, notifying a change that occurred before the last recorded event will require a potentially complex reorganization of the history of the entity in which some closed states could have to be deleted, split or merged.

9.7.1 Current state of entities

We still use constant 9999-12-31 to indicate the *infinite future* that will be assigned to the **end** column to indicate *current states*. The SQL views declared in Script 9.9 extract the description of the current project and employee entities[13]. The latter are shown in Figures 9.11 and 9.12. Columns **start** and **end** now are assigned user data and therefore are visible.

```
create view PROJECT(TITLE,THEME,BUDGET,start,end)
as select TITLE,THEME,BUDGET,start,end
   from   H_PROJECT
   where  end = '9999-12-31';

create view EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start,end)
as select CODE,NAME,SALARY,CITY,PROJECT,start,end
   from   H_EMPLOYEE
   where  end = '9999-12-31';
```

Script 9.9 - Extracting the current states

12. This constraint can be (partially) removed in other forms of historical data, for instance those based on individual attribute history. Anyway, correctly coping with this scenarios requires a bitemporal database.
13. These views **cannot** be declared `with check option`! Why?

TITLE	THEME	BUDGET	start	end
AGRO-2000	Crop improvement	82000	2017-07-05	9999-12-31
BIOTECH	Biotechnology	140000	2017-09-17	9999-12-31

Figure 9.11 - The current state of PROJECT entities (valid time)

CODE	NAME	SALARY	CITY	PROJECT	start	end
A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31
C45	Carlier	3100	Lille	BIOTECH	2016-12-11	9999-12-31
D107	Delecourt	3300	Grenoble	BIOTECH	2017-04-03	9999-12-31
G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31
M158	Mercier	3000	Paris	BIOTECH	2017-01-14	9999-12-31

Figure 9.12 - The current state of EMPLOYEE entities (valid time)

9.7.2 Creating of an entity

To create an entity, we just execute an **insert** query on the PROJECT or EMPLOYEE view:

```
insert into PROJECT(TITLE,THEME,BUDGET,start)
   values ('BIOTECH','Biotechnology',180000,'2014-11-18');
insert into EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
   values ('G96','Godin',3400,'Genève','BIOTECH','2014-11-23');
```

The effect of the first query is to insert in table H_PROJECT the following row, in which value end = 9999-12-31 tells that this state is current:

```
('BIOTECH','Biotechnology',180000,'2014-11-18','9999-12-31')
```

The *project* entity we intend to create must have a primary key value that has never been used before. Since the event date is provided by the user, we must check its validity, through function `isDate(d)`, that indicates whether value **d** is a valid date. The trigger that controls the creation of PROJECT entities is shown in Script 9.10.

The creation of an EMPLOYEE entity needs an additional checking: the PROJECT entity referenced by the value of column PROJECT must exist and be active. The trigger associated with the creation of PROJECT entities is shown in Script 9.11.

9.7.3 Evolution of an entity

The state of an entity is modified through an update query on views PROJECT or EMPLOYEE:

```
create trigger TRG_PRO_INSERT instead of insert on PROJECT
begin
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   if not isDate(new.start)
      raise_exception('invalid start date');
   insert into H_PROJECT(TITLE,THEME,BUDGET,start)
   values (new.TITLE,new.THEME,new.BUDGET,new.start);
end;
```

Script 9.10 - Trigger controlling the **creation** of a new **project** (valid time)

```
create trigger TRG_EMP_INSERT instead of insert on EMPLOYEE
begin
   if exists(select * from H_EMPLOYEE where CODE = new.CODE)
      raise_exception('duplicate entity PK');
   if isNotDate(new.start)
      raise_exception('invalid start date');
   if not exists(select * from H_PROJECT
                 where  TITLE = new.PROJECT
                 and    end = '9999-12-31');
      raise_exception('target PROJECT does not exist');
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
   values (new.CODE,new.NAME,new.SALARY,
           new.CITY,new.PROJECT,new.start);
end;
```

Script 9.11 - Trigger controlling the **creation** of a new **employee** (valid time)

```
update PROJECT
set start = '2015-02-27',
    THEME = 'Genetic engineering', BUDGET = 160000
where TITLE = 'BIOTECH';

update EMPLOYEE
set start = '2017-05-19', SALARY = 3900
where CODE = 'G96';
```

The execution of these queries requires two actions. First closing the current state by modifying its **end** column:

```
('BIOTECH','Biotechnology',180000,
 '2014-11-18','2015-02-27')
```

then inserting a new current state which starts at time point `2015-02-27`:

```
('BIOTECH','Genetic engineering',160000,
 '2015-02-27','9999-12-31')
```

Checking the value of column **start** is a bit more complex than that required in `insert` queries. Not only this data must be syntactically correct (function **`isDate`**) but the *chronological order* must be satisfied. This means that the new value of **start** must be greater than that of the current state of the entity. Technically:

```
if not isDate(new.start)
   or new.start <= (select start from H_PROJECT
                    where  TITLE = new.TITLE
                    and    end = '9999-12-31')
   raise_exception('invalid start date');
```

Controlling *referential integrity* when updating an employee also is a bit more tricky than for transaction time. Of course, the target project must be active but this is not sufficient: this project must also have existed at the **start** time of the updated employee. For example, if the employee is claimed to work on project AGRO-2000 since 2016-08-01, this update must be rejected because this project started on 2016-08-31 only.

```
if new.PROJECT <> old.PROJECT
   and not (exists(select * from H_PROJECT
                   where  TITLE = new.PROJECT
                   and    end = '9999-12-31')
            and new.start >= (select min(start)
                              from   H_PROJECT
                              where  TITLE = new.PROJECT))
   raise_exception('target PROJECT does not exist');
```

As in the transaction time triggers, the value of the entity id (PROJECT.TITLE and EMPLOYEE.CODE) must be preserved, even if the user tries to modify it.

Suggested triggers controlling the evolution of projects and employees are shown in Scripts 9.12 and 9.13. The role of the `when` clause will be explained in the next section.

```
create trigger TRG_PRO_UPDATE instead of update on PROJECT
when new.end = '9999-12-31'
begin
   if new.THEME = old.THEME and new.BUDGET = old.BUDGET
      raise_exception('no attribute modified');
   if isNotDate(new.start)
      or new.start <= (select start from H_PROJECT
                       where  TITLE = new.TITLE
                       and    end = '9999-12-31')
      raise_exception('invalid start date');                ./..
```

```
   update H_PROJECT
   set    end = new.start
   where  TITLE = old.TITLE and end = '9999-12-31';
   insert into H_PROJECT(TITLE,start,THEME,BUDGET)
   values (old.TITLE,new.start,new.THEME,new.BUDGET);
end;
```

Script 9.12 - Trigger controlling the **modification** of a **project** (valid time)

```
create trigger TRG_EMP_UPDATE instead of update on EMPLOYEE
when new.end = '9999-12-31'
begin
   if new.NAME = old.NAME and new.SALARY = old.SALARY
      and new.CITY = old.CITY and new.PROJECT = old.PROJECT
      raise_exception('no attribute modified');
   if isNotDate(new.start)
      or new.start <= (select start from H_EMPLOYEE
                       where  CODE = new.CODE
                       and    end = '9999-12-31')
      raise_exception('invalid start date');
   if new.PROJECT <> old.PROJECT
      and not (exists(select * from H_PROJECT
                      where  TITLE = new.PROJECT
                      and    end = '9999-12-31')
               and new.start >= (select min(start)
                                 from   H_PROJECT
                                 where  TITLE = new.PROJECT))
      raise_exception('target PROJECT does not exist');
   update H_EMPLOYEE
   set    end = new.start
   where  CODE = old.CODE and end = '9999-12-31';
   insert into H_EMPLOYEE(CODE,NAME,SALARY,CITY,PROJECT,start)
   values (old.CODE,new.NAME,new.SALARY,new.CITY,new.PROJECT,
           new.start);
end;
```

Script 9.13 - Trigger controlling the **modification** of an **employee** (valid time)

9.7.4 Deleting an entity

Deleting an entity does not follow the technique developed for transaction time. Indeed, the query must convey a user data, namely the deletion date, with which the current state will be closed. Therefore a `delete` query cannot work:

```
delete PROJECT where TITLE = 'SURVEYOR';
```

Instead, we will use a special version of the `update` query, that only changes the value of the **end** attribute of the entity:

```
update PROJECT set end = '2017-05-19' where TITLE = 'SURVEYOR';
```

Let us suppose that the last current state of project SURVEYOR is the following:

```
('SURVEYOR','Satellite monitoring',345000,
 '2016-12-20','9999-12-31')
```

If this project was closed on May 19, 2017, then its new last state will be:

```
('SURVEYOR','Satellite monitoring',345000,
 '2016-12-20','2017-05-19')
```

Like in project updating, we must check the validity of the `end` date. In addition, we verify than no active employee is still working on this project any more. All this is implemented in trigger `TRG_PRO_DELETE` (Script 9.14).

```
create trigger TRG_PRO_DELETE instead of update on PROJECT
when new.end < '9999-12-31'
begin
   if isNotDate(new.end)
      or new.end <= (select start from H_PROJECT
                     where  TITLE = new.TITLE
                     and    end = '9999-12-31')
      raise_exception('invalid end date');
   if exists(select * from H_EMPLOYEE
             where  PROJECT = old.TITLE
             and    end = '9999-12-31')
      raise_exception('dependent EMPLOYEEs still exist');
   update H_PROJECT
   set    end = new.end
   where  TITLE = new.TITLE and end = '9999-12-31';
end;
```

Script 9.14 - Trigger controlling the **deletion** of a **project** (valid time)

9.8 From entity-based to tuple-based history

The discussion of entity-based temporal data can be generalized to more abstract patterns in which the data describe the history of a series of time-varying columns, whatever the meaning one assigns to this assembly.

For example, a row may represent a relationship between two (or more) entities that held during a specified period. This row would comprise the primary keys of the entities, possibly some additional data associated with this relationship, and an interval. Let us suppose that an employee can at any time work on more than one project. The schema of our example database must then be modified: column PROJECT of H_EMPLOYEE is removed and replaced by a new table, called H_WORK_ON, with columns (CODE,TITLE,start,end) and primary key (CODE, TITLE,start).

As another example, we could extract from table H_EMPLOYEE columns (PROJECT,CITY,start,end) that tells from which cities the employees who worked on which project came and during which periods.[14]

To cope with such various interpretation of temporal data, we qualify such temporal table *tuple-based*.[15]

9.9 Alternative temporal database models

The data organization of temporal databases that we have developed in this study (*entity-based history*) is both intuitive and easy to manage. As we will show in the next part of this study, it is also (fairly) easy to process. However, there are quite a lot of alternative ways to organize historical data in a database. We will mention and briefly discuss some of them.

9.9.1 Table horizontal splitting

If processing the current states of the entities is the highest priority (anyway, it is the most frequent use of a database), then it can be more efficient to split each historical table into a main table (called, say, EMPLOYEE) that contains all the current states and another, secondary table (called H_EMPLOYEE) in which all past states are stored.[16]

This way, the main table will be managed and processed in a much more efficient way: smaller table, shorter scans, smaller and faster indexes.

14. This extraction will be discussed in the second part of this study under the name *temporal projection*.
15. A *tuple* is a data structure formed by a series of **t** values (kind of generalization of cou***ple***, tri***ple***, etc.) *Tuple* is the theoretical name of *row* in a table.
16. This is the way several DBMS implement transaction time historical data.

This distribution of current and past states in two different tables may make temporal processing more complex. A straightforward solution consists in recording current states both in the historical table and in the current state table.

9.9.2 Table vertical splitting

In many historical tables, data management maintains the history of some columns, called *time-varying* columns, while for the others, called *constant columns*, only the last state is recorded, because they are never updated or because there is no need to keep their history.

In an entity-based history table, the values of constant columns of an entity are duplicated in all the state rows of this entity. To reduce this waste of space, we can collect all the constant columns, included the components of the primary key, in a table while the other columns (together with the components of the primary key) are gathered into a standard temporal table.

9.9.3 Attribute-based history

The *Entity-based* organization has a major drawback: when one attribute changes, all the values of the other attributes are copied in the new current state, therefore creating important redundancies. In the *attribute-based* organization, the history of each attribute is stored in an independent historical table. When an attribute changes, only its historical table is affected. This model can be studied as the *Column-oriented data model*, discussed in study *Schema-less databases - Part 1*, to which a temporal dimension is added.

The history of the employee population is implemented into four tables (as many as there are attributes, primary key excluded), two of which are represented in Figure 9.13.

CODE	SALARY	start	end
A237	4000	24	30
A237	3800	30	999999
A68	3200	8	13
A68	3700	13	26
A68	3900	26	39
C45	3100	11	22
C45	3400	22	28
C45	3100	28	999999
D107	3800	15	17
D107	4100	17	35
D107	3300	35	999999
D122	3200	10	16
D122	3800	16	37
D122	4100	37	42
G96	3400	3	5
G96	3300	5	23
G96	3900	23	29
G96	4100	29	38

CODE	CITY	start	end
A237	Grenoble	24	999999
A68	Toulouse	8	26
A68	Paris	26	39
C45	Lille	11	999999
D107	Grenoble	15	17
D107	Genève	17	27
D107	Grenoble	27	999999
D122	Paris	10	37
D122	Toulouse	37	42
G96	Genève	3	999999
M158	Paris	4	999999
N240	Toulouse	19	25
N240	Grenoble	25	32
N240	Genève	32	43

```
| G96  | 3900   | 38    | 999999 |
| M158 | 2900   | 4     | 6      |
| M158 | 3000   | 6     | 14     |
| M158 | 3600   | 14    | 34     |
| M158 | 3000   | 34    | 999999 |
| N240 | 3100   | 19    | 25     |
| N240 | 3700   | 25    | 43     |
+------+--------+-------+--------+
```

Figure 9.13 - Independent historical tables H_SALARY and H_CITY store the evolution of two attributes of employees.

The triggers that control the creation and deletion of entities are fairly straightforward. Scripts 9.15, 9.16, 9.17 manage the historical attribute tables according to the transaction time dimension. When the condition of execution are met, the operation is distributed among all the historical tables. Adaptation to valid time is as easy.

We notice that, in the trigger controlling the `update` operation (Script 9.16), only attribute tables corresponding to a change are affected.

```
create trigger TRG_EMP_INSERT instead of insert on EMPLOYEE
begin
   if exists(select * from H_NAME where CODE = new.CODE)
      raise_exception('duplicate entity PK');
   if not exists(select * from H_THEME
                 where  TITLE = new.PROJECT
                 and    end = $future$)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   ...
   insert into H_STATUS(CODE,SALARY,start) values
      (new.CODE,new.SALARY,(select max(TimeID) from CLOCK));
   insert into H_CITY(CODE,CITY,start) values
      (new.CODE,new.CITY,(select max(TimeID) from CLOCK));
   ...
end;
```

Script 9.15 - Inserting the data of a new **employee** in an attribute-based historical database (transaction time)

```
create trigger TRG_EMP_UPDATE instead of update on EMPLOYEE
when new.NAME <> old.NAME or new.SALARY  <> old.SALARY
  or new.CITY <> old.CITY or new.PROJECT <> old.PROJECT
                                                    ./...
```

```
begin
   if new.PROJECT <> old.PROJECT
      and not exists(select * from H_THEME
                     where  TITLE = new.PROJECT
                     and    end = $future$)
      raise_exception('target PROJECT does not exist');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   ...
   if new.SALARY <> old.SALARY begin
      update H_SALARY
      set    end = (select max(TimeID) from CLOCK)
      where  CODE = old.CODE and end = 999999;
      insert into H_SALARY(CODE,SALARY,start) value
         (old.CODE,new.SALARY,(select max(TimeID) from CLOCK));
   end;
   if new.CITY <> old.CITY begin
      update H_CITY
      set    end = (select max(TimeID) from CLOCK)
      where  CODE = old.CODE and end = 999999;
      insert into H_CITY(CODE,CITY,start) values
         (old.CODE,new.CITY,(select max(TimeID) from CLOCK));
   end;
   ...
end;
```

Script 9.16 - Updating the data of an **employee** in an attribute-based historical database (transaction time)

```
create trigger TRG_EMP_DELETE
instead of delete on EMPLOYEE
begin
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   ...
   update H_SALARY
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   update H_CITY
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   ...
end;
```

Script 9.17 - Deleting the data of an **employee** in an attribute-based historical database (transaction time)

Alternative trigger organization

In each of these triggers, all the operations on historical tables are gathered into a single piece of code. A quite different organization could be thought of, in which a specific *update trigger* is associates with each historical table (Script 9.18). The change events are filtered with a `when` clause so that only the triggers of the tables in which a change is required will fire. Of course, we must first get a new timestamp from table CLOCK. This can be done with an additional trigger that just includes statement `insert into CLOCK(TTime)` and that fires when at least one change is observed.

Unfortunately, though this distributed organization may appear quite elegant, its execution protocol may be less obvious. Indeed, we have created N + 1 triggers, where N is the number of attributes that may change, and these triggers are of the same type: same table (view EMPLOYEE), same event (`update`), same position (`instead of`).

First, not all DBMS allow multiple triggers of the same kind. Secondly, the trigger updating table CLOCK must fire **before** all the others. This means that the DBMS must provide some way to specify in which order triggers of the same type will fire. Most DBMS provide it, though often in non standard ways,[17] except SQLite, in which this order is arbitrary or undetermined.

Converting an *entity-based* history table to the *attribute-based* format is fairly easy but requires a special operator that will be studied in the second part on the case study: the *temporal projection*. The inverse conversion will be performed through a *temporal join*, an operator that also will be described in the next part.

```
create trigger TRG_SALARY_UPDATE instead of insert on EMPLOYEE
when new.SALARY <> old.SALARY
begin
   update H_STATUS
   set    end = (select max(TimeID) from CLOCK)
   where  CODE = old.CODE and end = 999999;
   insert into H_SALARY(CODE,SALARY,start) values
      (old.CODE,new.STATUS,(select max(TimeID) from CLOCK));
end;
```

Script 9.18 - The trigger controlling changes of attribute **Status** of **employee** entities, in a distributed trigger organization

17. In some DBMS, the order is explicitly specified in the DDL code of the trigger (Oracle [from 11g], InterBase and, to some extent, SQL Server) while in others, the order is that of trigger creation (SQL standard, DB2, MySQL) or the alphabetical order of trigger names (PostgreSQL).

9.9.4 Event-based history

If we look at Figure 9.14 (copy of Figure 9.3), in which all the modification events affecting project BIOTECH are positioned on the timeline, we could wonder why we have not merely recorded these events instead of the successive states of the project. This is what we develop in this section.

Figure 9.15 shows how events are described in what we could call the *event-based history* of project BIOTECH. For each event, we record the entity primary key (TITLE), the values of other attributes as they result from the operation, the time point of the event and the nature of the operation. This fragment of history shows that project BIOTECH has been created in time 2 then updated three times. We know that the project still is active because no delete operation has been recorded yet.

The event-based history of all the projects of our study is shown in Figure 9.16.

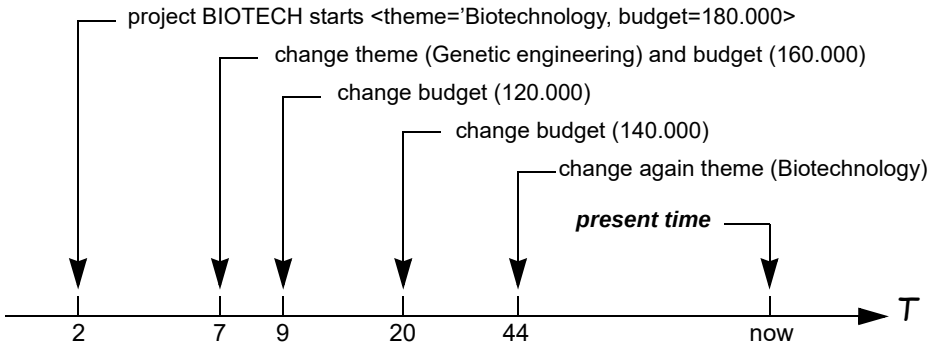


Figure 9.14 - History of project BIOTECH (copy of Figure 9.3)

BIOTECH	Biotechnology	180000	2	create
BIOTECH	Genetic engineering	160000	7	update
BIOTECH	Genetic engineering	120000	9	update
BIOTECH	Genetic engineering	140000	20	update
BIOTECH	Biotechnology	140000	44	update

Figure 9.15 - The event-based history of project BIOTECH

TITLE	THEME	BUDGET	time	oper
AGRO-2000	Crop improvement	65000	21	create
AGRO-2000	Crop improvement	75000	36	update
AGRO-2000	Crop improvement	82000	41	update
BIOTECH	Biotechnology	180000	2	create
BIOTECH	Genetic engineering	160000	7	update
BIOTECH	Genetic engineering	120000	9	update
BIOTECH	Genetic engineering	140000	20	update
BIOTECH	Biotechnology	140000	44	update
SURVEYOR	Satellite monitoring	310000	12	create
SURVEYOR	Satellite monitoring	375000	18	update

```
| SURVEYOR  | Satellite monitoring | 345000 | 31   | update |
| SURVEYOR  | Satellite monitoring | 345000 | 40   | delete |
+-----------+----------------------+--------+------+--------+
```

Figure 9.16 - The event-based history of all the projects

The event-based history of an entity type is controlled by three triggers as usual. The trigger of Script 9.19 controls the creation of project entities with transaction time dimension.

```
create trigger TRG_PRO_INSERT instead of insert on PROJECT
begin
   if exists(select * from H_PROJECT where TITLE = new.TITLE)
      raise_exception('duplicate entity PK');
   insert into CLOCK(TTime)
      select addToDate(max(TTime),random_i(0,5)) from CLOCK;
   insert into H_PROJECT(TITLE,THEME,BUDGET,time,oper) values
      (new.TITLE,new.THEME,new.BUDGET,
       (select max(TimeID) from CLOCK),'create');
end;
```

Script 9.19 - Controlling the creation of an entity in the **event-based** history model (transaction time)

9.9.5 Document-oriented history

This organization is derived from the *object model* (also called *document-oriented model*) described in study *Schema-less databases - Part 3*. It can be applied to the *entity-based* and *attribute-based* models. The idea is to associate with each attribute (except those forming the primary key) the list of values it has taken, each of them completed by the list of the intervals during which it took this value.

Let us consider the example of the employee entity with CODE = `C45`. From its creation (on **11**) until time **22** (excluded), it was assigned salary **`3100`**. On time **22**, this salary was changed to **`3400`**. This value then changed to **`3100`** on **28** and is still valid now (**999999**). So, the value of SALARY was **`3100`** in intervals `[11,22]` and `[28,999999]` and **`3400`** in interval `[22,28]`. If we encode these facts in JSON, we create a description that looks like this:

```
[{"value":3100,"intervals":[[11,22],[28,999999]]},
 {"value":3400,"intervals":[[22,28]]}]
```

This description is an array of JSON objects, each of them comprising two attributes, namely "**value**" and "**intervals**". Attribute "**value**" specifies the salary value while attribute "**intervals**" is an array of intervals. Each interval is an array of two time points. Figure 9.17 shows the state of all the temporal attributes of employee entity **`D107`** from its creation until now.

```
CODE:   D107
NAME:   [{"value":"Delecourt", "intervals":[[15,999999]]}]
SALARY: [{"value":"3800",      "intervals":[[15,17]]},
         {"value":"4100",      "intervals":[[17,35]]}]
         {"value":"3300",      "intervals":[[35,999999]]}]
CITY:   [{"value":"Grenoble",  "intervals":[[15,17],[27,999999]]},
         {"value":"Genève",    "intervals":[[17,27]]}]
PROJECT:[{"value":"SURVEYOR",  "intervals":[[15,35]]},
         {"value":"BIOTECH",   "intervals":[[35,999999]]}]
```

Figure 9.17 - Temporal attributes of employee entity D107 represented by JSON objects

Figure 9.18 is a representation of the history of employee entities according to the document-oriented model. Each row of this table contains the complete history of an employee entity. To make data more readable, we have replaced the fairly cumbersome JSON syntax by a lighter, though still intuitive, format. In addition, infinite future `'999999'` has been replaced by `'--'`.

CODE	**NAME**	**Salaries**	**Cities**	**Projects**
A237	Antoine	4000 [24,30]	Grenoble [24,--]	AGRO-2000 [24,30]
		3800 [30,--]		[33,--]
				BIOTECH [30,33]
A68	Albert	3200 [8,13]	Toulouse [8,26]	BIOTECH [8,13]
		3700 [13,26]	Paris [26,39]	SURVEYOR [13,39]
		3900 [26,39]		
C45	Carlier	3100 [11,22]	Lille [11,--]	BIOTECH [11,22]
		[28,--]		[28,--]
		3400 [22,28]		AGRO-2000 [22,28]
D107	Delecourt	3800 [15,17]	Grenoble [15,17]	SURVEYOR [15,35]
		4100 [17,35]	[27,--]	BIOTECH [35,--]
		3300 [35,--]	Genève [17,27]	
D122	Declercq	3200 [10,16]	Paris [10,37]	BIOTECH [10,16]
		3800 [16,37]	Toulouse [37,42]	SURVEYOR [16,37]
		4100 [37,42]		AGRO-2000 [37,42]
G96	Godin	3400 [3,5]	Genève [5,--]	BIOTECH [3,23]
		3300 [5,22]		AGRO-2000 [23,--]
		3900 [22,29]		
		[38,--]		
		4100 [29,38]		
M158	Mercier	2900 [4,6]	Paris [4,--]	BIOTECH [4,14]
		3000 [6,14]		[34,--]
		[34,--]		SURVEYOR [14,34]
		3600 [14,34]		
N240	Nguyen	3100 [19,25]	Toulouse [19,25]	BIOTECH [19,25]
		3700 [25,43]	Grenoble [25,32]	SURVEYOR [25,32]
			Genève [32,43]	AGRO-2000 [32,43]

Figure 9.18 - Document-oriented history of employee entities (transaction time)

The document-oriented format of historical data can be derived from the entity-based and attribute-based data, and conversely. The conversion algorithms will be developed in the second part of this case study.

9.10 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory SQLfast/Scripts/Case-Studies/Case_Temporal_DB. Actually, they can be run from main script **TDB-MAIN.sql**, that displays the selection box of Figure 9.19 (the missing items will be added in the second part of this study).

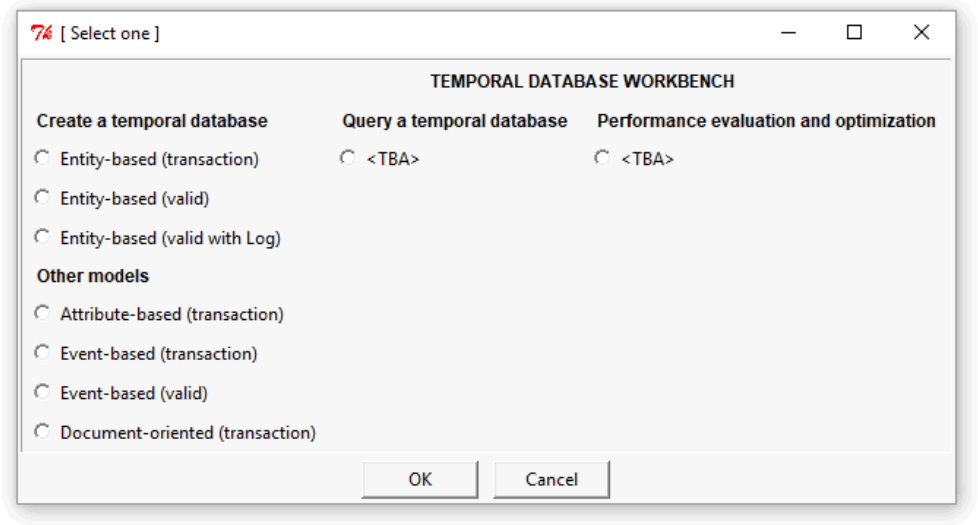


Figure 9.19 - Selecting a temporal data model

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to let them develop their own applications.

9.11 Some references

The scientific community has been very active on temporal databases since the eighties and has produced thousands of contributions related to concepts, models and query languages. Richard Snodgrass and Christopher Jensen have been among the most prolific contributors, notably through the TSQL2 language, an SQL extension addressing the temporal dimension of data. Some of the concepts of this language have been progressively (and very slowly!) incorporated into the SQL standards. Nowadays, most major DBMS have extended their SQL offering with temporal features inspired by TSQL2.

A collection of *must have* references are available on the site of Richard Snodgrass at the University of Arizona (*https://www2.cs.arizona.edu/~rts/publications.html*).

– Jensen C. S. and Snodgrass R. T. *Temporal Data Management*, IEEE Trans. on Knowledge and Data Engineering, Vol. 11, No 1, Jan. 1999, 9 pages.
  A short survey of temporal database concepts and design. Free pdf version available.

– Snodgrass, R., T. *Developing Time-Oriented Database Applications in SQL*, 2000, Morgan-Kaufmann, 528 pages.
  An in-depth analysis of the most essential concepts and algorithms in temporal database. Free pdf version available. More on this reference in the second part.

– Jensen C. S. and Snodgrass R. T. (editors). *Temporal Database Entries for the Springer Encyclopedia of Database Systems*. 345 pages.
  The collection of the entries of the *Springer Encyclopedia of Database Systems*. related to temporal databases. Free pdf version available.

– See also webpage http://www.cs.arizona.edu/people/rts/sql3.html, entitled *TSQL2 and SQL3 Interactions*, examine how the proposals of temporal TSQL2 language are being integrated in SQL standards and in some popular RDBMS.

A different model of temporal databases has been proposed by H. Darwen:

– Date C. J., Darwen H. and Lorentzos, N. L. *Time and Relational Theory: Temporal Databases in the Relational Model and SQL*. Morgan-Kaufmann, 2014. 558 pages.

Case study **10**

Temporal databases - Part 2

Objective. In this second part of the study of temporal data, we examine the various ways to query and transform them. We first examine simple temporal and non temporal queries, providing themselves temporal and non temporal results. Then, we extend to the temporal dimension the main families of queries of standard, non temporal, SQL: projection (entity-based and generalized), inner join and outer join, aggregation (count, max, min, average, sum). We also describe the SQLfast temporal library LTemp that offers a series of operators intended to write concise and efficient temporal scripts. The various temporal data models described in part 1 are revisited in order to develop conversion algorithms from one model to each of the other ones. Finally, we address the problem of performance by comparing the various algorithms of the temporal operators. To get realistic execution time measures we apply these algorithms to larger temporal databases. The last section is devoted to a short description of the SQL:2011 standard, that introduces some (but not all) concepts of temporal databases.

Keywords. temporal relations, temporal query, temporal projection, coalescing, temporal inner join, temporal outer join, temporal aggregation, stable interval, temporal data model conversion, temporal operator performance, SQL:2011, LTemp library.

10.1 Introduction

A temporal database can provide a response to a wide variety of time-related queries. It is easy to understand that extracting, or consulting, data from a set of historical tables is more complex than similar operations on traditional databases, limited to current states of the application domain. This feeling is both true and false. Some temporal queries will have a simple and intuitive formulation in SQL, while others will require far more complex SQL translation than their expression in plain English.

10.2 Temporal relations

Temporal queries will often be based on definite relationships between two instants, between an instant and an interval or between two intervals. We will briefly describe these relationships, which specify the relative positions of their arguments. They all derive from Allen's interval algebra which comprises 13 base relations[1]. The three tables below classify the temporal relations and assign them their SQL interpretation. They postulate that instants and intervals share the same time granularity.

Relations between instants **t1** and **t2**		
	relation	**SQL expression**
r1	t1 and t2 simultaneous	t1 = t2
r2	t1 and t2 distinct	t1 <> t2
r3	t1 before t2	t1 < t2
r3	t1 after t2	t1 > t2
r5	t1 not before t2	t1 >= t2
r6	t1 not after t2	t1 <= t2

Relations between instant **t** and temporal interval **I** ≡ [**s**,**e**)		
	relation	**SQL expression**
r7	t before I	`t < s`
r8	t after I	`t >= e`
r9	I starts with t	`t = e`
r10	I ends with t	`t = e - 1`
r11	I includes t	`s <= t and t < e`
r12	I starts before t	`s < t`
r13	I ends after t	`e - 1 > t`

1. https://en.wikipedia.org/wiki/Allen's_interval_algebra

Relations between intervals **I1** ≡ [**s1**,**e1**) and **I2** ≡ [**s2**,**e2**)		
	relation	**SQL expression**
r14	I1 before I2	`e1 <= s2`
r15	I2 follows I1 (aka *I1 meets I2*)	`s1 = e2`
r16	I1 starts I2	`s1 = s2`
r17	I1 ends I2	`e1 = e2`
r18	I1 during I2	`(s1 >= s2) and (e1 <= e2)`
r19	I1 equals to I2	`(s1 = s2) and (e1 = e2)`
r20	I1 and I2 disjoint (*implied by r14 and r15*)	`(e1 <= s2 or (s1 >= e2)`
r21	I1 and I2 overlap (*implied by r16 to r19*)	`(e1 > s2) and (s1 < e2)` `max(s1,s2) < min(e1,e2)`

Intersection of intervals

Let us consider intervals **I1** ≡ [**s1**,**e1**) and **I2** ≡ [**s2**,**e2**). If they overlap, that is, if e1 > s2 and s1 < e2 (or if max(s1,s2) < min(e1,e2)), their intersection can be computed as:

```
intersect(I1,I2) ≡ [max(s1,s2),min(e1,e2))
```

This expression can be generalized to more than two intervals:

```
intersect(I1,..,In) ≡ [max(s1,..,sn),min(e1,..,en))
```

Note

It may come as a surprise that we do not suggest in this study to develop a user defined temporal function that checks whether two intervals overlap, something like

```
overlap(s1,e1,s2,e2)
```

Such a function should simplify many queries and make them easier to write and to understand, and this, especially since several DBMS (SQLite excluded, unfortunately) already include such a function in their core library (the overlaps() function of SQL:2011 for example). The answer is related to execution performance of queries. Most of them can be considerably accelerated thanks to indexes defined on temporal columns start and end. If these columns appear explicitly in SQL conditions, the query optimizer is able to take these indexes into account when computing the best execution plan.

So, in the limited context of this case study, the answer is twofold: nice idea for small temporal databases but it will prevent the DBMS to optimize many queries operating on large, complex databases.

Anyway, function `toverlap` [2] is available in the SQLfast library, but this does not invalidate the above reasoning. Therefore, to use with caution!

10.3 The example temporal databases

We will develop and discuss the different classes of queries on the basic temporal model according to which all the states, past and current, are stored in the same tables (*entity-based*, or more generally, *tuple-based* model). Here below, we recall the structure and contents of historical tables H_EMPLOYEE, H_PROJECT and the views showing the current states of employees and projects.

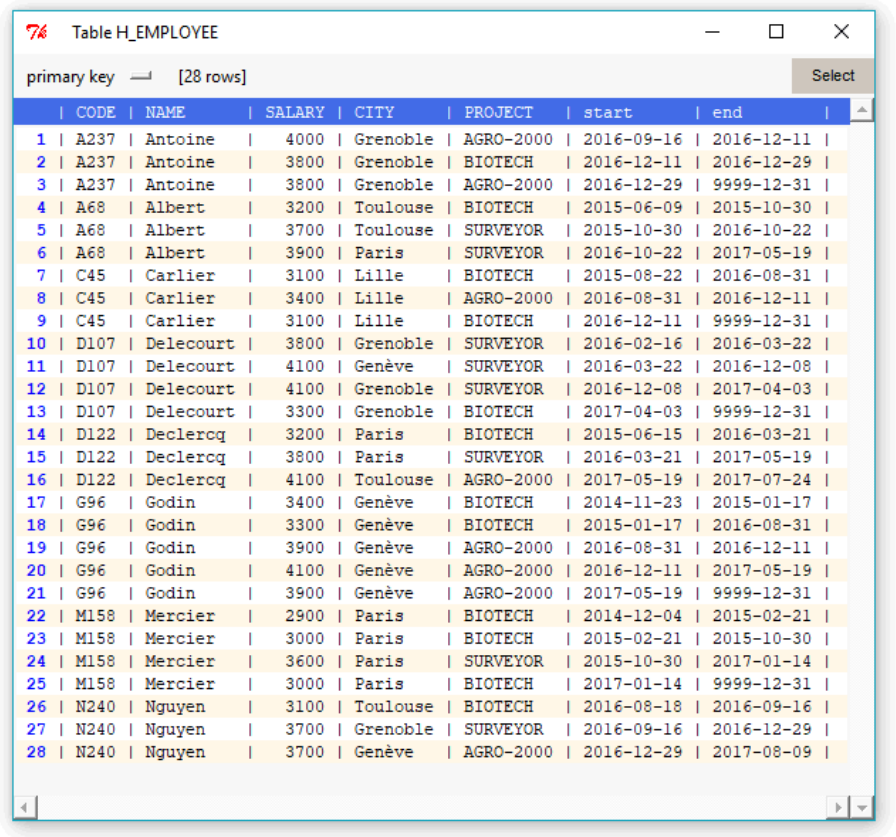


Table H_EMPLOYEE

primary key [28 rows] Select

	CODE	NAME	SALARY	CITY	PROJECT	start	end
1	A237	Antoine	4000	Grenoble	AGRO-2000	2016-09-16	2016-12-11
2	A237	Antoine	3800	Grenoble	BIOTECH	2016-12-11	2016-12-29
3	A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31
4	A68	Albert	3200	Toulouse	BIOTECH	2015-06-09	2015-10-30
5	A68	Albert	3700	Toulouse	SURVEYOR	2015-10-30	2016-10-22
6	A68	Albert	3900	Paris	SURVEYOR	2016-10-22	2017-05-19
7	C45	Carlier	3100	Lille	BIOTECH	2015-08-22	2016-08-31
8	C45	Carlier	3400	Lille	AGRO-2000	2016-08-31	2016-12-11
9	C45	Carlier	3100	Lille	BIOTECH	2016-12-11	9999-12-31
10	D107	Delecourt	3800	Grenoble	SURVEYOR	2016-02-16	2016-03-22
11	D107	Delecourt	4100	Genève	SURVEYOR	2016-03-22	2016-12-08
12	D107	Delecourt	4100	Grenoble	SURVEYOR	2016-12-08	2017-04-03
13	D107	Delecourt	3300	Grenoble	BIOTECH	2017-04-03	9999-12-31
14	D122	Declercq	3200	Paris	BIOTECH	2015-06-15	2016-03-21
15	D122	Declercq	3800	Paris	SURVEYOR	2016-03-21	2017-05-19
16	D122	Declercq	4100	Toulouse	AGRO-2000	2017-05-19	2017-07-24
17	G96	Godin	3400	Genève	BIOTECH	2014-11-23	2015-01-17
18	G96	Godin	3300	Genève	BIOTECH	2015-01-17	2016-08-31
19	G96	Godin	3900	Genève	AGRO-2000	2016-08-31	2016-12-11
20	G96	Godin	4100	Genève	AGRO-2000	2016-12-11	2017-05-19
21	G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31
22	M158	Mercier	2900	Paris	BIOTECH	2014-12-04	2015-02-21
23	M158	Mercier	3000	Paris	BIOTECH	2015-02-21	2015-10-30
24	M158	Mercier	3600	Paris	SURVEYOR	2015-10-30	2017-01-14
25	M158	Mercier	3000	Paris	BIOTECH	2017-01-14	9999-12-31
26	N240	Nguyen	3100	Toulouse	BIOTECH	2016-08-18	2016-09-16
27	N240	Nguyen	3700	Grenoble	SURVEYOR	2016-09-16	2016-12-29
28	N240	Nguyen	3700	Genève	AGRO-2000	2016-12-29	2017-08-09

2. `toverlap`, for *temporal overlap*, applicable to closed-open intervals.

Table H_PROJECT

primary key [11 rows] Select

	TITLE	THEME	BUDGET	start	end
1	AGRO-2000	Crop improvement	65000	2016-08-31	2017-04-14
2	AGRO-2000	Crop improvement	75000	2017-04-14	2017-07-05
3	AGRO-2000	Crop improvement	82000	2017-07-05	9999-12-31
4	BIOTECH	Biotechnology	180000	2014-11-18	2015-02-27
5	BIOTECH	Genetic engineering	160000	2015-02-27	2015-06-15
6	BIOTECH	Genetic engineering	120000	2015-06-15	2016-08-23
7	BIOTECH	Genetic engineering	140000	2016-08-23	2017-09-17
8	BIOTECH	Biotechnology	140000	2017-09-17	9999-12-31
9	SURVEYOR	Satellite monitoring	310000	2015-10-30	2016-06-25
10	SURVEYOR	Satellite monitoring	375000	2016-06-25	2016-12-20
11	SURVEYOR	Satellite monitoring	345000	2016-12-20	2017-05-19

View EMPLOYEE

CODE [5 rows] Select

	CODE	NAME	SALARY	CITY	PROJECT	start	end
1	A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31
2	C45	Carlier	3100	Lille	BIOTECH	2016-12-11	9999-12-31
3	D107	Delecourt	3300	Grenoble	BIOTECH	2017-04-03	9999-12-31
4	G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31
5	M158	Mercier	3000	Paris	BIOTECH	2017-01-14	9999-12-31

View PROJECT

TITLE [2 rows] Select

	TITLE	THEME	BUDGET	start	end
1	AGRO-2000	Crop improvement	82000	2017-07-05	9999-12-31
2	BIOTECH	Biotechnology	140000	2017-09-17	9999-12-31

This database is fine to develop temporal queries and scripts, to analyze their behavior and to observe their effect, but it is far too small to be useful when we evaluate and compare their execution performance. To this goal, we will use larger experimental data sets comprising the history of 100 to 200,000 employees.

10.4 Non temporal queries

A history table also shows the current states of the entities and can therefore answer the classic queries specific to non-temporal databases. Expressed on the views of the current states, these queries are identical to those of the equivalent non-temporal databases.

- What is the current state of employee M158?

```
select CODE,NAME,SALARY,CITY,PROJECT
from   EMPLOYEE
where  CODE = 'M158';
```

Script 10.1 - Current state of employee M158

```
+------+---------+--------+-------+---------+
| CODE | NAME    | SALARY | CITY  | PROJECT |
+------+---------+--------+-------+---------+
| M158 | Mercier | 3000   | Paris | BIOTECH |
+------+---------+--------+-------+---------+
```

- Same query against the historical table.

```
select CODE,NAME,SALARY,CITY,PROJECT
from   H_EMPLOYEE
where  CODE = 'M158' and end = '9999-12-31';
```

Script 10.2 - Current state of employee M158 (from the historical table).

- Show the project(s) with the highest budget.

```
select distinct TITLE,BUDGET
from   PROJECT
where  BUDGET = (select max(BUDGET) from PROJECT);
```

Script 10.3 - Project(s) with the highest budget.

```
+---------+--------+
| TITLE   | BUDGET |
+---------+--------+
| BIOTECH | 140000 |
+---------+--------+
```

10.5 Temporal queries with temporal result

A temporal query consults past and current entity states. Its result may or may not be temporal. The state of an entity, or of a set of entities, at a definite time point generally is called a **snapshot**.

We will first consider simple queries that produce temporal data extracted from a single table. In these queries, variable Tfuture denotes infinite future, typically 9999-12-31.

- Show the history of employees between 2016-01-01 and 2016-12-31.

  The interesting aspect of this query is the *clipping* of the intervals of the first and last state of each employee to adjust them to the interval of the query.

```
select CODE,NAME,SALARY,CITY,PROJECT
       max(start,'2016-01-01') as "Start",
       min(end,'2016-12-31') as "End",
from   H_EMPLOYEE
where  start < '2016-12-31' and '2016-01-01' < end
order by NAME,"Start","End";
```

Script 10.4 - History of employees between 2016-01-01 and 2016-12-31

CODE	NAME	SALARY	CITY	PROJECT	Start	End
A68	Albert	3700	Toulouse	SURVEYOR	2016-01-01	2016-10-22
A68	Albert	3900	Paris	SURVEYOR	2016-10-22	2016-12-31
...	...	...	...	...	...	...
N240	Nguyen	3700	Grenoble	SURVEYOR	2016-09-16	2016-12-29
N240	Nguyen	3700	Genève	AGRO-2000	2016-12-29	2016-12-31

- Show the life of the projects

  To make data more readable, infinite future is displayed as *null* value.

```
select TITLE,min(start) as Started,
       case when max(end) = '$Tfuture$'
            then null
            else max(end) end as Closed
from   H_PROJECT
group by TITLE
```

Script 10.5 - Life of the projects

TITLE	Started	Closed
AGRO-2000	2016-08-31	--
BIOTECH	2014-11-18	--
SURVEYOR	2015-10-30	2017-05-19

- Show the start date, end date and life span (duration or age) of each closed project

  This query makes use of temporal function durationDays(d1,d2), which returns the number of days between time points d1 and d2 (typically dates). If the arguments denote time or datetime values, the result is rounded to the closest integer.

```
select TITLE, min(start) as "Start", max(end) as "End",
       durationDays(min(start),max(end)) as Age
from   H_PROJECT
group by TITLE
having max(end) < '$Tfuture$';
```

Script 10.6 - Start date, end date and duration of each closed project

```
+----------+------------+------------+-----+
| TITLE    | Start      | End        | Age |
+----------+------------+------------+-----+
| SURVEYOR | 2015-10-30 | 2017-05-19 | 567 |
+----------+------------+------------+-----+
```

- We observe that several expressions appear more than once, which may make the query fairly obscure and its execution potentially inefficient (depending on the optimizer). We suggest to rewrite the query with a CTE.

```
with MinMax(TITLE,MinStart,MaxEnd)
as (select TITLE, min(start) as MinStart, max(end) as MaxEnd
    from   H_PROJECT
    group by TITLE
    having MaxEnd < '$Tfuture$')
select TITLE,MinStart as "Start",MaxEnd as "End",
       durationDays(MinStart,MaxEnd) as Duration
from   MinMax;
```

Script 10.7 - Same with a CTE

- Display the start date and current age of each active project

  The *age* of an active project is computed from its start date to the current date (given by system variable date). This is a typical example of non deterministic query: run it tomorrow and it will give another result, though the database has not been updated!

```
with StartAge(TITLE,MinStart)
as (select TITLE, min(start) as MinStart
    from   H_PROJECT
    group by TITLE
    having max(end) = '$Tfuture$')
select TITLE,MinStart as "Start",
       durationDays(MinStart,'$date$') as Age
from   StartAge;
```

Script 10.8 - Start date and current age of each active project (through a CTE)

TITLE	Start	Age
AGRO-2000	2016-08-31	917
BIOTECH	2014-11-18	1569

- When did the budget of each project exceed 150,000?

```
select TITLE,BUDGET,start,end
from   H_PROJECT
where  BUDGET > 150000;
```

Script 10.9 - When did the budget of each project exceed 150,000?

Warning. Though the result shown below looks good, this query can produce a non-standardized history, as states of some projects may not be consecutive and some consecutive states may by identical (the exact term is *value-equivalent*). Actually, the operation requested is a *projection*. More on this later on. Same remark on the next query.

TITLE	BUDGET	start	end
BIOTECH	180000	2014-11-18	2015-02-27
BIOTECH	160000	2015-02-27	2015-06-15
SURVEYOR	310000	2015-10-30	2016-06-25
SURVEYOR	375000	2016-06-25	2016-12-20
SURVEYOR	345000	2016-12-20	2017-05-19

- What was (and when) the maximum budget of each project?

```
select TITLE,BUDGET,start,end
from   H_PROJECT P
where  BUDGET = (select max(BUDGET)
                 from   H_PROJECT
                 where  TITLE = P.TITLE);
```

Script 10.10 - What was (and in what periods) the maximum budget of each project?

TITLE	BUDGET	start	end
BIOTECH	180000	2014-11-18	2015-02-27
SURVEYOR	375000	2016-06-25	2016-12-20
AGRO-2000	82000	2017-07-05	9999-12-31

10.6 Temporal queries with non temporal result

These queries explore time dependent data (H_* tables) but absorbs the time dimension, either by reducing the search space to a specific time point or through aggregation (statistical) functions ***on the time boundaries***.

- Show the state of employee D107 on 2017-02-01 (*snapshot*)

```
select CODE,NAME,SALARY,CITY,PROJECT
from   H_EMPLOYEE
where  CODE = 'D107'
and    start <= '2017-02-01' and '2017-02-01' < end;
```

Script 10.11 - Snapshot of employee D107 on 2017-02-01

CODE	NAME	SALARY	CITY	PROJECT
D107	Delecourt	4100	Grenoble	SURVEYOR

- How many employees were assigned to the SURVEYOR project on 2017-02-01?

```
select count(distinct CODE) as Staff
from   H_EMPLOYEE
where  PROJECT = 'SURVEYOR'
and    start <= '2017-02-01' and '2017-02-01' < end;
```

Script 10.12 - How many employees were assigned to the SURVEYOR project on 2017-02-01?

```
+-------+
| Staff |
+-------+
| 3     |
+-------+
```

- How many projects has employee N240 been assigned to throughout her career?

```
select count(distinct PROJECT) as Projects
from   H_EMPLOYEE
where  CODE = 'N240';
```

Script 10.13 - How many projects has the N240 employee been assigned to throughout her career?

```
+----------+
| Projects |
+----------+
| 3        |
+----------+
```

- How many projects was employee M158 assigned to while living in Paris?

```
select count(distinct PROJECT) as Projects
from   H_EMPLOYEE
where  CODE = 'M158' and CITY = 'Paris';
```

Script 10.14 - How many projects was employee M158 assigned to while living in Paris?

```
+----------+
| Projects |
+----------+
| 2        |
+----------+
```

- Which project(s) has employee N240 been assigned to the longest?

  Note: SQLite does not allow subquery quantifiers *any* or *all*. In this expression, the subquery selects the highest value within the durations of employee N240 in her projects.

  For current states, when the employee still is a member of the staff of the project, the *end* value in the computation of the duration is replaced by the current date. As already noticed, this query is non deterministic.

```
select PROJECT,
       sum(durationDays(start,min(end,'$date$'))) as Duration
from   H_EMPLOYEE
where  CODE = 'N240'
group by PROJECT
having Duration =
     (select sum(durationDays(start,min(end,'$date$'))) as Dur
      from   H_EMPLOYEE
      where  CODE = 'N240'
      group by PROJECT
      order by Dur desc limit 1);
```

Script 10.15 - Which project(s) has employee N240 been assigned to the longest?

```
+---------+----------+
| PROJECT | Duration |
+---------+----------+
| BIOTECH | 703      |
+---------+----------+
```

10.7 Complex temporal queries

The queries developed so far are fairly mundane, once the bases of SQL have been mastered. Unfortunately, and surprisingly, other seemingly harmless queries will require much more complex SQL expressions. We will examine and resolve three conventional operators that require the greatest care when applied to temporal data, namely,

1. temporal projection (Section 10.8),
2. temporal join (Section 10.9) and outer join (Section 10.10).
3. temporal aggregation (Section 10.11).

Thanks to these operators, we will be able to resolve, sometimes in several steps, most of the questions that are addressed to a temporal database, and which are currently beyond our reach.

10.8 Temporal projection

The *projection* is the simplest operator that can be applied to non temporal data. It consists in restricting the rows of a selection to certain columns of the initial table. Considering the PROJECT table, its projection on columns TITLE and THEME, therefore discarding BUDGET values, is expressed in SQL as follows:

```
select TITLE,THEME
from   PROJECT;
```

Script 10.16 - SQL expression of the projection of PROJECT on columns TITLE and THEME

If the columns of the *select list* do not include all the components of a unique (or primary) key, then it is recommended to remove duplicates through the `distinct` modifier:

```
select distinct THEME
from   PROJET
```

Script 10.17 - SQL expression of a projection that produces a set of unique values

10.8.1 Entity-based temporal projection

We first study a simple variant of projection, in which the target columns include all the components of a unique key. We will call it *entity-based projection*.

Let us apply the pattern of Script 10.16 to table H_PROJECT, from which we want to display not only the couples of (TITLE,THEME) values of each project but also the intervals in which each couple was valid. Let us try this query:

```
select TITLE,THEME,start,end
from   H_PROJECT;
```

Script 10.18 - First attempt to compute the projection of table H_PROJECT on columns TITLE and THEME

With this result:

TITLE	THEME	start	end
AGRO-2000	Crop improvement	2016-08-31	2017-04-14
AGRO-2000	Crop improvement	2017-04-14	2017-07-05
AGRO-2000	Crop improvement	2017-07-05	9999-12-31
BIOTECH	Biotechnology	2014-11-18	2015-02-27
BIOTECH	Genetic engineering	2015-02-27	2015-06-15
BIOTECH	Genetic engineering	2015-06-15	2016-08-23
BIOTECH	Genetic engineering	2016-08-23	2017-09-17
BIOTECH	Biotechnology	2017-09-17	9999-12-31
SURVEYOR	Satellite monitoring	2015-10-30	2016-06-25
SURVEYOR	Satellite monitoring	2016-06-25	2016-12-20
SURVEYOR	Satellite monitoring	2016-12-20	2017-05-19

Figure 10.1 - Projecting H_PROJECT on TITLE, THEME - First (failing) trial

Though these rows are not incorrect, anyway they represent true facts, the way they display these facts is not what we expected. Let us add the clause `order by start` to this query, and the result, though equivalent, becomes unreadable. The problem is that these rows include several consecutive identical states, generally called *value-equivalent states*. For instance, the first three rows of this result should be merged into this single row:

```
| AGRO-2000 | Crop improvement     | 2016-08-31 | 9999-12-31 |
```

In order to merge these states, let us try this query, that only keep the lowest and highest time limits of consecutive value-equivalent states.

```
insert into H_THEME_2
select TITLE,THEME,min(start) as "Start",max(end) as "End"
from   H_PROJECT
group by TITLE,THEME;
```

Script 10.19 - Second (still failing) attempt to compute the projection of table H_PROJECT on columns TITLE and THEME

```
+-----------+----------------------+------------+------------+
| TITLE     | THEME                | Start      | End        |
+-----------+----------------------+------------+------------+
| AGRO-2000 | Crop improvement     | 2016-08-31 | 9999-12-31 |
| BIOTECH   | Biotechnology        | 2014-11-18 | 9999-12-31 |
| BIOTECH   | Genetic engineering  | 2015-02-27 | 2017-09-17 |
| SURVEYOR  | Satellite monitoring | 2015-10-30 | 2017-05-19 |
+-----------+----------------------+------------+------------+
```

Figure 10.2 - Projecting H_PROJECT on TITLE, THEME - Second (failing) trial

At first glance, the result looks much better, until we observe a curious phenomenon: one of the two states of project BIOTECH is embedded within the other one! So, these rows tell that, from 2015-02-27 to 2017-09-17, this project was in two distinct states, with two different values of THEME, which is absurd.

Obviously, the correct answer should be the following:

```
+-----------+----------------------+------------+------------+
| TITLE     | THEME                | start      | end        |
+-----------+----------------------+------------+------------+
| AGRO-2000 | Crop improvement     | 2016-08-31 | 9999-12-31 |
| BIOTECH   | Biotechnology        | 2014-11-18 | 2015-02-27 |
| BIOTECH   | Genetic engineering  | 2015-02-27 | 2017-09-17 |
| BIOTECH   | Biotechnology        | 2017-09-17 | 9999-12-31 |
| SURVEYOR  | Satellite monitoring | 2015-10-30 | 2017-05-19 |
+-----------+----------------------+------------+------------+
```

Figure 10.3 - Projecting H_PROJECT on TITLE, THEME - The correct expected result

When we compare the rows of Figures 10.1 and 10.3, we immediately see that each row of the correct result is formed by merging, or reducing[3], a series of consecutive value-equivalent rows in Figure 10.1. We call this series a *maximal suite of identical consecutive states* or MSICS.

In temporal table R, an MSICS comprises *one or several* consecutive states (s1, s2, ..., sn) such that:

- all these states are value-equivalent
- either s1 is the first state of R or its previous state is not value-equivalent; in short, the first state does not follow a value-equivalent state
- either sn is the last state of R or its next state is not value-equivalent; in short, the last state is not followed by a value-equivalent state.

Practically, if **R** represents project states, an MSICS **P** is defined by two project states (**P1**, **Pn**), such that:

- **P1** precedes **Pn** (unless the MSICS comprises one state only),
- **P1** and **Pn** are value-equivalent; let us call their value **v**,
- there is no state **P0** with value **v** that directly precedes **P1**,
- there is no state **Pm** with value **v** that directly follows **Pn**,
- all the states between **P1** and **Pn** have the same value **v**; or, reversing the condition, no state between **P1** and **Pn** has a value different from **v**.

These conditions are depicted in Figure 10.4.

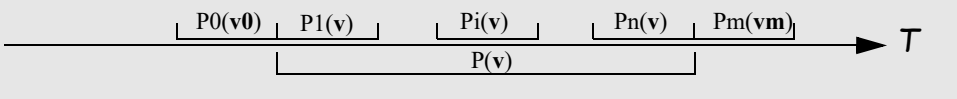


Figure 10.4 - Definition of MSICS P comprising states (P1,..., Pn)

The algorithm reduces each MSICS to the unique state (v,s1,en). We express it through the SQL query of Script 10.20

The first two conditions on subquery could be merged into this one:

```
and not exists (select * from H_PROJECT P0
                where  (P0.TITLE,P0.THEME)=(P1.TITLE,P1.THEME)
                and    (P0.end = P1.start
                        or Pn.end = P0.start))
```

3. The technical term for this reduction operation is ***coalescing***. Since SQL already includes a function called coalesce, we will keep the name ***reduction***.

```
select P1.TITLE, P1.THEME, P1.start, Pn.end
from   H_PROJECT P1, H_PROJECT Pn
where  (P1.TITLE,P1.THEME)=(Pn.TITLE,Pn.THEME)
and    P1.start <= Pn.start
and not exists (select * from H_PROJECT P0
                where  (P0.TITLE,P0.THEME)=(P1.TITLE,P1.THEME)
                and    P0.end = P1.start)
and not exists (select * from H_PROJECT Pm
                where  (Pm.TITLE,Pm.THEME)=(P1.TITLE,P1.THEME)
                and    Pn.end = Pm.start)
and not exists (select * from H_PROJECT Pi
                where  Pi.TITLE = P1.TITLE
                and    Pi.THEME <> P1.THEME
                and    P1.end <= Pi.start
                and    Pi.end <= Pn.start);
```

Script 10.20 - Projecting temporal table H_PROJECT on columns (TITLE,THEME)

However, performance tests show that this concise form is *five times slower* than the former one (observed with SQLite 3.28). So, we abandon the idea.

This algorithm works fine if the following strict conditions are met:

– The temporal table records the history of a set of *similar entities*, e.g., *employees*, *projects*, etc. These entities are identified by an entity primary key, e.g., TITLE for projects in H_PROJECT or CODE for employees in H_EMPLOYEE.

– The history of each entity is *normalized*, that is, it includes no *gaps*, no *overlapping states*, even though the latter are value-equivalent and no *consecutive value-equivalent states*.

– The projection columns include the primary key of the entities, e.g., TITLE, THEME for H_PROJECT or CODE, CITY, PROJECT for H_EMPLOYEE.

Otherwise, it fails to produce a correct result. We will cope with this problem in Section 10.8.3, but, before, we will study ways to represent missing information.

10.8.2 Temporal projection with null values

Let us suppose that we do not always know the city in which an employee has been living. The usual way to translate this (absence of) knowledge consists in setting column CITY to *null* (CITY must be declared nullable). Figure 10.5 shows that the city of employee A68 was unknown during interval [2015-10-30, 2016-10-22).

In the projection of H_EMPLOYEE on CODE, CITY, how do we represent the missing information on employee A68? Do we generate a row telling that the value of CITY is *null* during this interval, or do we merely discard such row? Figure 10.6 shows the state of the projection according to these alternatives.

CODE	NAME	SALARY	CITY	PROJECT	start	end
...	...	...	...	...	...	...
A68	Albert	3200	Toulouse	BIOTECH	2015-06-09	2015-10-30
A68	Albert	3700	--	SURVEYOR	2015-10-30	2016-10-22
A68	Albert	3900	Paris	SURVEYOR	2016-10-22	2017-05-19
...	...	...	...	...	...	...

Figure 10.5 - Information may be unknown for some period

CODE	CITY	start	end
...	...	...	...
A68	Toulouse	2015-06-09	2015-10-30
A68	--	2015-10-30	2016-10-22
A68	Paris	2016-10-22	2017-05-19
...	...	...	...

CODE	CITY	start	end
...	...	...	...
A68	Toulouse	2015-06-09	2015-10-30
A68	Paris	2016-10-22	2017-05-19
...	...	...	...

Figure 10.6 - To ways to represent missing information in a projection

The first representation preserves *null* values, known to induce complexity in data processing. The second representation is more concise and more natural: the best way to represent the absence of information is ... the absence of representation!

On the other hand, rebuilding source table H_EMPLOYEE will use different operators, namely *temporal inner join* (Section 10.9) with the first representation and *temporal outer join* (Section 10.10) in the second one.

The algorithm of Script 10.20 generates projections in the first representation. Discarding *null* states is quite easy (Script 10.21).[4]

```
delete from H_CITY where CITY is null;
```

Script 10.21 - Translating the first representation into the second one

10.8.3 Generalized temporal projection

Let us consider the projection of H_EMPLOYEE on column CITY, which is clearly not a unique key for employees. The result, shown below, should tell us in *which cities* and *when* at least one employee worked (or still is working) on a project:

4. More generally: `delete from H_T where (col1,.., coln) is (null,.., null)`

CITY	start	end
Genève	2016-03-22	2016-12-08
Genève	2016-12-29	2017-08-09
Genève	2016-12-29	9999-12-31
Grenoble	2016-02-16	2016-03-22
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-09-16	9999-12-31
Grenoble	2016-12-08	9999-12-31
Grenoble	2016-12-08	9999-12-31
Paris	2015-06-15	2017-05-19
Paris	2016-10-22	2017-05-19
Paris	2016-10-22	9999-12-31
Toulouse	2015-06-09	2016-10-22
Toulouse	2016-08-18	2016-09-16
Toulouse	2017-05-19	2017-07-24

What do we observe?

- Many states overlap, which makes the data difficult to interpret.
- This history shows *gaps* (Genève was *inactive* between 2016-12-08 and 2016-12-29). Though such gaps do not necessarily represent anomalies, they are a new phenomenon.
- Worse, states are missing:. For instance, employee G96 (Godin) worked on project BIOTECH from 2014-11-23 to 2016-08-31 while living in Genève. No trace of this fact in the result.

Actually, the result we expect is the following:

CITY	start	end
Genève	2014-11-23	9999-12-31
Grenoble	2016-02-16	2016-03-22
Grenoble	2016-09-16	9999-12-31
Lille	2015-08-22	9999-12-31
Paris	2014-12-04	9999-12-31
Toulouse	2015-06-09	2016-10-22
Toulouse	2017-05-19	2017-07-24

So, we conclude that the algorithm we just developed fails. Obviously, we must build a more general algorithm that solves all the projection patterns. In particular, it must apply to temporal tables which include:

– overlapping value-equivalent states,

– consecutive value-equivalent states,

– gaps.

In addition, it must tolerate incorrect data, according to which an entity is (erroneously) reported to be in more than one state. It must preserve them, leaving their processing to another process.

Before developing this improved algorithm, we define the concept of *interval extension*. Interval *A extends interval B* if their union has no gap and is larger than B. More precisely,

- interval A *extends* interval B *to the left* (Figure 10.7, left) either if B follows A (eA = sB) or if A starts before B and A and B overlap (sA < sB and eA > sB); in short, if sA < sB and eA >= sB.
- interval A' *extends* interval B *to the right* (Figure 10.7, right), either if A' follows B (sA' = eB) or if B starts before A' and A' and B overlap (sA' < eB and eA' > eB); in short, if sA' <= eB and eA' > eB.

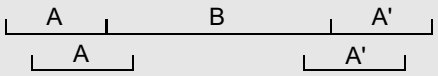


Figure 10.7 - Two ways for A (or A') to extend B

The key concept of *maximum suite of identical consecutive states* (MSICS) must be extended to cope with overlapping states. So, we will consider reducing each *maximal suite of identical consecutive* ***or overlapping*** *states* (or MSICOS). The suite (S1, ..., Sn) is a MSICOS that must be reduced to (v,s1,en) if and only if:

1. S1 and Sn have the same value v
2. either S1 = Sn or S1 starts before Sn and does not ends after Sn (s1 <= sn and e1 <= en)
3. there is no state S0 with value v that would extend S1 to the left
4. there is no state Sm with value v that would extend Sn to the right
5. there is no state of the MSICOS between S1 and Sn with a value different from v
6. there is no missing state (gap) in the MSICOS between S1 and Sn.

Conditions 1 to 5 are fairly easy to express in SQL: they are simple extensions of those of Script 10.20. However, condition 6 is more complicated since it describes the *absence of non-existent objects*.

The algorithm we will develop is based on the idea described in [Böhlen,1996][5]. Let us consider state Sg with value v, that ends after e1 and before sn, that is, e1 <= eg < sn. If Sg is extended to the right by any state Sk with value v, then Sg *is not followed by a gap*. Hence the rule that implements condition 6:

5. Böhlen, M., R., Snodgrass, R., T., Soo, M., D., Coalescing in Temporal Databases, in *Proc. VLDB Conf.*, Sept. 1996

if there exists a state Sg with value v such that e1 <= eg < sn, this state must be extended to the right by a state Sk with value v

or, reversing this rule, which is easier to translate in SQL:

there is no state Sg with value v such that e1 <= eg < sn, that is not extended to the right by a state Sk with value v

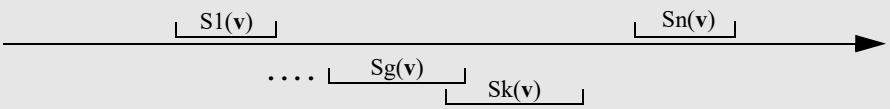


Figure 10.8 - Checking whether a gap exists between S1 and Sn

The algorithm that implements conditions 1 to 6 is translated into the query of Script 10.22.

```
select distinct E1.CITY, E1.start, En.end
from   H_EMPLOYEE E1, H_EMPLOYEE En
where  E1.CITY = En.CITY
and    E1.start <= En.start and E1.end <= En.end
and not exists (select *
                from   H_EMPLOYEE E0
                where  E0.CITY = E1.CITY
                and    E0.start <  E1.start
                and    E0.end   >= E1.start)
and not exists (select *
                from   H_EMPLOYEE Em
                where  Em.CITY = E1.CITY
                and    Em.start <= En.end
                and    Em.end   >  En.end)
and not exists (select *
                from   H_EMPLOYEE Eg
                where  Eg.CITY = E1.CITY
                and    E1.end <= Eg.end
                and    Eg.end <  En.start
                and    not exists(select *
                                  from   H_EMPLOYEE Ek
                                  where  Ek.CITY = E1.CITY
                                  and    Eg.end   <  Ek.end
                                  and    Ek.start <= Eg.end));
```

Script 10.22 - Projecting temporal table H_EMPLOYEE on column CITY

Building this query is a nice intellectual exercise, but the result is not particularly easy to understand. In addition, executing this query on large temporal tables will probably be quite expensive (we will check this later).

Fortunately, we can develop a procedural algorithm that is much simpler and faster. Let us consider this elementary query

```
select CITY,start,end
from   H_EMPLOYEE
order by CITY,start;
```

that produces the following result:

```
+----------+------------+------------+
| CITY     | start      | end        |
+----------+------------+------------+
| ...      | ...        | ...        |
| Genève   | ...        | ...        |
| Grenoble | 2016-02-16 | 2016-03-22 |  (a)
| Grenoble | 2016-09-16 | 2016-12-11 |  (b)
| Grenoble | 2016-09-16 | 2016-12-29 |  (c)
| Grenoble | 2016-12-08 | 2017-04-03 |  (d)
| Grenoble | 2016-12-11 | 2016-12-29 |  (e)
| Grenoble | 2016-12-29 | 9999-12-31 |  (f)
| Grenoble | 2017-04-03 | 9999-12-31 |  (g)
| Lille    | ...        | ...        |  (h)
| ...      | ...        | ...        |
+----------+------------+------------+
```

If we read these rows sequentially, we get the rows of each city in chronological order, that is, in increasing (actually *not decreasing*) values of the start column. Let us focus on city Grenoble, for which we read the successive rows. We reason as follows:

- When we read row (a), we identify the first state of a MSICOS and we memorize it as its tentative reduction [Grenoble, 2016-02-16, 2016-03-22).
- We read row (b). We observe that its start value (2016-09-16) is greater than the end value of the current reduction (2016-03-22). This means that we encounter a gap that closes the current MSICOS. So, (Grenoble, 2016-02-16, 2016-03-22) is the (trivial) reduction of the current MSICOS and belongs to the projection we are computing. Row (b) becomes the first row of the next MSICOS. We initialize the new current reduction to (Grenoble, 2016-09-16, 2016-12-11).
- We read row (c). Since its start value (2016-09-16) is lower than the end value of the current reduction (2016-12-11), this row extends the latter. So, we modify the current reduction to (Grenoble, 2016-09-16, 2016-12-29). So far, rows (b) and (c) have been reduced.
- Successively reading rows (d), (e), (f) and (g) leads us to the same observation, each one extending the current reduction. At this point, the current MSICOS is formed with rows (b) to (g) and its current reduction is (Grenoble, 2016-09-16, 9999-12-31).
- Reading next row (h) gives us a different value of CITY. This closes the current MSICOS, confirms the current reduction (Grenoble, 2016-09-16, 9999-12-31) and starts a new MSICOS.

When all the source rows have been read, the projection of H_EMPLOYEE on column CITY looks like:

CITY	start	end
...	...	...
Genève	...	...
Grenoble	2016-02-16	2016-03-22
Grenoble	2016-09-16	9999-12-31
Lille	...	...
...	...	...

Translating this iterative algorithm in SQLfast is straightforward (Script 10.23). Tuple (city, sta, end) represents the new row that has just been read and tuple (curCity, curSta, curEnd) stores the current state of the reduction. The blue section is the core of the procedure. It checks whether the new row extends the reduction or if it closes it.

```
create temp table H_CITY(City char(10),start date,end date);
set first = 1;
for city,sta,end = [select CITY,start,end
                    from   H_EMPLOYEE
                    order by CITY,start];
    if ($first$);
        set curCity,curSta,curEnd = $city$,$sta$,$end$;
        set first = 0;
    else;
        if ('$city$' = '$curCity$' and '$sta$' <= '$curEnd$');
            if ('$end$' > '$curEnd$') set curEnd = $end$;
        else;
            insert into H_CITY
            values ('$curCity$','$curSta$','$curEnd$');
            set curCity,curSta,curEnd = $city$,$sta$,$end$;
        endif;
    endif;
endfor;
if (not $first$)
    insert into H_CITY
    values ('$curCity$','$curSta$','$curEnd$');
```

Script 10.23 - SQLfast procedure that stores in table H_CITY the projection of H_EMPLOYEE on column CITY

As with most scripting languages, iterative scripts written in SQLfast may be slow when running against large temporal tables. For this reason, the SQLfast distribution includes a Python library (**LTemp.py**) offering a series of efficient functions that process temporal data.

From this library, function **project** computes the projection of a source table on a list of columns and stores the result in a target table. The arguments of this function are expressed as a query in an SQL-like mini-language. For example, the projection computed by Scripts 10.22 and 10.23 can be described by expression:

```
select CITY from H_EMPLOYEE into H_CITY
```

A `where` clause can also be specified:

```
select CITY from H_EMPLOYEE where SALARY > 6000 into H_CITY
```

This function is used as shown in Script 10.24.[6]

```
function status = LTemp:project
                  {select CITY from H_EMPLOYEE into H_CITY};
if ($status$ = 0) select * from H_CITY;
```

Script 10.24 - Using function **project** from library LTemp

A simplified version of this function, called **reduce**,[7] modifies the source table itself by reducing its MSICOS. Script 10.25 shows how it can be used to return the projection of H_EMPLOYEE on column CITY. It is useful notably to reduce intermediate results in complex temporal data processing, for instance when computing aggregates, as we will see later.

```
create temp table H_CITY as
  select CITY,start,end from H_EMPLOYEE where SALARY > 6000;
function status = LTemp:reduce {select CITY from H_CITY};
if ($status$ = 0) select * from H_CITY;
```

Script 10.25 - Using function **reduce** from library LTemp

10.9 Temporal join

In non temporal databases, *joining* tables, that is, applying a *join* operator, is the most common way to combine data from several tables. For example, extending the data of current employees with the theme of the project they are working on will be performed by *joining* views EMPLOYEE and PROJECT (Script 10.26).

6. Result *status* can be ignored, so that we can also write:
   `function LTemp:project {select CITY from H_EMPLOYEE into H_CITY}`
7. Reminder: the standard name of the **reduce** operator in the vocabulary of temporal databases, is **coalesce**.

```
select CODE,NAME,E.PROJECT,THEME
from   EMPLOYEE E,PROJECT P
where  E.PROJECT = P.TITLE;

select CODE,NAME,E.PROJECT,THEME
from   EMPLOYEE E join PROJECT P
       on (E.PROJECT = P.TITLE);
```

Script 10.26 - Two common forms for *joining* non temporal tables

These forms of *join* produce the same result:

```
+------+-----------+-----------+------------------+
| CODE | NAME      | PROJECT   | THEME            |
+------+-----------+-----------+------------------+
| C45  | Carlier   | BIOTECH   | Biotechnology    |
| A237 | Antoine   | AGRO-2000 | Crop improvement |
| M158 | Mercier   | BIOTECH   | Biotechnology    |
| D107 | Delecourt | BIOTECH   | Biotechnology    |
| G96  | Godin     | AGRO-2000 | Crop improvement |
+------+-----------+-----------+------------------+
```

Just like for the *projection*, studying the *join* operator between temporal tables starts with a naive trial (10.27).

```
select CODE,NAME,E.PROJECT,THEME,E.start,E.end
from   H_EMPLOYEE E,H_PROJECT P
where  E.PROJECT = P.TITLE
and    E.start = P.start;
```

Script 10.27 - Joining temporal tables H_EMPLOYEE and H_PROJECT - First trial

The result below shows that the result is wrong. Since employees at each instant of their life worked on a project, the result must include at least one row for each employee state, which is not the case.

```
+------+----------+-----------+------------------+------------+------------+
| CODE | NAME     | PROJECT   | THEME            | start      | end        |
+------+----------+-----------+------------------+------------+------------+
| D122 | Declercq | BIOTECH   | Genetic engi[..] | 2015-06-15 | 2016-03-21 |
| A68  | Albert   | SURVEYOR  | Satellite mo[..] | 2015-10-30 | 2016-10-22 |
| M158 | Mercier  | SURVEYOR  | Satellite mo[..] | 2015-10-30 | 2017-01-14 |
| C45  | Carlier  | AGRO-2000 | Crop improve[..] | 2016-08-31 | 2016-12-11 |
| G96  | Godin    | AGRO-2000 | Crop improve[..] | 2016-08-31 | 2016-12-11 |
+------+----------+-----------+------------------+------------+------------+
```

This query clearly does not work as intended. Rows from both tables are joined on the basis of rare and independent events: when a change of state of an employee accidently coincides with an unrelated change of state of his project.

Let us solve a simple example of what we expect when joining both tables. First, we consider state [2015-02-21, 2015-10-30] of employee M158 (Mercier, then living in Paris):

CODE	NAME	CITY	PROJECT	start	end
M158	Mercier	Paris	BIOTECH	2015-02-21	2015-10-30

The interval of this state overlap with the interval of several states of table H_PROJECT. So, we conclude that this state *references*, not one state of H_PROJECT, as is the rule in non temporal data, but **all the states** of H_PROJECT whose interval **overlaps** with that of M158 employee state. They are easy to identify:

TITLE	THEME	BUDGET	start	end
BIOTECH	Biotechnology	180000	2014-11-18	2015-02-27
BIOTECH	Genetic eng[..]	160000	2015-02-27	2015-06-15
BIOTECH	Genetic eng[..]	120000	2015-06-15	2016-08-23

Joining these rows produces the expected result:

CODE	NAME	..	PROJECT	..	start	end
M158	Mercier	..	BIOTECH	..	2015-02-21	2015-02-27
M158	Mercier	..	BIOTECH	..	2015-02-27	2015-06-15
M158	Mercier	..	BIOTECH	..	2015-06-15	2015-10-30

The reasoning can be represented graphically as shown in Figure 10.9.

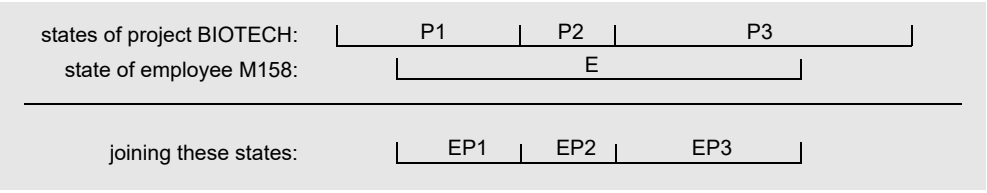


Figure 10.9 - Joining state E of table E_EMPLOYEE with overlapping states P1, P2, P3 of table H_PROJECT

We observe that the intervals of the first and last H_PROJECT states have been reduced to align them to the interval of H_EMPLOYEE, an operation called *interval clipping*.

Translating these operations into an SQL query is fairly easy (Script 10.28).

```
select CODE,NAME,PROJECT,THEME,
       max(P.start,E.start) as "start",
       min(P.end,E.end) as "end"
from   H_EMPLOYEE E, H_PROJECT P
where  E.PROJECT = P.TITLE
and    P.start < E.end and E.start < P.end;
```

Script 10.28 - Temporal join of H_EMPLOYEE with H_PROJECT

Script 10.29 expresses the extraction of a snapshot from the join.

```
set T = 2016-01-01;
select CODE,NAME,PROJECT,THEME
from  (select CODE,NAME,PROJECT,THEME,
              max(P.start,E.start) as "Start",
              min(P.end,E.end) as "End"
       from   H_EMPLOYEE E, H_PROJECT P
       where  E.PROJECT = P.TITLE
       and    P.start < E.end and E.start < P.end)
where  "Start" <= '$T$' and '$T$' < "End";
```

Script 10.29 - Snapshot of the temporal join of H_EMPLOYEE with H_PROJECT

CODE	NAME	PROJECT	THEME
G96	Godin	BIOTECH	Genetic engineering
D122	Declercq	BIOTECH	Genetic engineering
C45	Carlier	BIOTECH	Genetic engineering
A68	Albert	SURVEYOR	Satellite monitoring
M158	Mercier	SURVEYOR	Satellite monitoring

A more elegant (and reusable) translation of this snapshot is shown in Script 10.30. It defines the view that expresses the join then extracts the snapshot from this view.

```
create view H_EMP_PROJ(CODE,NAME,PROJECT,THEME,start,end) as
  select CODE,NAME,PROJECT,THEME,
         max(P.start,E.start), min(P.end,E.end)
  from   H_EMPLOYEE E, H_PROJECT P
  where  E.PROJECT = P.TITLE
  and    (P.start < E.end) and (E.start < P.end);

select CODE,NAME,PROJECT,THEME
from   H_EMP_PROJ
where  start <= '$T$' and '$T$' < end;
```

Script 10.30 - View-based snapshot of the temporal join of H_EMPLOYEE with H_PROJECT

If this view is used in this query only, expressing it as a Common Table Expression (CTE) could be more appropriate.

10.10 Temporal outer join

Let us consider two tables resulting from the projection of table H_EMPLOYEE:

- table H_EMP1 comprises the projection of H_EMPLOYEE on columns CODE, NAME and CITY ,
- table H_EMP2 comprises the projection of H_EMPLOYEE on columns CODE, SALARY and PROJECT.

Each column of the source table is included in one of these table, except CODE, the entity primary key, which is included in both.

We could think that tables H_EMP1 and H_EMP2 are equivalent to table H_EMPLOYEE, that is, that joining the projections will produce the exact contents of H_EMPLOYEE. So, we could get rid of H_EMPLOYEE and replace it with H_EMP1 and H_EMP2, since, if needed, we always could rebuild H_EMPLOYEE.

This is true under two strict conditions:

- CODE is the entity primary key in each table[8],
- for each state in H_EMP1 with interval v1, there exists in H_EMP2 a sequence of contiguous states with the same value of CODE that covers v1, and conversely.

Said in other words,

- column CODE in each of these tables is a temporal foreign key to the other,
- the temporal projections of these tables on CODE are identical.

The need for outer joins

What would happen if a state is missing in one of the tables, H_EMP2 for example (see Section 10.8.2)? Simple: some of the matching state(s) in the other table (H_EMP1) will be lost in the join. This would be the case, for example, if, in some states of the source table, both SALARY and PROJECT are *null* while columns NAME and/or CITY are not *null*. These states in H_EMPLOYEE would generate states in H_EMP1 but none in table H_EMP2, leaving gaps in this table. In joining H_EMP1 and H_EMP2 into H_EMP12, we clearly lose information, so that H_EMP12 and H_EMPLOYEE no longer are equivalent.

In *non temporal* databases, this problem can be solved with a variant of the join operator named *outer join*. When joining two tables T1 and T2, a missing row in

8. More generally, CODE is a unique key in each snapshot of these tables.

table T2 is replaced by *null* values, therefore preserving the data of T1. Reading the join arguments from left to right, this form is called *left outer join*. If the missing rows are in T1, we name it *right outer join*. If both tables may have missing rows, the operator is a *full outer join*. In the relational model, the ordinary join is more precisely called *inner join*, as opposed to expression *outer join*.

Temporal outer join: description

To study the outer join operator applied to temporal data, we consider temporal tables PRO and EMP of Figure 10.10. They are reduced versions of tables H_PROJECT and H_EMPLOYEE studied so far. Table PRO describes projects `'p1'` and `'p2'` while table EMP describes the evolution of employees `'e1'` and `'e2'`. Columns ProID and EmpID are entity primary keys of their respective tables. Column Proj of EMP is a foreign key to table PRO.

table PRO

ProID	Name	Budg	start	end
p1	bio	120	2	4
p1	bio	105	6	9
p1	bio	130	10	12
p1	bio	142	12	13
p2	tec	125	3	5
p2	tec	130	5	8
p2	tec	142	10	12

table EMP

EmpID	Sal	Proj	start	end
e1	10	p1	1	3
e1	12	p1	3	5
e1	15	p1	6	7
e1	16	p1	8	11
e2	12	p1	3	6
e2	15	p2	7	10
e2	16	p2	11	12

Figure 10.10 - The two source temporal tables to study the outer join operator

The states of these tables are fairly *chaotic*: the history of each table is incomplete, and even incorrect: several states are missing and referential integrity is violated. Some PRO states do not match any EMP state while some EMP states do not match PRO states. The idiosyncrasies of the data are better shown in Figure 10.11. In short, a good basis to study the join of the table on Proj = ProID!

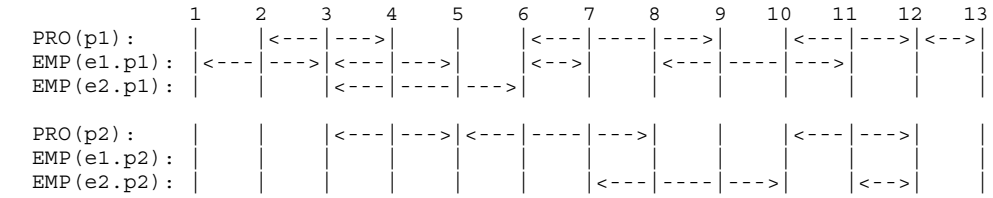


Figure 10.11 - The time lines of each PRO entity and its associated EMP entities

As a consequence, a standard temporal join (Figure 10.12) will lose data from both tables. Figure 10.13 shows that attempting to recover the source data from their inner join through temporal projection (Figure 10.13) entails the loss of many states. This is particularly clear in the synthetic view of Figure 10.14.

EmpID	Sal	ProID	Name	Budg	Start	End
e1	10	p1	bio	120	2	3
e1	12	p1	bio	120	3	4
e1	15	p1	bio	105	6	7
e1	16	p1	bio	105	8	9
e1	16	p1	bio	130	10	11
e2	12	p1	bio	120	3	4
e2	15	p2	tec	130	7	8
e2	16	p2	tec	142	11	12

Figure 10.12 - Inner join of PRO and EMP on columns ProID = Proj

table PRO

ProID	Name	Budg	start	end
p1	bio	120	2	4
p1	bio	105	6	7
p1	bio	105	8	9
p1	bio	130	10	11
p2	tec	130	7	8
p2	tec	142	11	12

table EMP

EmpID	Sal	Proj	start	end
e1	10	p1	2	3
e1	12	p1	3	4
e1	15	p1	6	7
e1	16	p1	8	9
e1	16	p1	10	11
e2	12	p1	3	4
e2	15	p2	7	8
e2	16	p2	11	12

Figure 10.13 - (Unsuccessful) tentative recovery of source data from their inner join

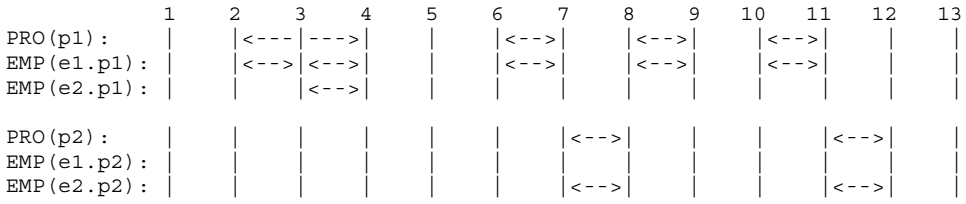


Figure 10.14 - The time lines of each PRO entity and its associated EMP entities derived from the inner join of their tables

Let us now go back to the concept of *outer join* applied to temporal data.

Considering tables PRO and EMP, *in this order*, the *temporal left outer join* preserves each state p of PRO, extended with the values of EmpID and Sal of each row of EMP the interval of which overlaps with the interval of p. When no row matches in EMP, p is extended with *null* values. Figure 10.15 shows the result of the *left outer join* of PRO with EMP. We observe that all states of PRO are preserved, sometimes complemented by *null* values when no matching state has been found in EMP.

ProID	Name	Budg	EmpID	Sal	Start	End
p1	bio	105	--	--	7	8
p1	bio	130	--	--	11	12
p1	bio	142	--	--	12	13
p1	bio	120	e1	10	2	3
p1	bio	120	e1	12	3	4
p1	bio	105	e1	15	6	7
p1	bio	105	e1	16	8	9
p1	bio	130	e1	16	10	11
p1	bio	120	e2	12	3	4
p2	tec	125	--	--	3	5
p2	tec	130	--	--	5	7
p2	tec	142	--	--	10	11
p2	tec	130	e2	15	7	8
p2	tec	142	e2	16	11	12

Figure 10.15 - Left outer join of PRO with EMP on columns ProID = Proj

The *right outer join* of PRO with EMP, that preserves all the states of EMP, is built similarly. The *full outer join* preserves the data of both source tables. Its result is shown in Figures 10.16. It can be computed as the union of the results of the *left* and *right outer joins*.

ProID	Name	Budg	EmpID	Sal	Start	End
p1	bio	105	--	--	7	8
p1	bio	130	--	--	11	12
p1	bio	142	--	--	12	13
p1	--	--	e1	10	1	2
p1	bio	120	e1	10	2	3
p1	bio	120	e1	12	3	4
p1	--	--	e1	12	4	5
p1	bio	105	e1	15	6	7
p1	bio	105	e1	16	8	9
p1	--	--	e1	16	9	10
p1	bio	130	e1	16	10	11
p1	bio	120	e2	12	3	4
p1	--	--	e2	12	4	6
p2	tec	125	--	--	3	5
p2	tec	130	--	--	5	7
p2	tec	142	--	--	10	11
p2	tec	130	e2	15	7	8
p2	--	--	e2	15	8	10
p2	tec	142	e2	16	11	12

Figure 10.16 - Full outer join of PRO with EMP on columns ProID = Proj

Implementing temporal outer joins

Now, we have to find an algorithm that computes the *left outer join* of PRO and EMP based on the observations we made here above. To this aim, we will apply a well known problem solving strategy: *transforming our problem into another one already solved*.

The examination of EMP shows a set of *missing intervals wrt* of PRO. An interval (t1,t2) is called *missing* in EMP wrt project entity p if:

– (t1,t2) exists in PRO for p,

– there no consecutive states in EMP for which Proj = p and the intervals of which cover (t1,t2).

The missing interval of EMP *wrt* PRO are represented in red in Figure 10.17. The history of project p1 has no matching states in EMP for intervals [7,8) and [11,13). So, they are the four missing intervals of EMP.

In an *inner join*, these intervals will lead to the loss of the matching intervals in PRO, and in a *left outer join*, they will complement the states of PRO with *null* values.

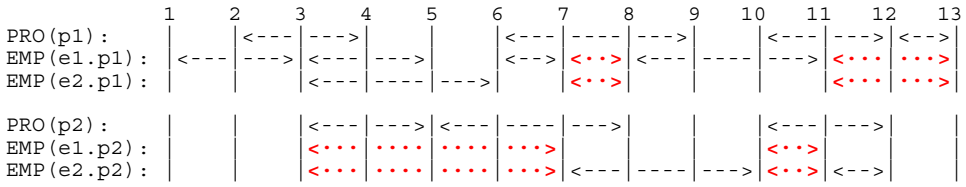


Figure 10.17 - Identifying missing intervals in the time lines of each PRO entity and of its associated EMP entities

Let us fill them with *null* values. Figure 10.18 shows the contents of table EMP in which we have inserted, for each missing interval, a row made of *null* values for columns EmpID and Sal.

EmpID	Sal	Proj	start	end
e1	10	p1	1	3
e1	12	p1	3	5
e2	12	p1	3	6
e1	15	p1	6	7
--	--	p1	7	8
e1	16	p1	8	11
--	--	p1	11	13
--	--	p2	3	7
e2	15	p2	7	10
--	--	p2	10	11
e2	16	p2	11	12

Figure 10.18 - Table EMP complemented with *null* missing intervals wrt PRO

This modified table EMP is just a transient working data set. It is incorrect since it violates the primary key and *not null* constraints. However it exhibits an interesting property: it constitutes a complete history of employees. This means that each state of PRO will match at least one state in modified EMP. This has an important consequence: the *left outer join* of PRO with EMP can be computed through a simple *inner join*.

The missing states are computed and stored in table EMP_MISSING by Script 10.31, then added to the source states to constitute table EMP_COMPLETED (Script 10.32).

```
create table EMP_MISSING as
  select EMPa.Proj, null as "EmpID",null as "Sal",
         EMPa.end as Tstart, EMPb.start as Tend
  from   EMP EMPa, EMP EMPb
  where  EMPa.Proj = EMPb.Proj
  and    EMPa.end < EMPb.start
  and not exists (select *
                  from   EMP EMPc
                  where  EMPc.Proj = EMPa.Proj
                  and    EMPc.start < EMPb.start
                  and    EMPa.end < EMPc.end)
     union
  select EMP.Proj,null,null,
         min(PRO.start) as Tstart,min(EMP.start) as Tend
  from   PRO, EMP
  where  PRO.ProID = EMP.Proj
  group by EMP.Proj
  having Tstart < Tend
     union
  select EMP.Proj,null,null,
         max(EMP.end) as Tstart,max(PRO.end) as Tend
  from   PRO, EMP
  where  PRO.ProID = EMP.Proj
  group by EMP.Proj
  having Tstart < Tend
     union
  select null as EmpID,null as Sal,ProID as Proj,
         min(start) as "start",max(end) as "end"
  from   PRO
  where  ProID not in (select Proj from EMP)
  group by ProID;
```

Script 10.31 - Generating the missing states of source table EMP

```
create table EMP_COMPLETED as
  select Proj,EmpID,Sal,start as Tstart,end as Tend from EMP
     union
  select Proj,null,null,Tstart,Tend from EMP_MISSING;
```

Script 10.32 - Table EMP is completed with its missing states

The algorithm that computes the missing intervals is made up of four parts:

– The first part extracts the *internal* missing intervals, that is, those that constitutes gaps in EMP.

– The second and third parts create left and right *external* missing intervals when the history of the current project starts before and/or ends after the series of states of EMP that reference this project. For instance, the third part creates missing state [11,13) of project p1 in EMP.
– The fourth part copes with project entities that are **not** referenced by employees. A *null* state is created in EMP for each value of ProID in PRO that does not appear in column Proj in EMP. By the way, this part also fill **empty** table EMP with a *null* state for each project that will preserve all the states of PRO.

Finally, a simple inner join of PRO with EMP_COMPLETED produces the desired *left outer join* in table LEFT (Script 10.33). A similar procedure will generate the right outer join in table RIGHT.

```
create table LEFT as
  select PRO.ProID,Name,Budg,EmpID,Sal,
         max(PRO.start,EMPC.Tstart) as "Start",
         min(PRO.end,EMPC.Tend) as "End"
  from   PRO, EMP_COMPLETED EMPC
  where  PRO.ProID = EMPC.Proj
  and    (PRO.start < EMPC.Tend) and (EMPC.Tstart < PRO.end);
```

Script 10.33 - A simple inner join produces the *left outer join* of PRO with EMP

The full outer join is computed as the union of LEFT and RIGHT (Script 10.34).

```
select * from LEFT union select * from RIGHT;
```

Script 10.34 - Full outer join of PRO and EMP

Computing outer joins with the LTemp library

The completed version of EMP *wrt* PRO, called above EMP_COMPLETED, can be more simply (and generally much faster) computed by function **normalize** of temporal library **LTemp**. Its argument is a pseudo-SQL query that specifies the source table to be completed (EMP), the reference table (PRO), the join condition (Proj = ProID) and the target table (EMP_COMPLETED).

```
function LTemp:normalize {update EMP wrt PRO on Proj = ProID
                          into EMP_COMPLETED};
```

Script 10.35 - Computing EMP_COMPLETED through function **normalize** of library **LTemp**

If clause `'wrt PRO'` is missing, only internal missing intervals of EMP are added. If clause `'into EMP_COMPLETED'` is missing, the missing intervals are inserted into table EMP itself. So, the following form is also valid:

```
function LTemp:normalize {update EMP on Proj};
```

Reversibility of the outer joins

To convince us that outer joins preserve one or both of their source arguments, just check that the script below produces the exact contents of table EMP:

```
function LTemp:project {select EmpID,Sal,ProID from FULL
                        into NewEMP};
delete from NewEMP where (EmpID,Sal) is (null,null);
```

10.11 Temporal aggregation

In general, an aggregation query for a non-temporal table provides a unique quantity calculated on a set of rows of that table (Script 10.36, top), or for each group of rows constituted according to a grouping criterion (Script 10.36, bottom). Their results are shown in Figure 10.19. Let us remind that, according to the SQL standard(s), the *select list* (the data elements between key words **select** and **from**) may comprise *constants*, elements of the *grouping criterion* and *aggregate functions* only, as well as expressions comprising these elements.

```
select sum(BUDGET) as Tbudget
from   PROJECT

select PROJECT, count(*) as Nemp
from   EMPLOYEE
group by PROJECT;
```

Script 10.36 - Aggregation queries on non temporal data

Tbudget
222000

PROJECT	Nemp
AGRO-2000	2
BIOTECH	3

Figure 10.19 - Aggregated data computed on non temporal data

Querying a snapshots is as easy, since it just is a *current state* that existed in the past. The queries of Script 10.37 show the same statistics extracted from the state on January 1st, 2017 and Figure 10.20 shows the result.

```
select sum(BUDGET) as Tbudget
from   H_PROJECT
where  start <= '2017-01-01' and '2017-01-01' < end;

select PROJECT, count(*) as Nemp
from   H_EMPLOYEE
where  start <= '2017-01-01' and '2017-01-01' < end
group by PROJECT;
```

Script 10.37 - Aggregation queries on a snapshot

Tbudget
550000

PROJECT	Nemp
AGRO-2000	3
BIOTECH	1
SURVEYOR	4

Figure 10.20 - Aggregated data computed on a snapshot (on 2017-01-01)

This was the easy part. Easy, because we have got rid of the time dimension before aggregating the data. Now we will cope with this dimension by deriving *evolution data*, which will prove a bit more complex, notably because the concept of time-dependent aggregation is not as easy to define as it seems to be at first glance.

10.11.1 A first simple problem: counting

Let us start with something (seemingly) simple: showing the evolution, from the creation of a project to now, of the number of employees who worked on this project. Figure 10.21 shows all the states of H_EMPLOYEE related to project AGRO-2000, sorted by increasing values of column start.

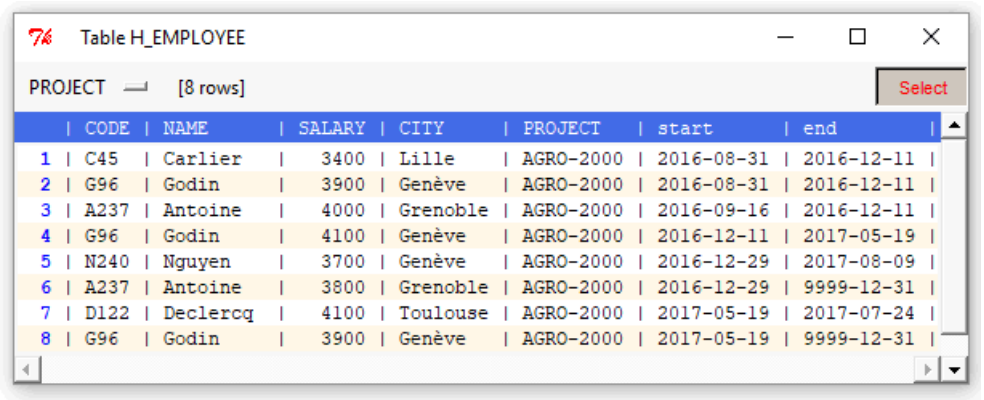


Table H_EMPLOYEE

PROJECT [8 rows] Select

	CODE	NAME	SALARY	CITY	PROJECT	start	end
1	C45	Carlier	3400	Lille	AGRO-2000	2016-08-31	2016-12-11
2	G96	Godin	3900	Genève	AGRO-2000	2016-08-31	2016-12-11
3	A237	Antoine	4000	Grenoble	AGRO-2000	2016-09-16	2016-12-11
4	G96	Godin	4100	Genève	AGRO-2000	2016-12-11	2017-05-19
5	N240	Nguyen	3700	Genève	AGRO-2000	2016-12-29	2017-08-09
6	A237	Antoine	3800	Grenoble	AGRO-2000	2016-12-29	9999-12-31
7	D122	Declercq	4100	Toulouse	AGRO-2000	2017-05-19	2017-07-24
8	G96	Godin	3900	Genève	AGRO-2000	2017-05-19	9999-12-31

Figure 10.21 - Subset of the H_EMPLOYEE table describing the history of the staff of project AGRO-2000

In Figure 10.22, we represent graphically the sequence of the states of the employees who worked (and sometimes still work) on this project.

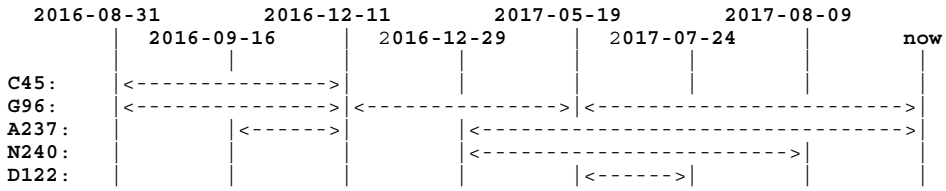


Figure 10.22 - Graphical representation of the life of the employees who worked on project AGRO-2000

To make intervals comparable, we split them into smaller intervals during which no modification occurred for any employee. We call them *stable intervals*. Now, each state of an employee can be defined as a sequence of one or more stable intervals. Since the set of stable intervals is common to all the employees of the project, counting the number of employees in each of these intervals is straightforward, as shown in figure 10.23.

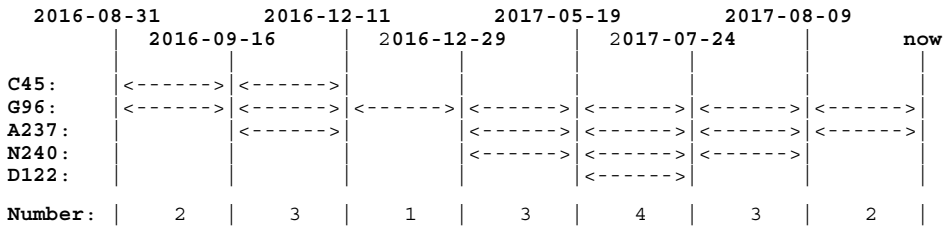


Figure 10.23 - Expressing employee states as a sequence of *stable intervals*

Hence the result we are looking for (Figure 10.24).

PROJECT	Number	Start	End
AGRO-2000	2	2016-08-31	2016-09-16
AGRO-2000	3	2016-09-16	2016-12-11
AGRO-2000	1	2016-12-11	2016-12-29
AGRO-2000	3	2016-12-29	2017-05-19
AGRO-2000	4	2017-05-19	2017-07-24
AGRO-2000	3	2017-07-24	2017-08-09
AGRO-2000	2	2017-08-09	9999-12-31

Figure 10.24 - Evolution of the number of employees in project AGRO-2000

What this small example does not show is that two (or more) successive stable intervals may produce the same statistics. For instance, when, at the same date, an employee leaves a project while this project hires a new employee. In such a case we *reduce* this result by merging such value-equivalent states.

Now, we must implement these reasonings into SQL statements. Clearly, we must proceed in four steps:

– extracting all the events affecting the employees,

– creating the stable intervals,

– counting the number of employees in each stable interval

– and reducing value-equivalent stable interval.

a) Extracting all the events affecting the employees

Each event (i.e., data modification) generates a time point, which generally closes the current state and starts a new one, in which case this time point appears as both start and end values in two states. Except of course the first and last events of a series of successive states, which appear in a single state. To collect these time points, we just extract all the start and end values, without duplicates. Script 10.38 stores them in temporary table EVENT.

```
create temp table EVENT as
  select PROJECT,Instant
  from (select PROJECT,start as Instant from H_EMPLOYEE
          union
        select PROJECT,end as Instant from H_EMPLOYEE);
```

Script 10.38 - Extracting the time points of table T_EMPLOYEE

The contents of this table looks like:

PROJECT	Instant
AGRO-2000	2016-08-31
AGRO-2000	2016-09-16
AGRO-2000	2016-12-11
AGRO-2000	2016-12-29
AGRO-2000	2017-05-19
AGRO-2000	2017-07-24
AGRO-2000	2017-08-09
AGRO-2000	9999-12-31
BIOTECH	2014-11-23
...	...

b) Creating the stable intervals

From table EVENT, we derive the *stable intervals*. Conceptually, the process is simple: with each event E, we associate the lowest Instant value among all the values greater than that of E. An SQL translation is proposed in Script 10.39, followed by the contents of table STABLE_INTERVAL.

The way these intervals are built produces a complete normalized history, even though some intervals may have no value, for instance due to the fact that no project has any employee during them.

```
create temp table STABLE_INTERVAL as
  select PROJECT,
           Instant as Start,
         (select min(Instant)
          from   EVENT
          where  PROJECT = E.PROJECT
          and    Instant > E.Instant) as End
  from   EVENT E
  where  End is not null;
```

Script 10.39 - Building the stable intervals from table EVENT

PROJECT	Start	End
AGRO-2000	2016-08-31	2016-09-16
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-12-11	2016-12-29
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-08-09	9999-12-31
BIOTECH	2014-11-23	2014-12-04
...	...	...

Figure 10.25 - The stable intervals of project AGRO-2000

c) Counting the number of states in each stable interval

Now, we must split the states of H_EMPLOYEE according to the stable intervals computed in table STABLE_INTERVAL to produce the eighteen interval instances of Figure 10.23. For this, we associate with each stable interval all the states of H_EMPLOYEE with which they overlap. This typically is obtained by a temporal join (Script 10.40).

```
create temp view STABLE_EMPLOYEE as
  select S.PROJECT,S.start,S.end
  from   STABLE_INTERVAL S, H_EMPLOYEE E
  where  E.PROJECT = S.PROJECT
  and    E.start <= S.start and S.start < E.end;
```

Script 10.40 - Computing the stable intervals of E_EMPLOYEE

The contents of this view is shown below. Each row represents the contribution of one employee to a stable interval. This could be shown by adding E.CODE to the *select list* of the query.

PROJECT	Start	End
AGRO-2000	2016-08-31	2016-09-16
AGRO-2000	2016-08-31	2016-09-16
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-09-16	2016-12-11
AGRO-2000	2016-12-11	2016-12-29
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2016-12-29	2017-05-19
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-05-19	2017-07-24
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-07-24	2017-08-09
AGRO-2000	2017-08-09	9999-12-31
AGRO-2000	2017-08-09	9999-12-31
BIOTECH	2014-11-23	2014-12-04
...	...	...

Finally, applying the aggregate function is quite easy, since it is nothing more than a standard, non temporal, computation (Script 10.41).

```
select PROJECT,count(*) as Number,start,end
from   STABLE_EMPLOYEE
group by PROJECT,start,end;
```

Script 10.41 - Finally, computing the evolution of the number of employees per project

The result is that of Figure 10.24. By developing the reference of the view in this query by its definition, we obtain the synthetic query of Script 10.42.

```
create temp table E_COUNT(PROJECT,Number,Start,End) as
  select S.PROJECT,count(*),S.start,S.end
  from   H_EMPLOYEE E, STABLE_INTERVAL S
  where  E.PROJECT = S.PROJECT
  and    E.start <= S.start and S.start < E.end
  group by S.PROJECT,S.start,S.end;
```

Script 10.42 - computing the evolution of the number of employees per project - Synthetic query

d) Reducing value-equivalent stable intervals

Consecutive value-equivalent states are merged through the reduce function (Script 10.43).

```
function LTemp:reduce {select PROJECT,Number from E_COUNT};
```

Script 10.43 - Reducing value-equivalent states

10.11.2 Value-based statistics: avg, max and min

Counting states was an excellent exercise to help us solve more complex statistics. Let us start with avg, min and max aggregation functions.

We intend to compute the evolution of the average, minimum and maximum salary for each project. As we did for counting the employees, we start with table STABLE_INTERVAL (Script 10.39 and Figure 10.25). We know that, by construction, the salary of employees never changed during any of those stable interval. So, we can add its unique salary to each interval to form view VALUED_STABLE_INTERVAL (Script 10.44).

```
create temp view VALUED_STABLE_INTERVAL as
  select S.PROJECT,E.SALARY,S.start,S.end
  from   STABLE_INTERVAL S, H_EMPLOYEE E
  where  E.PROJECT = S.PROJECT
  and    E.start <= S.start and S.start < E.end;
```

Script 10.44 - Adding the salary value to each stable interval of E_EMPLOYEE

The contents of this view is shown in Figure 10.26 and is graphically represented in Figure 10.27.

PROJECT	SALARY	Start	End
AGRO-2000	3400	2016-08-31	2016-09-16
AGRO-2000	3900	2016-08-31	2016-09-16
AGRO-2000	3400	2016-09-16	2016-12-11
AGRO-2000	3900	2016-09-16	2016-12-11
AGRO-2000	4000	2016-09-16	2016-12-11
AGRO-2000	4100	2016-12-11	2016-12-29
AGRO-2000	3700	2016-12-29	2017-05-19
AGRO-2000	3800	2016-12-29	2017-05-19
AGRO-2000	4100	2016-12-29	2017-05-19
AGRO-2000	3700	2017-05-19	2017-07-24
AGRO-2000	3800	2017-05-19	2017-07-24
AGRO-2000	3900	2017-05-19	2017-07-24
AGRO-2000	4100	2017-05-19	2017-07-24
AGRO-2000	3700	2017-07-24	2017-08-09
AGRO-2000	3800	2017-07-24	2017-08-09
AGRO-2000	3900	2017-07-24	2017-08-09
AGRO-2000	3800	2017-08-09	9999-12-31
AGRO-2000	3900	2017-08-09	9999-12-31
BIOTECH	...	...	...
...	...	...	...

Figure 10.26 - The stable intervals of H_EMPLOYEE showing their SALARY values (tabular view)

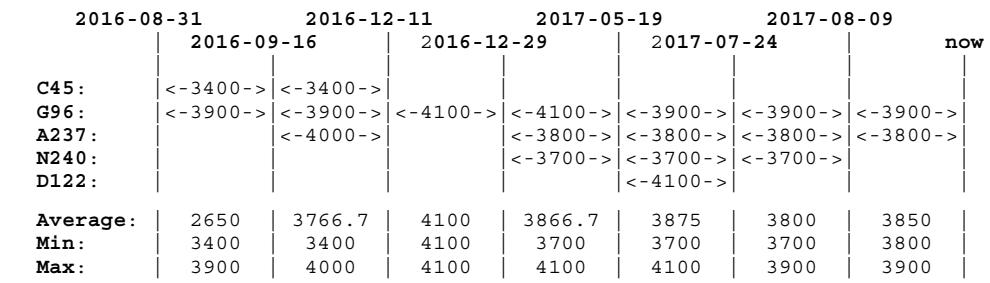


Figure 10.27 - The stable intervals of H_EMPLOYEE showing their SALARY values (graphical view)

Computing the standard statistics (average, minimum, maximum) of SALARY in each stable state is just child's play (Script 10.45).

```
select  PROJECT,
        round(avg(SALARY),1) as Average,
        min(SALARY) as Min,
        max(SALARY) as Max,Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,Start,End;
```

Script 10.45 - Computing standard statistics of the stable interval of H_EMPLOYEE

After reducing the value-equivalent intervals, we get this result:

PROJECT	Average	Min	Max	Start	End
AGRO-2000	3650.0	3400	3900	2016-08-31	2016-09-16
AGRO-2000	3766.7	3400	4000	2016-09-16	2016-12-11
AGRO-2000	4100.0	4100	4100	2016-12-11	2016-12-29
AGRO-2000	3866.7	3700	4100	2016-12-29	2017-05-19
AGRO-2000	3875.0	3700	4100	2017-05-19	2017-07-24
AGRO-2000	3800.0	3700	3900	2017-07-24	2017-08-09
AGRO-2000	3850.0	3800	3900	2017-08-09	9999-12-31
BIOTECH	...	...	...	...	...
...	...	...	...	...	...

10.11.3 Value-based statistics: sum

This last exercise is a bit more complex because it involves the *length of the intervals*. Still considering view VALUED_STABLE_INTERVAL (and Figure 10.27), we would like to know the *total salary cost* of each interval. Adding expression sum(SALARY) to query 10.45 is useless for two reasons: the cost we are looking for depends on the number of days in each interval and SALARY is a monthly salary.

– The number of days in interval [d1,d2) is given by function `durationDays(d1,d2)`. So, the length of an interval is obtained by expression `durationDays(start,end)`. However, this does not work for current states, for which end = '9999-12-31'. Replacing **end** with the current date ($date$) is also not satisfactory since its value changes from day to day, leading to unstable lengths. In such case, we suggest to replace **end** by the more realistic constant **maxValue**, computed as the highest **end** date of H_EMPLOYEE + 100 days:

```
extract maxDate = select addToDate(max(end),100)
                  from   H_EMPLOYEE
                  where  end < '$Tfuture$';
```

The length of interval [start,end) will then be computed by expression:

```
durationDays(start,min(end,'$maxDate$'))
```

– Converting a monthly salary into a daily salary is a bit tricky. Indeed, months have different lengths but cost all the same salary, so that we are forced to work with some kind of average daily salary. On the average, in one year, 1 month comprises 365/12 = 30.4167 days. Let them be rounded to 30.42 days for this exercise.

Therefore, the salary cost of interval [start,end) is computed by expression:

```
SALARY/30.42*durationDays(start,min(end,'$maxDate$'))
```

The correct solution is shown in Script 10.46. Its result is shown in Figure 10.28, where TCost has been rounded to one decimal (`round()` function not shown).

```
select  PROJECT,
        sum(SALARY/30.42 *
            durationDays(Start,min(End,'$maxDate$'))) as TCost,
        Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,Start,End;
```

Script 10.46 - Computing the total salary cost of each stable interval

PROJECT	TCost	Start	End
AGRO-2000	3839.6	2016-08-31	2016-09-16
AGRO-2000	31946.1	2016-09-16	2016-12-11
AGRO-2000	2426.0	2016-12-11	2016-12-29
AGRO-2000	53767.3	2016-12-29	2017-05-19
AGRO-2000	33629.2	2017-05-19	2017-07-24
AGRO-2000	5996.1	2017-07-24	2017-08-09
AGRO-2000	25312.3	2017-08-09	9999-12-31
BIOTECH	...	...	...
...	...	...	...

Figure 10.28 - Salary cost of stable intervals

The same query, slightly modified, will compute the total salary cost (so far) of each project (Script 10.47).

```
select  PROJECT,
        sum(SALARY/30.42 *
            durationDays(Start,min(End,'$maxDate$'))) as TCost
from    VALUED_STABLE_INTERVAL
group by PROJECT;
```

Script 10.47 - Computing the total salary cost of each project

PROJECT	TCost
AGRO-2000	156916.5
BIOTECH	280328.7
SURVEYOR	243445.1

10.11.4 Multidimensional statistics

The statistics discussed so far have considered one dimension of employee entities, namely PROJECT or CITY, that appeared in source table VALUED_STABLE_INTERVAL. The structure of this table can be formally defined by this pattern[9]:

PROJECT,interval $\rightarrow\!\!\rightarrow$ SALARY

which tells that a *project* and an *interval* together determine a *set of salaries*, on which various aggregate functions can be applied. The presentation below of the data of Figure 10.26 makes this relationship explicit:

PROJECT	SALARY	Start	End
AGRO-2000	3400,3900	2016-08-31	2016-09-16
AGRO-2000	3400,3900,4000	2016-09-16	2016-12-11
AGRO-2000	4100	2016-12-11	2016-12-29
AGRO-2000	3700,3800,4100	2016-12-29	2017-05-19
AGRO-2000	3700,3800,3900,4100	2017-05-19	2017-07-24
AGRO-2000	3700,3800,3900	2017-07-24	2017-08-09
AGRO-2000	3800,3900	2017-08-09	9999-12-31
...	...	...	...

This idea can be generalized to grouping criteria comprising more than one column. For example, let us examine this one:

PROJECT,CITY,interval $\rightarrow\!\!\rightarrow$ SALARY

through which we associate a set of salaries with each couple (PROJECT,CITY), partitioned by stable intervals. Tables EVENT, STABLE_INTERVAL and VALUED_STABLE_INTERVAL are computed accordingly (Script 10.48).

```
create table EVENT as
  select PROJECT,CITY,Instant
  from (select PROJECT,CITY,start as Instant from H_EMPLOYEE
        union
        select PROJECT,CITY,end as Instant from H_EMPLOYEE);
```

9. Yes, it's an *embedded multivalued dependency*, but don't tell anyone!

```
create table STABLE_INTERVAL as
  select PROJECT,CITY,
         Instant as Start,
         (select min(Instant)
          from   EVENT
          where  (PROJECT,CITY) = (V.PROJECT,V.CITY)
          and    Instant > V.Instant) as End
  from   EVENT V
  where  End is not null;

create table VALUED_STABLE_INTERVAL as
  select S.PROJECT,S.CITY,E.CODE,E.SALARY,S.Start,S.End
  from   STABLE_INTERVAL S,H_EMPLOYEE E
  where  (S.PROJECT,S.CITY) = (E.PROJECT,E.CITY)
  and    S.Start >= E.start and S.End <= E.end;
```

Script 10.48 - Computing interval tables for grouping criterion (PROJECT,CITY)

The contents of VALUED_STABLE_INTERVAL now looks like this (the codes of the employees have been added for clarity[10]):

PROJECT	CITY	CODE	SALARY	Start	End
AGRO-2000	Genève	G96	3900	2016-08-31	2016-12-11
AGRO-2000	Genève	G96	4100	2016-12-11	2016-12-29
AGRO-2000	Genève	G96	4100	2016-12-29	2017-05-19
AGRO-2000	Genève	N240	3700	2016-12-29	2017-05-19
AGRO-2000	Genève	G96	3900	2017-05-19	2017-08-09
AGRO-2000	Genève	N240	3700	2017-05-19	2017-08-09
AGRO-2000	Genève	G96	3900	2017-08-09	9999-12-31
AGRO-2000	Grenoble	A237	4000	2016-09-16	2016-12-11
AGRO-2000	Grenoble	A237	3800	2016-12-29	9999-12-31
AGRO-2000	Lille	C45	3400	2016-08-31	2016-12-11
AGRO-2000	Toulouse	D122	4100	2017-05-19	2017-07-24
BIOTECH	...	...	...	...	...

Figure 10.29 - Contents of table VALUED_STABLE_INTERVAL following pattern PROJECT,CITY,interval $\rightarrow\!\!\rightarrow$ CODE,SALARY

We note that the sequence of stable intervals of cities within AGRO-2000 are independent. For instance, interval [2016-09-16,2016-12-11) of (AGRO-2000,Grenoble) is meaningless for (AGRO-2000,Genève). Each row describes the contribution of one employee to one stable interval of a city in a project. (AGRO-2000,Genève) comprises 5 stable intervals with two contributors, G96 and N240. G96 contributes to 5 intervals (5 rows) while N240 appears in 2 intervals (2 rows) for a total of 7 contributions (7 rows).

10. So that the pattern actually is PROJECT,CITY,interval $\rightarrow\!\!\rightarrow$ CODE,SALARY

The contents of this table, limited to project AGRO-2000, is represented graphically in Figure 10.30, in which city names have been abbreviated.

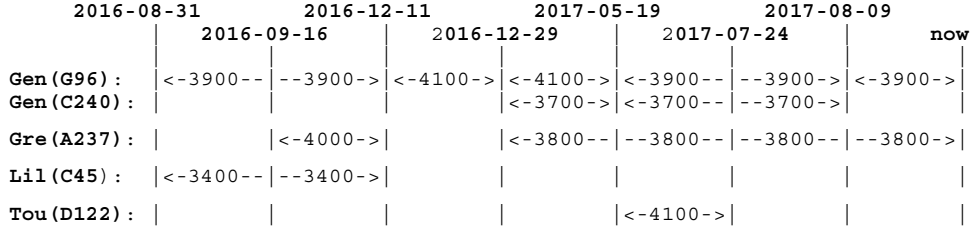


Figure 10.30 - Distribution among stable intervals of the H_EMPLOYEE states of the employees who contributed to project AGRO-2000, grouped by the city of the contribution

Now, we are ready to query table VALUED_STABLE_INTERVAL. We will examine how to count, compute average and sums, as we did for unidimensional statistics.

a) Counting employees

Script 10.49 computes the evolution of the number of employees on each project from each city. It is a natural extension of Script 10.41.

```
select PROJECT,CITY,count(*) as Number,Start,End
from   VALUED_STABLE_INTERVAL
group by PROJECT,CITY,Start,End
order by PROJECT,CITY,Start,end;
```

Script 10.49 - computing the evolution of the number of employees per project and city (reduction not shown)

PROJECT	CITY	Number	Start	End
AGRO-2000	Genève	1	2016-08-31	2016-12-29
AGRO-2000	Genève	2	2016-12-29	2017-08-09
AGRO-2000	Genève	1	2017-08-09	2017-12-31
AGRO-2000	Grenoble	1	2016-09-16	2016-12-11
AGRO-2000	Grenoble	1	2016-12-29	2017-12-31
AGRO-2000	Lille	1	2016-08-31	2016-12-11
AGRO-2000	Toulouse	1	2017-05-19	2017-07-24
BIOTECH	...	...	...	...
...	...	...	...	...

b) Computing average salaries

In Script 10.50, a natural extension of the query of Script 10.45, we compute the evolution of the average salary of the employees of each project from each city.

```
select  PROJECT,CITY,
        group_concat(distinct SALARY) as Salaries,
        round(avg(SALARY),1) as Average,Start,End
from    VALUED_STABLE_INTERVAL
group by PROJECT,CITY,Start,End
order by PROJECT,CITY,Start;
```

Script 10.50 - Computing the average salary in each stable interval of 'AGRO-2000'

PROJECT	CITY	Salaries	Average	Start	End
AGRO-2000	Genève	3900	3900.0	2016-08-31	2016-12-11
AGRO-2000	Genève	4100	4100.0	2016-12-11	2016-12-29
AGRO-2000	Genève	4100,3700	3900.0	2016-12-29	2017-05-19
AGRO-2000	Genève	3900,3700	3800.0	2017-05-19	2017-08-09
AGRO-2000	Genève	3900	3900.0	2017-08-09	9999-12-31
AGRO-2000	Grenoble	4000	4000.0	2016-09-16	2016-12-11
AGRO-2000	Grenoble	3800	3800.0	2016-12-29	9999-12-31
AGRO-2000	Lille	3400	3400.0	2016-08-31	2016-12-11
AGRO-2000	Toulouse	4100	4100.0	2017-05-19	2017-07-24
BIOTECH	...	...	...	...	...
...	...	...	...	...	...

c) Computing total salary costs

Same exercise for the sum statistics. Query 10.51 is an extension of Script 10.46.

```
select PROJECT,CITY
       sum(SALARY/30.42 *
           durationDays(Start,min(End,'$maxDate$'))) as TCost,
       Start,End
from   VALUED_STABLE_INTERVAL
group by PROJECT,CITY,Start,End;
```

Script 10.51 - Computing the evolution of the total salary cost of each stable interval

PROJECT	CITY	TCost	Start	End
AGRO-2000	Genève	13042.6	2016-08-31	2016-12-11
AGRO-2000	Genève	2419.7	2016-12-11	2016-12-29
AGRO-2000	Genève	36059.0	2016-12-29	2017-05-19
AGRO-2000	Genève	20432.7	2017-05-19	2017-08-09
AGRO-2000	Genève	74803.3	2017-08-09	9999-12-31
AGRO-2000	Grenoble	11278.7	2016-09-16	2016-12-11
AGRO-2000	Grenoble	100668.9	2016-12-29	9999-12-31
AGRO-2000	Lille	11370.5	2016-08-31	2016-12-11
AGRO-2000	Toulouse	8872.1	2017-05-19	2017-07-24
BIOTECH	...	...	...	...
...	...	...	...	...

d) Statistics on a subset of the aggregation criterion

Statistics on *non temporal data* can also be analyzed through the pattern we used to describe aggregation criteria and values to aggregate. We can, for instance, derive a synthetic table (let us call it PC_S_EMPLOYEE) based on this formula, in which the temporal dimension is discarded:

PROJECT,CITY $\rightarrow\!\!\rightarrow$ SALARY

Such a table can be extracted through this query:

```
create table PC_S_EMPLOYEE as
select PROJECT,CITY,SALARY
from   EMPLOYEE;
```

The nice feature of this table is that it provides the support of several aggregation criteria: (PROJECT,CITY) of course, but also its subsets (PROJECT) and (CITY). The following query is perfectly valid:

```
select PROJECT,
       count(*) as Number,
       avg(SALARY) as AvgSalary
from   PC_S_EMPLOYEE
group by PROJECT;
```

Unfortunately, this does not work for temporal data!

Temporal table VALUED_STABLE_INTERVAL built by the queries of Script 10.48 **does not allow** data grouping by PROJECT or CITY alone. Indeed, as the life lines of Figure 10.30 shows it, the events of, say, (AGRO-2000, Genève) are independent of (and therefore different from) those of, say, (AGRO-2000, Grenoble). As a consequence, grouping data by PROJECT or CITY alone would be meaningless. The following query **will not** provide the evolution of the number of employees in each project:

```
select PROJECT,count(*) as Number,Start,End
from   VALUED_STABLE_INTERVAL
group by PROJECT,Start,End;
```

We will address this restriction in the next section.

10.11.5 Temporal data with non-partitioned intervals

The history generated by the procedures developed so far are based on a partition of the source rows according to one or more columns. This partition is defined by expressions such as this one:

PROJECT,CITY,interval $\rightarrow\!\!\rightarrow$ SALARY

Each subset of the partition is identified by a couple (p,c) such that H_EMPLOYEE includes at least one row where PROJECT = p and CITY = c. This subset divides its time line into successive stable intervals. As we have observed, the divisions of two subsets do not match, so that they cannot be compared. This is why these stable intervals cannot be used to derive more coarse statistics such as those defined by PROJECT,interval ⟶⟶ SALARY or CITY,interval ⟶⟶ SALARY.

In this section, we will show how to compute stable intervals **common** to all the couples (p,c), that is, a division of the time line synchronous with that of all these couples. For this, we first create the list of all the events of H_EMPLOYEE, independently of any partitioning criterion. We adapt the query of Script 10.38 by removing the reference to PROJECT (Script 10.52).

```
create temp table EVENT as
  select Instant
  from (select start as Instant from H_EMPLOYEE
          union
        select end as Instant from H_EMPLOYEE);
```

Script 10.52 - Extracting the time points of table H_EMPLOYEE

From this table, we build the stable intervals with the algorithm of Script 10.53 (derived from that of Script 10.39).

```
create temp table STABLE_INTERVAL as
  select Instant as Start,
         (select min(Instant)
          from   EVENT
          where  Instant > E.Instant) as End
  from   EVENT E
  where  End is not null;
```

Script 10.53 - Building the stable intervals from table EVENT

Finally, we complete these intervals with all the values from H_EMPLOYEE that will be used to compute the desired statistics (Script 10.54).

```
create temp table VALUED_STABLE_INTERVAL as
  select E.CODE,E.SALARY,E.CITY,E.PROJECT,S.Start,S.End
  from   STABLE_INTERVAL S, H_EMPLOYEE E
  where  E.Start <= S.Start and S.Start < E.End;
```

Script 10.54 - Completing stable intervals with additional data from H_EMPLOYEE

The new state of table VALUED_STABLE_INTERVAL is shown in Figure 10.31 (sorted by PROJECT,CITY,Start). Each row describes the contribution of one employee to one interval. An employee may contribute to several intervals (poten-

tially with the same values of PROJECT, CITY and SALARY) and an interval may be contributed by several employees.

PROJECT	CITY	CODE	SALARY	Start	End
AGRO-2000	Genève	G96	3900	2016-08-31	2016-09-16
AGRO-2000	Genève	G96	3900	2016-09-16	2016-10-22
AGRO-2000	Genève	G96	3900	2016-10-22	2016-12-08
AGRO-2000	Genève	G96	3900	2016-12-08	2016-12-11
AGRO-2000	Genève	G96	4100	2016-12-11	2016-12-29
AGRO-2000	Genève	G96	4100	2016-12-29	2017-01-14
AGRO-2000	Genève	N240	3700	2016-12-29	2017-01-14
AGRO-2000	Genève	G96	4100	2017-01-14	2017-04-03
AGRO-2000	Genève	N240	3700	2017-01-14	2017-04-03
AGRO-2000	Genève	G96	4100	2017-04-03	2017-05-19
AGRO-2000	Genève	N240	3700	2017-04-03	2017-05-19
AGRO-2000	Genève	N240	3700	2017-05-19	2017-07-24
AGRO-2000	Genève	G96	3900	2017-05-19	2017-07-24
AGRO-2000	Genève	N240	3700	2017-07-24	2017-08-09
AGRO-2000	Genève	G96	3900	2017-07-24	2017-08-09
AGRO-2000	Genève	G96	3900	2017-08-09	9999-12-31
AGRO-2000	Grenoble	A237	4000	2016-09-16	2016-10-22
...	...	...	...	...	...
BIOTECH	Genève	G96	3400	2014-11-23	2014-12-04
BIOTECH	Genève	G96	3400	2014-12-04	2015-01-17
...	...	...	...	...	...
BIOTECH	Toulouse	N240	3100	2016-08-18	2016-08-31
BIOTECH	Toulouse	N240	3100	2016-08-31	2016-09-16
SURVEYOR	Genève	D107	4100	2016-03-22	2016-08-18
SURVEYOR	Genève	D107	4100	2016-08-18	2016-08-31
SURVEYOR	Genève	D107	4100	2016-08-31	2016-09-16
SURVEYOR	Genève	D107	4100	2016-09-16	2016-10-22
SURVEYOR	Genève	D107	4100	2016-10-22	2016-12-08
SURVEYOR	Grenoble	D107	3800	2016-02-16	2016-03-21
SURVEYOR	Grenoble	D107	3800	2016-03-21	2016-03-22
SURVEYOR	Grenoble	N240	3700	2016-09-16	2016-10-22
...	...	...	...	...	...
SURVEYOR	Toulouse	A68	3700	2016-09-16	2016-10-22

Figure 10.31 - Non-partitioned version of VALUED_STABLE_INTERVAL

This single version of VALUED_STABLE_INTERVAL will allow us to compute all the statistics we have described in the preceding sections.

a) Counting employees

To compute the evolution of the number of employees from each city working on each project, Script 10.55 first counts them[11] in each stable interval, then reduces

11. Since the *select list* of the query includes column `End`, the grouping criterion should be written "`group by PROJECT,CITY,Start,End`" according to the SQL standard. We have omitted the `End` component, which is (illegally) allowed by SQLite (and by MySQL, among others) for performance reason (the execution is 10 to 15% faster).

value-equivalent intervals. Note that expression `count(distinct CODE)` can be more simply written `count(*)`.

```
create temp table H_COUNT as
  select PROJECT,CITY,count(distinct CODE) as Number,Start,End
  from   VALUED_STABLE_INTERVAL
  group by PROJECT,CITY,Start;
function LTemp:reduce {select PROJECT,CITY,Number from H_COUNT};
```

Script 10.55 - Computing the evolution of the number of employees in *each project* and *each city*

```
create temp table H_COUNT as
  select PROJECT,count(distinct CODE) as Number,Start,End
  from   VALUED_STABLE_INTERVAL
  group by PROJECT,Start;
function LTemp:reduce {select PROJECT,Number from H_COUNT};
```

Script 10.56 - Computing the evolution of the number of employees in *each project*

```
create temp table H_COUNT as
  select CITY,count(distinct CODE) as Number,Start,End
  from   VALUED_STABLE_INTERVAL
  group by CITY,Start;
function LTemp:reduce {select CITY,Number from H_COUNT};
```

Script 10.57 - Computing the evolution of the number of employees in *each city*

CITY	Number	Start	End
Genève	1	2014-11-23	2016-03-22
Genève	2	2016-03-22	2016-12-08
Genève	1	2016-12-08	2016-12-29
Genève	2	2016-12-29	2017-08-09
Genève	1	2017-08-09	9999-12-31
Grenoble	1	2016-02-16	2016-03-22
Grenoble	2	2016-09-16	2016-12-08
Grenoble	3	2016-12-08	2016-12-29
Grenoble	2	2016-12-29	9999-12-31
Lille	1	2015-08-22	9999-12-31
Paris	1	2014-12-04	2015-06-15
Paris	2	2015-06-15	2016-10-22
Paris	3	2016-10-22	2017-05-19
Paris	1	2017-05-19	9999-12-31
Toulouse	1	2015-06-09	2016-08-18

```
| Toulouse | 2       | 2016-08-18 | 2016-09-16 |
| Toulouse | 1       | 2016-09-16 | 2016-10-22 |
| Toulouse | 1       | 2017-05-19 | 2017-07-24 |
+----------+---------+------------+------------+
```

Figure 10.32 - Result of Script 10.57: evolution of the number of employees in *each city*

b) Computing average, min and max salaries

The evolution of *average*, *min* and *max* statistics of SALARY values is also derived from the non-partitioned version of VALUED_STABLE_INTERVAL (Scripts 10.58).

```
create temp table H_AVG_SALARY as
  select PROJECT,CITY,
         round(avg(SALARY),1) as Average,
         min(SALARY) as Min,
         max(SALARY) as Max,Start,End
  from   VALUED_STABLE_INTERVAL
  group by PROJECT,CITY,Start;
function LTemp:reduce {select PROJECT,CITY,Average,Min,Max
                       from   H_AVG_SALARY};
```

Script 10.58 - Computing the evolution of the main statistics of SALARY

Producing these statistics per project or per city is quite similar.

c) Computing salary costs

Table H_COST sums and records, for each stable interval, the cost of all the employees who were active in this interval. It is created by the query of Script 10.59.

```
create temp table H_COST as
  select PROJECT,CITY,
         sum(SALARY)/30.42 *
         durationDays(Start,min(End,'$maxDate$'))) as TCost,
         Start,End
  from   VALUED_STABLE_INTERVAL
  group by PROJECT,CITY,Start;
```

Script 10.59 - Computing the salary cost of each stable interval

The evolution of salary cost just requires an appropriate presentation of H_COST (Script 10.60). In Script 10.61, the values of TCost are summed for each interval of each project.

```
select PROJECT,CITY,round(TCost,1) as "TCost",Start,End
from   H_COST
order by PROJECT,CITY,Start;
```

Script 10.60 - Evolution of the salary cost of project and city

```
select PROJECT,round(sum(TCost),1) as "TCost",
       min(Start) as "Start",max(End) as "End"
from   H_COST
group by PROJECT,Start;
```

Script 10.61 - Evolution of the salary cost of each project

By summing the values of TCost for each project and/or each city, we get their total salary cost (Scripts 10.62 and 10.63). The table of the total cost of each project is shown in Figure 10.33.

```
select PROJECT,CITY,int(sum(TCost)) as TotalCost
from   H_COST
group by PROJECT,CITY;
```

Script 10.62 - Computing the total salary cost of each project in each city

```
select PROJECT,int(sum(TCost)) as TotalCost
from   H_COST
group by PROJECT;
```

Script 10.63 - Computing the total salary cost of each project

PROJECT	TotalCost
AGRO-2000	156,917
BIOTECH	280,329
SURVEYOR	243,445

Figure 10.33 - Total cost of each project (Script 10.63)

10.11.6 Efficient computing of VALUED_STABLE_INTERVAL

The three operations that produce the basic VALUED_STABLE_INTERVAL table (through Script 10.48) from which we have computed our statistics can prove too

slow for large temporal table. They can be replaced by function **valuedInterval** of the **LTemp** library, which is both simpler and much faster. The sequence of Script 10.48 can be replaced by Script 10.64, which produces the data of Figure 10.29.

```
function LTemp:valuedInterval
         {select PROJECT,CITY,CODE,SALARY from H_EMPLOYEE
          group by CITY,PROJECT
          into VALUED_STABLE_INTERVAL};
```

Script 10.64 - Computing VALUED_STABLE_INTERVAL through function LTemp:valuedInterval

The gain in execution time results mainly from the way the stable intervals are computed from the EVENT table, through a single scan of this table instead of executing a self-join particularly expensive.

To produce non-partitionned intervals (Scripts 10.52, 10.53, 10.54), we simply omit the `group by` clause (Script 10.65).

```
function LTemp:valuedInterval
         {select PROJECT,CITY,CODE,SALARY from H_EMPLOYEE
          into VALUED_STABLE_INTERVAL};
```

Script 10.65 - Computing VALUED_STABLE_INTERVAL for common intervals through function LTemp:valuedInterval

10.11.7 Temporal data with fixed and unit intervals

In the techniques discussed so far, the stable intervals are naturally derived from the data themselves. Though the concept of stable interval is intuitive, it does not scale easily (Section 10.11.4, d) and sometimes leads to non trivial queries. In addition, it provides synthetic data that are difficult to plot on a timeline due to the uneven distribution of the time points. Hence the idea to extract statistics based on *fixed-length intervals*: seconds, days, months, years, etc.

The algorithm of Script 10.66 generates a table in which each row is assigned to a day within interval [minDate,maxDate). The day is represented by unit interval [Start,End), the length of which is the granularity of the time measure (Figure 10.34).

To generate other fixed-length interval, we replace expression `nextDate(D)` by `addToDate(D,n)`, where **n** is the period. So, `addToDate(D,7)` will provide weekly intervals.

```
create temp table CALENDAR(Start date,End date);
with recursive Dates(Start,End)
as (select '$minDate$',nextDate('$minDate$')
      union all
    select D.End, nextDate(D.End) as Last
    from   Dates D
    where  Last <= '$maxDate$'
   )
insert into CALENDAR select * from Dates;
```

Script 10.66 - Building a daily calendar

This calendar is the ultimate version of STABLE_INTERVAL. Joined with table H_EMPLOYEE (Script 10.67), it provides a version of VALUED_STABLE_INTERVAL in which each row describes one day in the life of one employee (Figure 10.35).[12]

Start	End
2014-11-23	2014-11-24
2014-11-24	2014-11-25
2014-11-25	2014-11-26
2014-11-26	2014-11-27
2014-11-27	2014-11-28
2014-11-28	2014-11-29
2014-11-29	2014-11-30
2014-11-30	2014-12-01
2014-12-01	2014-12-02
...	...

Figure 10.34 - A unit (daily) calendar

From now on, we can ignore temporal concerns in aggregation functions and apply them as we usually do in non-temporal processing. Moreover, the queries will be more regular. The price to pay is that this table may be quite large. In our case study, it comprises 6,110 rows instead of 28 in H_EMPLOYEE and 130 in non-partitioned VALUED_STABLE_INTERVAL.

```
create temp table VALUED_STABLE_INTERVAL as
  select E.CODE,E.NAME,E.PROJECT,E.CITY,E.SALARY,C.Start,C.End
  from   CALENDAR C,H_EMPLOYEE E
  where  C.Start >= E.start
  and    C.End   <= E.end;
```

Script 10.67 - Daily distribution of table H_EMPLOYEE

12. Converting a temporal table according to unit intervals is often called **unfolding**.

PROJECT	CODE	NAME	CITY	SALARY	Start	End
AGRO-2000	G96	Godin	Genève	3900	2016-08-31	2016-09-01
AGRO-2000	G96	Godin	Genève	3900	2016-09-01	2016-09-02
AGRO-2000	G96	Godin	Genève	3900	2016-09-02	2016-09-03
AGRO-2000	G96	Godin	Genève	3900	2016-09-03	2016-09-04
AGRO-2000	G96	Godin	Genève	3900	2016-09-04	2016-09-05
AGRO-2000	G96	Godin	Genève	3900	2016-09-05	2016-09-06
...	...	...	.....	...	...	...
AGRO-2000	G96	Godin	Genève	4100	2017-03-15	2017-03-16
AGRO-2000	G96	Godin	Genève	4100	2017-03-16	2017-03-17
AGRO-2000	G96	Godin	Genève	4100	2017-03-17	2017-03-18
AGRO-2000	G96	Godin	Genève	4100	2017-03-18	2017-03-19
BIOTECH	...	...	.....	...	...	...
...	...	...	.....	...	...	...

Figure 10.35 - Daily history of employees (table VALUED_STABLE_INTERVAL)

Script 10.68 shows that source table H_EMPLOYEE can be rebuilt from VALUED_STABLE_INTERVAL. So both tables contain the same information.

```
function LTemp:project
        {select CODE,NAME,SALARY,CITY,PROJECT
         from   VALUED_STABLE_INTERVAL into H_EMPLOYEE};
```

Script 10.68 - Reconstructing H_EMPLOYEE from VALUED_STABLE_NTERVAL

This version of VALUED_STABLE_INTERVAL can now be used to compute all the statistics of the preceding section.

Computing statistics against larger fixed-length intervals, such as weeks or months may be more complex because these intervals generally do not match those of source data, and therefore are not stable. Indeed, several stable intervals may contribute, often partially, to one fixed-length interval. Starting from *unit valued stable intervals* makes it easier to aggregate data on larger fixed-length intervals. Implementing this idea is left as an (easy) exercise to the reader.

A last remark: aggregation based on unit intervals has been called in the literature *Instantaneous temporal aggregation*.

10.12 Temporal library *LTemp*

We have shown the management of all the temporal data models studied in the first part of this case study and the exploitation of temporal data can be performed through pure SQL queries. However, this may raise two potential problems:

– Some of these SQL queries are fairly complex, as testified by the generalized projection operator (see Script 10.22). In addition, some operations require a sequence of several queries, which does not contribute to the readability of temporal data processing scripts.
– Due to the declarative nature of SQL, the execution of some queries may prove rather slow, particularly for large data sets, depending on the optimization strategies applied by the DBMS.[13] In some cases, the result of these queries can be obtained faster by a simple scan of the temporal table.

The SQLfast library **LTemp** provides a collection of operators the goal of which is to simplify and optimize the processing of temporal data. This library is a Python module that works on the currently opened database. Each function requires a single argument which is a query expressed in a pseudo-SQL language. It returns a numeric code indicating how the execution was carried out.

The table and column names that appear in the argument of the functions must obey the following syntactic rules: strict SQL naming conventions, no SQL reserved words[14], no quoted or bracketed names and no name beginning with an underscore character.

In all the temporal tables referenced in the function argument, the interval columns (*start*, followed by *end*, whatever their real name) must be the last ones of their schema.

At the present time, this library offers five operators, but more may be developed in the future.

a) Temporal projection

This operator executes a generalized projection of a source table on selected columns and stores the result in a target table.

Syntax

```
LTemp:project {<project-query>};
```

Argument

```
<project-query> ::= select <columns>
                    from   <source>
                   [where  <condition>]
                    into   <target>
<source>    ::= table name
<columns>   ::= column list
<condition> ::= SQL condition
```

13. For instance, the restricted version of the projection of Script 10.20 (with one level of subquery) is particularly fast, while the generalized version of Script 10.22 (with two levels of subquery) is much slower. This observation has been done for SQLite v3.28. The conclusion can be different for other versions or other DBMS.
14. Except for start and end for interval columns, courtesy of the benevolence of SQLite!

```
<target>    ::= table name
```

Semantics

Computes the generalized projection of table (or view) <source> on its columns <columns> and stores the result in table <target>. If <condition> is specified, only the rows that satisfy this condition are processed. If table <target> does not exist it is created.

Constraint

- The database containing table <source> must be opened.
- The columns of <columns> may not specify column aliases.
- The from clause may not specify a table alias.

Return code

0: successful execution

1. no database opened

2: syntax error in <project-query>

3: error in source data extraction

4: error in generating the result data

Example

```
function status = LTemp:project {select CITY,PROJECT
                                 from   H_EMPLOYEE
                                 where  Salary > 6000
                                 into H_CITY_PRO};
```

b) Temporal reduction

This operator implements a simplified version of the projection. It reduces the consecutive value-equivalent states in the source table itself.

Syntax

```
LTemp:reduce {<reduce-query>};
```

Argument

```
<reduce-query> ::= select <columns>
                   from   <source>
<source>  ::= table name
<columns> ::= column list
```

Semantics

Reduces the consecutive value-equivalent states in the source table itself. Same effect as LTemp:project where the target table is the source table.

Constraint

- The database containing table <source> must be opened.
- <source> must be a base table, not a view.
- The columns of <columns> may not specify column aliases.
- The from clause may not specify a table alias.

Return code

0: successful execution
1. no database opened
2: syntax error in <project-query>
3: error in source data extraction
4: error in generating the result data

Example

```
function status = LTemp:reduce {select CITY,PROJECT from H_CP};
```

c) Temporal intervals (basic)

This generates the stable intervals of the source table, augmented by additional column values and stores the result in a target table. This operator prepares data for computing aggregates on the additional columns.

Syntax

```
LTemp:valuedInterval {<interval-query>};
```

Argument

```
<interval-query> ::= select <columns>
                      from   <source>
                     [where  <condition>]
                     [group by <groupBy>]
                      into   <target>
<source>    ::= table name
<columns>   ::= column list
<condition> ::= SQL condition
<groupBy>   ::= column list
<target>    ::= table name
```

Example

```
function status = LTemp:valuedInterval
                      {select PROJECT,CITY,SALARY
                       from   H_EMPLOYEE
                       where  SALARY > 6000
                       group by PROJECT,CITY
```

```
                        into VALUED_STABLE_INTERVAL};

function status = LTemp:valuedInterval
                       {select PROJECT,CITY,SALARY
                        from   H_EMPLOYEE
                        into COMMON_VALUED};
```

Semantics

This operator is used as the first step of temporal aggregate functions. It computes the largest intervals in source table <source> (among the rows that satisfy <condition>) in which the values of <groupBy> are constant. Then, it associates with each of these intervals the values of <columns> that hold in the source table in this interval. From this result, that has been stored in table <target>, various temporal aggregate functions can be applied on <columns> for each <groupBy> group. Table <target> is created if it does not exist.

When <groupBy> is absent, the intervals are computed differently. An interval is considered stable throughout table <source> if the values of <columns> (not only of <groupBy> as in the former formula) are constant in this interval.

Constraint

- The database containing table <source> must be opened.
- The columns of <columns> may not specify column aliases.
- The columns of <groupBy> form a subset of those of <columns>.
- The from clause may not specify a table alias.

Return code

0: successful execution

1. no database opened

2: syntax error in <project-query>

3: error in source data extraction

4: error in generating the result data

Example

```
function status = LTemp:valuedInterval
                       {select PROJECT,CITY,SALARY
                        from   H_EMPLOYEE
                        where  SALARY > 6000
                        group by PROJECT,CITY
                        into VALUED_STABLE_INTERVAL};

function status = LTemp:valuedInterval
                       {select PROJECT,CITY,SALARY
                        from   H_EMPLOYEE
                        into COMMON_VALUED};
```

d) Temporal intervals (extended)

Same function as `valuedInterval` with more flexible *select list*. An element of the *select list* can be a column name but also a constant or a scalar SQL expression.

Syntax

```
LTemp:valuedIntervalE {<interval-query>};
```

Argument

```
<interval-query> ::= select <columns>
                      from   <source> <alias>
                     [where  <condition>]
                     [group by <groupBy>]
                      into   <target>
<source>    ::= table name
<exp-list>  ::= list of SQL scalar expressions
                where column names are prefixed
                with <alias> for <source> columns
                and 'S' for Start and End
                from "_STABLE_INTERVAL"
<condition> ::= SQL condition where column names are prefixed
                with <alias>
<groupBy>   ::= column list where column names are prefixed
                with <alias>
<target>    ::= table name
```

Semantics

This operator is used as the first step of temporal aggregate functions. It computes the largest intervals in source table `<source>` (among the rows that satisfy `<condition>`) in which the values of `<groupBy>` are constant. Then, it associates with each of these interval the values of `<exp-list>` that hold in the source table in this interval. Elements of the select list can be a column name, a constant or any expression that evaluates into a single value.

From this result, that has been stored in `<target>`, various temporal aggregate functions can be applied on `<exp-list>` for each `<groupBy>` group.

When `<groupBy>` is absent, the intervals are computed differently. An interval is considered stable throughout table `<source>` if the values of `<exp-list>` are constant in this interval.

Constraint

- The database containing table `<source>` must be opened.
- The column names that appear in `<exp-list>` must be prefixed with `<alias>` except columns Start and End, defining the interval, that must be prefixed by alias `'S'`.
- The columns of `<groupBy>` form a subset of those of `<columns>`.

Return code

0: successful execution

1. no database opened

2: syntax error in `<project-query>`

3: error in source data extraction

4: error in generating the result data

Example

```
function status = LTemp:valuedIntervalsE
                 {select E.PROJECT,E.CITY,E.SALARY*12/365,
                         durationDays(S.Start,S.End) as Dur
                  from   H_EMPLOYEE E
                  where  E.SALARY > 6000
                  group by E.PROJECT, E.CITY
                  into VALUED_STABLE_INTERVAL};
```

e) Temporal normalization

This operator completes a source table by inserting fictitious states made up of *null* values for each missing interval. It is particularly useful to prepare tables for outer joins.

Syntax

```
LTemp:normalize {<normalize-query>};
```

Argument

```
<normalize-query> ::= update <source>
                      [wrt  <compare>]
                       on   <Scolumns> [= <Ccolumns>]
                      [into <target>]
<source> ::= table name
<compare> ::= table name
<Scolumns> ::= column list
<Ccolumns> ::= column list
<target> ::= table name
```

Semantics

If, for each value of `<Scolumns>`, table `<source>` is incomplete and includes gaps, aka *missing intervals*, this operator replaces them by fictitious states comprising the value of `<Scolumns>` + *null* values. So, the history of each value of `<Scolumns>` is now complete. If `<target>` is specified, the completed data are stored in table `<target>`. Otherwise, the new states are inserted in table `<source>` itself. It table `<target>` does not exist, it is created.

If table `<compare>` is specified, `<Ccolumns>` must also be specified. When comparing the history of a value of `<Scolumns>` in table `<source>` with the history of the same value of `<Ccolumns>` in `<compare>`, if the latter history

either starts earlier or ends later, then one or two fictitious states are added to the history of <source> in order to adjust both histories. If this way, the history of <source> does not start after that of <compare> and does not end before that of <compare>, and this, for each value of <Scolumns>. This function is mainly used in the preparation of the arguments of outer joins.

If table <source> is empty, a *null* state is generated for each value of <Ccolumns>. Its interval is the life span of this value in <compare>.

Constraints

- The database containing table <source> must be opened.
- If <target> is specified, <source> can be a view.
- The columns of <columns> may not specify column aliases.
- The from clause may not specify a table alias.

Return code

0: successful execution

1. no database opened

2: syntax error in <project-query>

3: error in source data extraction

4: error in generating the result data

Examples

```
function status = LTemp:normalize {update H_EMP on Proj;

function status = LTemp:normalize {update H_EMP on Proj
                                   into NewEMP;

function status = LTemp:normalize {update H_EMP wrt H_PRO
                                   on Proj = ProID
                                   into NewEMP;
function status = LTemp:normalize {update H_T2 wrt H_T1
                                   on A1,A2 = B1,B2
                                   into NewEMP;
```

10.13 Temporal data format conversion

The first part of this case study describes some of the most popular models of temporal data. The first one, *entity-based*, or more generally *tuple-based*, have been our main support to develop the algorithms of this second part. In this model, all the attributes of an entity are collected to form a single row (or tuple), with which we associate a time interval that specifies the period during which these values were valid, either in the real world (valid time) or in the database (transaction time).

The other formats are briefly described and illustrated in this first part, but we have not developed algorithms to convert temporal data from one model into another one. The reasons are twofold: some conversion rules are obvious (*horizontal splitting* for example) while others require operators that were not yet introduced at that stage.

In this section, we examine the conversion rules of entity-based temporal data to attribute-based, event-based and document-oriented models.

10.13.1 Entity-based to attribute-based history conversion

Each attribute table is obtained through a temporal projection. Script 10.69 generates attribute tables H_SAL, H_CIT and H_PRO from columns SALARY, CITY and PROJECT of table H_EMPLOYEE. *null* states can be deleted if needed.

The source table can be recovered through a simple *temporal join* of the attribute tables if *null* states are preserved. Otherwise, H_EMPLOYEE must be rebuilt through an *outer join*.

```
function LTemp:project
         {select CODE,SALARY  from H_EMPLOYEE into H_SAL};
function LTemp:project
         {select CODE,CITY    from H_EMPLOYEE into H_CIT};
function LTemp:project
         {select CODE,PROJECT from H_EMPLOYEE into H_PRO};
```

Script 10.69 - Projecting of H_EMPLOYEE on each non-PK column

10.13.2 Entity-based to event-based history conversion

While entity(or tuple)-based history represents the successive *states* of each entity, *event-based history* records the *state transitions* induced by data modification operation. Each row comprises a value for each entity attribute plus the nature of the event and the instant this event occurred. Figure 10.36 reminds us what the transaction time history of projects looks like.

TITLE	THEME	BUDGET	Time	Event
AGRO-2000	Crop improvement	65000	21	create
AGRO-2000	Crop improvement	75000	36	update
AGRO-2000	Crop improvement	82000	41	update
BIOTECH	Biotechnology	180000	2	create
BIOTECH	Genetic engineering	160000	7	update
BIOTECH	Genetic engineering	120000	9	update
BIOTECH	Genetic engineering	140000	20	update
BIOTECH	Biotechnology	140000	44	update
SURVEYOR	Satellite monitoring	310000	12	create

```
| SURVEYOR  | Satellite monitoring | 375000 | 18   | update |
| SURVEYOR  | Satellite monitoring | 345000 | 31   | update |
| SURVEYOR  | Satellite monitoring | 345000 | 40   | delete |
+-----------+----------------------+--------+------+--------+
```

Figure 10.36 - Event-based history of projects

The conversion of entity-based to event-based history relies on the following rules, stated for a definite project:

- *create*: this event is identified by the *first state* of the project. Its **Time** value is that of **start** of the row of this state.
- *delete*: this event corresponds to the *last state* of the project, provided the value of **end** of this state is not *infinite future*. Its **Time** value is that of **end** of the row of this state.
- *update*: this event is defined by a state of the project that is not the first one. Its **Time** value is that of **start** of the row of this state.

We observe that the last state of a deleted entity generates two events: an *update* or *create* event and a *delete* event.

Script 10.70 translates these rules to create event-based table EH_PROJECT. The creation of table EH_EMPLOYEE is quite similar.

```
create table EH_PROJECT as
   select TITLE,THEME,BUDGET,start as Time,'create' as Event
   from   H_PROJECT P
   where not exists(select *
                    from  H_PROJECT
                    where TITLE = P.TITLE and end = P.start)
      union
   select TITLE,THEME,BUDGET,end as Time,'delete' as Event
   from   H_PROJECT P
   where not exists(select *
                    from  H_PROJECT
                    where TITLE = P.TITLE and start = P.end)
   and   end <> $future$
      union
   select TITLE,THEME,BUDGET,start as Time,'update' as Event
   from   H_PROJECT P
   where  exists(   select *
                    from  H_PROJECT
                    where TITLE = P.TITLE and end = P.start);
```

Script 10.70 - PROJECT: from entity-based to event-based

Let us now examine how each event of EH_PROJECT can be translated into entity-based state.

- *create*: we create a state of the project (the first one) with the values of TITLE, THEME and BUDGET of the event row. Its **start** value is that of **Time** of the row of this event. If this event is followed by a next event on this project, then the **Time** value of the latter is assigned to the **end** value of the state in construction. Otherwise, if it is the only event of the project, then the state is current and its **end** value is set to infinite future.
- *update*: we create a state of the project. The values of columns TITLE, THEME, BUDGET, **start** and **end** are assigned with the same rules as those of the *create* event.
- *delete*: this event was just an additional by-product of the analysis of entity states. Its information has already been taken into account when examining the other two event rows. We can ignore it.

The algorithm of Script 10.71 generates entity-based view VEH_PROJECT from table EH_PROJECT. It proceeds in two steps: generation of past states and generation of current states.

- A *past state* is generated by two successive events of the project, the first one being a *create* or *update* event. Two events are successive if there is no other event between them.
- A *current state* is generated by a create or update event row that is not followed by another event of the same project.

We observe that *delete* events rows have not been used in the conversion process.

```
create view VEH_PROJECT(TITLE,THEME,BUDGET,start,end) as
  -- Generate past states --
  select P1.TITLE,P1.THEME,P1.BUDGET,P1.Time,P2.Time
  from   EH_PROJECT P1,EH_PROJECT P2
  where  P1.TITLE = P2.TITLE and P1.Time < P2.Time
  and    P1.oper in ('create','update')
  and    not exists(select * from EH_PROJECT
                    where  TITLE = P1.TITLE
                    and    Time > P1.Time and Time < P2.Time)
     union
  -- Generate current states --
  select P.TITLE,P.THEME,P.BUDGET,P.Time,$future$
  from   EH_PROJECT P
  where  P.oper in ('create','update')
  and    not exists(select * from EH_PROJECT
                    where  TITLE = P.TITLE
                    and    Time > P.Time);
```

Script 10.71 - PROJECT: from event-based to entity-based

10.13.3 Entity-based to document-oriented history conversion

We will describe a method that produces data according to a light format using SQLfast multilist data structures. This format is close to that described in Section 9.9.5 and illustrated by Figure 9.18 but easier to generate and to decode. Deriving from it the algorithm that generates pure JSON variant is immediate.

The Projects value of employee A237 is as follows (we work on the transaction time version for simplicity):

```
AGRO-2000:24-30,33-999999;BIOTECH:30-33
```

It is structured as a 4-level hierarchy:

> ***level 1***: the value of Projects is a list of *value-intervals* components, (here, `AGRO-2000:24-30,33-999999` and `BIOTECH:30-33`), separated by symbols `';'`;
>
> ***level 2***: a *value-interval* component is a list of two elements, separated by symbol `':'`; the first element is the *value* of a project title (e.g., `AGRO-2000`) and the second one provides the *intervals* associated with this value;
>
> ***level 3***: the *intervals* are structured as a list of intervals, separated by symbols `','`;
>
> ***level 4***: an *interval* is a list of two time points, separated by symbol `'-'`.

We will derive this structure from the attribute-based model generated by projecting H_EMPLOYEE on each column. Through Script 10.69, we get tables H_SAL, H_CIT and H_PRO. For simplicity, we ignore constant column NAME.

Then, we compute, for each distinct value of each column for each employee, the list of its intervals (Script 10.72 and Figure 10.37 for column PROJECT).

```
create table HH_PRO as
select CODE,PROJECT,group_concat(start||'-'||end,',') as Projects
from H_PRO group by CODE,PROJECT;
```

Script 10.72 - Grouping intervals for each PROJECT value of each employee entity

CODE	PROJECT	Projects
A237	AGRO-2000	24-30,33-999999
A237	BIOTECH	30-33
A68	BIOTECH	8-13
A68	SURVEYOR	13-39
C45	AGRO-2000	22-28
C45	BIOTECH	11-22,28-999999
D107	BIOTECH	35-999999
D107	SURVEYOR	15-35
D122	AGRO-2000	37-42
D122	BIOTECH	10-16
D122	SURVEYOR	16-37

```
| G96   | AGRO-2000 | 23-999999        |
| G96   | BIOTECH   | 3-23             |
| M158  | BIOTECH   | 4-14,34-999999   |
| M158  | SURVEYOR  | 14-34            |
| N240  | AGRO-2000 | 32-43            |
| N240  | BIOTECH   | 19-25            |
| N240  | SURVEYOR  | 25-32            |
+------+-----------+-----------------+
```

Figure 10.37 - Project history of each employee entity

Finally, we collect these values together with their associated intervals for each CODE value (Script 10.73 and Figure 10.38).

```
select CODE,group_concat(PROJECT||':'||Projects,';') as Projects
from   HH_PRO group by CODE;
```

Script 10.73 - Associating their value-intervals pairs with each CODE value

```
+------+-----------------------------------------------+
| CODE | Projects                                      |
+------+-----------------------------------------------+
| A237 | AGRO-2000:24-30,33-999999;BIOTECH:30-33       |
| A68  | BIOTECH:8-13;SURVEYOR:13-39]                  |
| C45  | AGRO-2000:22-28;BIOTECH:11-22,28-999999       |
| D107 | BIOTECH:35-999999;SURVEYOR:15-35]             |
| D122 | AGRO-2000:37-42;BIOTECH:10-16;SURVEYOR:16-37  |
| G96  | AGRO-2000:23-999999;BIOTECH:3-23              |
| M158 | BIOTECH:4-14,34-999999;SURVEYOR:14-34         |
| N240 | AGRO-2000:32-43;BIOTECH:19-25;SURVEYOR:25-32  |
+------+-----------------------------------------------+
```

Figure 10.38 - Document-oriented presentation of the projects of employee entities

The final document-based table is obtained by joining this table with those built in a similar way on SALARY and CITY. By integrating these small scripts into a single SQL query, we get the final algorithm of Script 10.74.

Rebuilding the source data from this presentation is of course quite possible. However, doing this with SQL queries only would be fairly complex. Considering the 4-level structure of the data, the most obvious approach would be a procedure comprising embedded loops, each decoding a level of the data structure. This is what Script 10.75 implements for column PROJECT.

– The main loop reads each row of DOC_EMPLOYEE, extracting the values of CODE (in variable cod) and Projects (in variable pro) of one employee.

– The next inner loop extracts successive *value-intervals* component of Projects (in variable project).

– The most inner loop extract from this component each individual interval (in variable interval).

– This interval is finally split into start and end values.

```
create table DOC_EMPLOYEE(CODE,Salaries,Cities,Projects);
with
   HH_SAL(CODE,SALARY,Salaries)
   as (select CODE,SALARY,group_concat(start||'-'||end,',')
       from   H_SAL group by CODE,SALARY),
   HH_CIT(CODE,CITY,Cities)
   as (select CODE,CITY,group_concat(start||'-'||end,',')
       from   H_CIT group by CODE,CITY),
   HH_PRO(CODE,PROJECT,Projects)
   as (select CODE,PROJECT,group_concat(start||'-'||end,',')
       from   H_PRO group by CODE,PROJECT)
insert into DOC_EMPLOYEE
select sal.CODE,sal.Salaries,cit.Cities,pro.Projects
from (select CODE,group_concat(SALARY || ':'||Salaries,';')
                  as Salaries
      from   HH_SAL group by CODE) sal,
     (select CODE,group_concat(CITY || ':'||Cities,';')
                  as Cities
      from   HH_CIT group by CODE) cit,
     (select CODE,group_concat(PROJECT || ':'||Projects,';')
                  as Projects
      from   HH_PRO group by CODE) pro
where sal.CODE = cit.CODE and sal.CODE = pro.CODE;
```

Script 10.74 - Converting the contents of H_EMPLOYEE into the document-oriented model

A simple join (or outer join, depending on whether source columns were nullable or not) will rebuild the original H_EMPLOYEE table.

```
create table H_PRO(CODE char(5),PROJECT char(20),
                   start integer,end integer);
for cod,pro = [select CODE,Projects from DOC_EMPLOYEE];
   for project = [item(;) $pro$];
      compute value = item('$project$',1,':');
      compute intervals = item('$project$',2,':');
      for interval = [item(,) $intervals$];
         compute start = item('$interval$',1,'-');
         compute end   = item('$interval$',2,'-');
         insert into H_PRO(CODE,PROJECT,start,end)
                     values ('$cod$','$value$',$start$,$end$);
      endfor;
   endfor;
endfor;
```

Script 10.75 - Rebuilding table H_PRO from document-oriented data

10.14 Performance analysis and optimization

TDB.db, the database we have used so far, was fine to elaborate temporal data models and to develop querying and transforming algorithms but it is not at all suitable for measuring the performances of these algorithms. To this aim, we will use a series of experimental temporal databases with increasing size. They all include a unique table, H_EMPLOYEE, with the same schema as that of TDB.db. They are named TDB-xxxx.db, where xxxx, ranging from 100 to 200000, is the number of employees the states of which they contain. They include about 10 states per employee (as a reminder, database TDB.db describes the 28 states of 8 employees). These eleven valid time databases can be downloaded from the SQLfast website (in Section *Technical complements*).

These databases are derived from the employees.db database available at https://github.com/siara-cc/employee_db. The latter was itself indirectly translated from an XML dataset developed by Fusheng Wang and Carlo Zaniolo in 2003. This dataset is available at http://timecenter.cs.aau.dk/software.htm.

To make the source database comply with our TDB.db database, we have applied the following transformations:

- The data are integrated into a single table H_EMPLOYEE
- Rows with invalid intervals are deleted
- Employee rows where properties (title, salary, etc.) are missing are deleted
- A one-letter prefix is added to emp_id to produce column CODE
- Columns first_name and last_name columns are merged to form column NAME
- Column title is converted into column PROJECT
- Column department is converted into column CITY
- Yearly salary is converted into monthly SALARY
- SALARY values are rounded to a multiple of 5
- This rounding produces consecutive value-equivalent states; they are reduced
- Symbolic future '9999-01-01' is changed to '9999-12-31'

In the following sections, we measure the execution time of the algorithms that implement the main temporal operators for each of the experimental databases. We also show that some of these algorithms can be dramatically improved by carefully selected indexes while for others, seemingly promissing indexes just slow them down.

It should not be forgotten that the performance figures depend on the optimization strategies of the DBMS. Different conclusions could be drawn with other DBMS or with variants of the SQL queries developed in this case study.

Scalability of the algorithms

A major concern, when we develop an algorithm, is the way it behaves when it processes larger data sets. This issue is commonly described by the *time complexity* of the algorithm, that is (said very simply), by the shape of the function that relates the running time against the size of the problem, generally noted N, here the number of employees.[15] If applying an operator on 10,000 employees takes 10 seconds, how much time will the algorithm require to process 20,000 employees? The best we can expect (in a reasonable world) is that this time will double to about 20 seconds. If this proves exact, then our algorithm will be said to have a *linear complexity*, noted O(N). If we are unlucky, its complexity could be O(N.logN), or even worse $O(N^2)$. We will pay attention to this aspect in the analysis of the algorithms of the temporal operators

10.14.1 Temporal projection

We have identified two patterns of SQL temporal projection operators (called predicative techniques), namely *entity-based projection* (Script 10.20) and *generalized projection* (10.22). The first one is devoted to the frequent, but limited case in which the projection columns include the components of the entity primary key, such as CODE for H_EMPLOYEE and TITLE for H_PROJECT. The second one does not impose any constraint on the composition of the projection columns and can be used to solve any pattern of projection.

Besides the SQL implementation of the projection operator, a *procedural algorithm* is available through function LTemp:project (Script 10.24).

Of the three algorithms, the *generalized predicative technique*, despite its elegance, is quickly disqualified from the other two algorithms: the projection on CITY,PROJECT of H_EMPLOYEE with 1,000 employees cost 9 seconds, compared to 16 milliseconds for the procedural technique (more than 560 times slower). This technique could be used, if desired, for datasets not larger than 200 employees, if a cost of 2.23 seconds is considered acceptable. So, we discard this technique to concentrate on entity-based predicative and procedural techniques. The execution times of these techniques, for experimental databases from 1,000 to 50,000 employees (about 10,000 to 500,000 states) are shown in the table of Figure 10.39 and in the graph of Figure 10.40.

These figures show that,

- both techniques are reasonable fast,
- the procedural technique is 40% faster than the predicative technique,
- the evolution of the execution time against the database size is linear: if we double the size, we just double the time. This is natural for the procedural algorithm, which merely scans the data, but it means that the SQL optimizer was able to generate a similar query plan despite the complexity of the SQL query.

15. See https://en.wikipedia.org/wiki/Time_complexity

	predicative	procedural
1K	0.031	0.015
2K	0.062	0.047
5K	0.141	0.093
10K	0.297	0.203
20K	0.594	0.437
50K	1.515	1.078

Figure 10.39 - Computing entity-based projection H_EMPLOYEE[CODE,CITY]

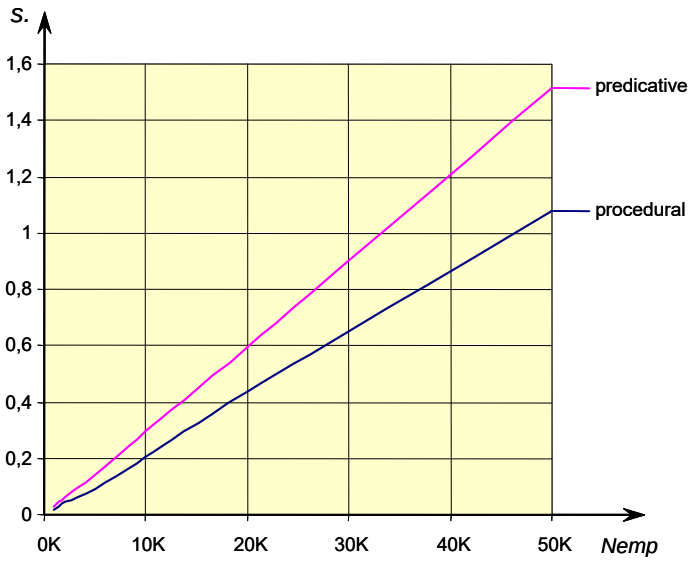


Figure 10.40 - Graphical representation of the execution time of the projection H_EMPLOYEE[CODE,CITY]

10.14.2 Temporal inner join

Joining tables requires at least two tables (unless one accepts to perform self joins only!) such as H_EMPLOYEE and H_PROJECT of database TDB.db. To get realistic performance figures we will prepare two joinable tables, H_EMP1 and H_EMP2, in databases TDB-100 to TDB-50000. The contents of H_EMP1 is generated by the projection of H_EMPLOYEE on columns CODE, NAME, CITY while H_EMP2 results from the projection on CODE, SALARY, PROJECT:

```
function LTemp:project {select CODE,NAME,CITY
                        from H_EMPLOYEE into H_EMP1};
```

```
function LTemp:project {select CODE,SALARY,PROJECT
                        from H_EMPLOYEE into H_EMP2};
```

CODE is the primary key (and therefore a unique index) in both tables. On each experimental database, table H_EMP1 contains about 1.1 state per employee (properties *name* and *city* are fairly stable) and H_EMP2 contains about 10 states per employee. Joining these tables on entity primary key CODE provides the time scores of Figure 10.41.

	H_TEMP1 * H_EMP2
1K	0.094
2K	0.109
5K	0.172
10K	0.266
20K	0.485
50K	0.969

Figure 10.41 - Joining tables H_EMP1 and EMP2

These results show that the execution times follow a linear rule fairly similar to that of procedural projection (Figure 10.40).

10.14.3 Temporal outer join

For this experiment, we will use tables H_EMP1 and H_EMP2 built to study inner join. In the largest, H_EMP2, we delete about 10% of the states, randomly selected, with the following query:

```
set step = 19;
with OID(oid) as
  (select random_i(0,$step$)
   union all
   select oid + random_i(1,$step$)
   from OID where oid < $max$ )
delete from H_EMP2 where rowid in (select oid from OID)
```

The left outer join of H_EMP1 with H_EMP2 proceeds in two steps: replacing the missing states in H_EMP2 with null states (the *normalization* step) then applying an inner join between H_EMP1 and H_EMP2 so completed. We have proposed two normalization techniques, namely predicative (Script 10.31) and procedural (Script 10.35).

The table of Figure 10.42 and the graph of Figure 10.43 show the execution times of the *normalization* step of H_EMP2 *wrt* H_EMP1 for databases TDB-1000 to TDB-50000.

	Predicative	Procedural
1K	0.219	0.047
2K	0.297	0.078
5K	0.657	0.141
10K	1.125	0.266
20K	2.141	0.547
50K	5.047	1.375

Figure 10.42 - Outer join preparation: normalizing incomplete table H_TEMP2

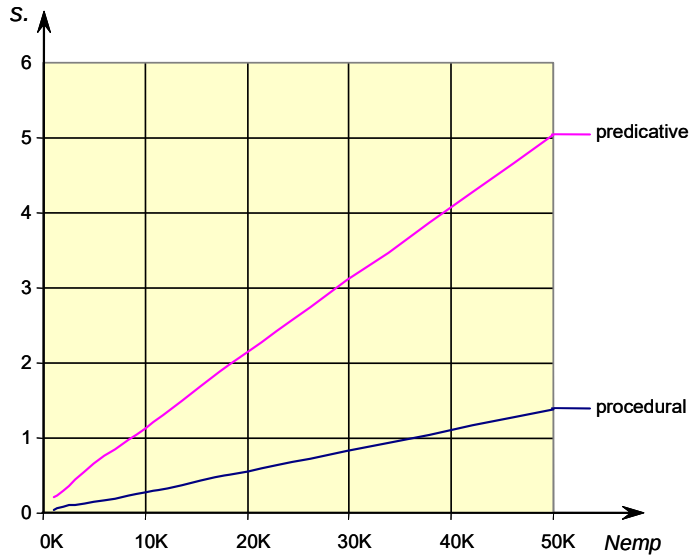


Figure 10.43 - Graphical representation of the execution time of the normalization of H_EMP2

Once again, the normalization time follows a linear rule for both techniques. The execution time of the whole outer join process is therefore also linear. Both techniques provide reasonable times for small to medium dataset sizes. For large datasets, the procedural technique clearly is recommended.

10.14.4 Aggregation

Temporal aggregation consists in applying standard **group by** queries on a special table, VALUED_STABLE_INTERVAL (STABLE_EMPLOYEE is just a variant of it), derived from the source table. Due to their different nature, we consider these two steps separately.

Step 1: computing VALUED_STABLE_INTERVAL

This operator comprises three main operations. We will examine each of them to analyze its behavior and to check if, and how, they can be improved.

Computing the events of each project (table EVENT)

The algorithm we propose computes the events as the union of the sets of start and end values of H_EMPLOYEE:

```
select PROJECT,start as Instant from H_EMPLOYEE
   union
select PROJECT,end as Instant from H_EMPLOYEE;
```

Its cost accounts for **15**% of the lowest total computing time. We observe that this algorithms (probably) will scan table H_EMPLOYEE twice and that duplicate elimination requires a preliminary sorting of the dates. To avoid these operations, we could propose the following version:

```
select PROJECT,start as Instant from H_EMPLOYEE
   union all
select PROJECT,max(End) as Instant from H_EMPLOYEE;
```

It is based on the fact that the sets of values of start and end are the same, except for the lowest within the set of start values and the highest within the set of end values. The gain is quite interesting: **0.7** s. against **3.2** s. for the initial version, these values being observed on the largest database (Nemp = 200,000). Unfortunately, these algorithms are not equivalent. The former is valid even if the history includes gaps, while the latter requires that the history be complete.

Therefore, we keep the initial version of the algorithm, which is more general.

Computing the stable intervals of each project (table STABLE_INTERVAL)

The stable intervals are computed through a self-join of table EVENT, which associate with each Instant value the next value in EVENT, except for the last one:

```
select PROJECT,
       Instant as Start,
       (select min(Instant) from EVENT
        where  PROJECT= E.PROJECT
        and    Instant > E.Instant) as End
from   EVENT E where End is not null;
```

So far, we have defined no index on EVENT. Let us create one on (PROJECT, Instant), that should speed up the join operation:

```
create index XEVENT on EVENT(PROJECT,Instant);
```

The effect of this index is particularly dramatic: while table STABLE_INTERVAL is created in **98** s. without index, the latter performs this creation in no more than **0.078** s. (index creation included), that is **0.08**% of the initial cost (observed for Nemp = 10,000)!

We naturally add this index.

Computing the valued stable intervals (table VALUED_STABLE_INTERVAL)

This operation consists in adding to each stable interval a set of values extracted from H_EMPLOYEE. This is done through a simple inner join:

```
select S.PROJECT,E.SALARY,S.start,S.end
from   STABLE_INTERVAL S, H_EMPLOYEE E
where  E.PROJECT = S.PROJECT
and    E.start <= S.start and S.start < E.end;
```

H_EMPLOYEE has a primary key made up of columns CODE and start, which means that a unique index has been created on these columns. As to STABLE_NTERVAL it has so far been given no index. Let create this one, which, hopefully, may support the join operator:

```
create index XSI on STABLE_INTERVAL(PROJECT,Start);
```

Once again this index has a decisive effect: table VALUED_STABLE_INTERVAL is created in **93.7** s. without index and in **12.7** s. with this index (index creation included), that is **13.6**% of the initial cost (also observed for Nemp = 10,000).

We suggest to keep this index.

The global process

Now, let us consider the cost of the whole operation. The table of Figure 10.44 shows the contribution of each subprocess for different configurations of the indexes. Label **pred (a+b)** indicates that index `EVENT(PROJECT,Instant)` is created (a=1) or not (a=0) and that index `STABLE_INTERVAL(PROJECT,Start)` is created (b=1) or not. The last row gives the execution time of the procedural algorithm as provided by function LTemp:valuedInterval. These times have been collected for the TDB-10000 database (10,000 employes).

Two important observations:

1. The gain provided by the indexes is very important, the execution time falling from 191.9 s. to 13 s., that is, **6.8**% of the initial time, or, more impressive, an improvement of **93.2**%.
2. The optimized predicative and procedural algorithms have the same cost, with a slight advantage for the predicative algorithm. This shows that native SQL queries may prove as efficient (and often more efficient) than procedural techniques.[16]

	EVENT	STABLE_INTERVAL	VALUED_ STABLE_INTERVAL	Total
size	28,017	28,011	30,677,511	
pred (0+0)	0.19	98.00	93.70	191.9
pred (0+1)	0.19	98.00	12.70	110.9
pred (1+0)	0.19	0.078	93.70	93.9
pred (1+1)	0.19	0.078	12.70	13.0
procedural				13.4

Figure 10.44 - Detailed cost for Nemp = 10,000 employees (100,755 states)

3. The optimized predicative algorithm generates the valued stable intervals at an impressive rate of 2.36 million states per second.

The five predicative and procedural algorithmic variants described above provide the results shown in the table of Figure 10.45.

	predicative				procedural
	0+0	0+1	1+0	1+1	
1K	12.3	9.9	3.0	0.6	0.6
2K	35.9	27.5	9.5	1.6	1.7
5K	102.8	71.1	35,4	5.9	6.1
10K	191.9	110.9	93.9	13.0	13.4
20K	350.9	160.3	218.7	28.4	28.6
50K	765.9	242.1	601.7	69.9	72.8

Figure 10.45 - Execution time (in seconds) of the main variants of algorithms for increasing values of Nemp

The figures are reported in the graph of Figure 10.46, which shows the trend of the running time of the algorithms against the increasing values of Ntemp. This trend suggests that the fastest implementations of the operator run in quasi-linear time, which is very good news.[17]

The choice between (optimized) predicative and procedural techniques is a matter of taste since their performances are fairly equivalent.

16. Such an observation has also been made for case study *Conway's Game of Life*, where careful SQL query tuning allowed the computing time to drop from 838 s. to just 0.6 s.
17. For TDB-200000: 300 s. for 656 million valued intervals (*predicative 1+1* technique).

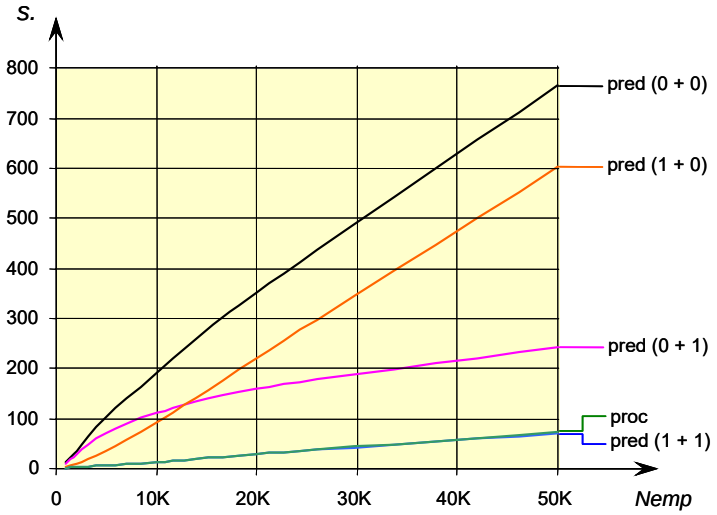


Figure 10.46 - Computing the valued stable intervals: evolution of the running time

Step 2: data aggregation

Now, let us examine the last part of temporal aggregate queries. Once the valued stable states have been extracted, computing the statistics is expressed through simple and straightforward **group-by** SQL queries that seem to leave little room for improvement.

A popular thumb rule states that an index on the **group-by** criterion of frequent aggregate queries is likely to speed up their execution. So, let us create an index on STABLE_EMPLOYEE(PROJECT,Start) to count employees per project or VALUED _STABLE_INTERVAL(PROJECT,Start) for the other statistics. It should make the computing of statistics faster since it is built on the aggregation criterion.

Actually, it does not, as Figure 10.47, that reports on the times of the **count** function, makes it clear. Column **count** gives the execution times when no index has been created on STABLE_EMPLOYEE. Column **(index)+count** gives the times for the same function when an index has been created (index creation time **not included**). Column **index** provides the index creation times. Even if we ignore the latter time, computing the **count** statistics in a table devoid from index is far faster than computing in an indexed table.

The trend is similar for the other statistics. So, we abandon the idea.

We might be surprised by the length of these processes. We must consider that they are carried out on very large tables. Table STABLE_EMPLOYEE of TDB-10000.db database comprises about 31 million rows while TDB-50000.db generates 163 million of them.

	count	(index)+count	index
1K	1.09	1.41	1.27
2K	3.74	4.67	4.24
5K	15.10	17.63	16.14
10K	34.60	40.61	37.32
20K	75.65	89.93	83.07
50K	204.44	305.80	288.25

Figure 10.47 - Can an index on STABLE_EMPLOYEE help when counting employees?

10.15 Temporal data management in current DBMS

Introducing temporal features in SQL standard took quite a long time, nearly three decades. Since the first scientific publications in the eighties (Allen's relations were published in 1983), hundreds of proposals have been published. They were devoted to the semantics of time (what does such terms as interval, time point, now, past and future mean; temporal calculus and algebra), to its historical, geographic and cultural aspects, to temporal languages for time-dependent data and to efficient implementation of temporal operators in DBMS. SQL:2011 was the first standard to introduce SQL extensions to represent and manipulate temporal data. Though these extensions are incomplete, they make modeling and manipulation of temporal data easier and more reliable. Perhaps as important, they also pave the way to a better understanding of the temporal data paradigm by practitionners. From these proposals, most major DBMS have introduced all or a part of the temporal concepts of SQL:2011. In the rest of this section, we describe these concepts and compare them to the material presented of this case study.

SQL:2011 introduce the core concept of *period*. *Period* is not a data type but a new schema object defined in the table declaration. It is built from two time values (dates or timestamps) and is given a user name. In a *valid time table*, these time values are controlled by the applications and form an *application-time period*.

In a *transaction time table* (called *system-versioned table*) the period is a *system-time period* with standard name SYSTEM_TIME. Its components are time-valued columns that are managed by the system.

An SQL:2011 table can be bitemporal, that is, with both valid time and transaction time dimensions.

10.15.1 Application-time temporal table

In this case study, both valid time and transaction time data describe the evolution of a collection of entities and of their associations. Evolution events always apply on their current states. There is no future state and past states cannot be updated.

The valid time data model of SQL:2011 is more general in that the values of both *start* and *end* temporal columns are under the responsibility of the application. This means, in particular, that

- the application is free to set both start and end times of a state,
- future events can be coped with, therefore generating future states,
- missing intervals (gaps) are allowed
- value-equivalent intervals may overlap,
- in some cases, even non value-equivalent intervals may overlap, thus specifying that an entity is in more than one state; however, the standard seems to consider this pattern as not recommended and provides for a way to reject it,
- (whole or part of) states can be modified and deleted.

The temporal dimension of a valid time table is specified by an application-controlled period, here below named Vperiod:

```
create table H_EMPLOYEE(
   CODE     char(5)  not null,
   NAME     char(10) not null,
   SALARY   integer  not null,
   CITY     char(10) not null,
   PROJECT  char(20) not null,
   Vstart   date     not null,
   Vend     date     not null,
   period for Vperiod (Vstart,Vend));
```

If a primary key is to be specified, it comprises some non temporal columns (here CODE) + the valid time period:

```
   primary key (CODE,Vperiod)
```

To avoid problematic data patterns, we add a constraint enforcing the uniqueness of rows on CODE values *at every time point*, that is, in every snapshot of the table

```
   primary key (CODE,Vperiod without overlaps)
```

Referential integrity is expressed through a foreign key comprising some non temporal columns (here PROJECT) + the valid time period.

```
   foreign key (PROJECT,period Vperiod)
           references H_PROJECT (TITLE,period Vperiod)
```

Any state of temporal data can be modified, be it current, past or future. We consider this state of project BIOTECH:

TITLE	THEME	BUDGET	start	end
...	...	...	...	...
BIOTECH	Biotechnology	180000	2014-11-18	2015-02-27
...	...	...	...	...

It tells that, from 2014-11-18 to 2015-02-27, the budget of this project was 180,000. Now, we are told that during January of 2015, this budget actually was 150,000. So, we translate this information as follows:

```
update H_PROJECT
for portion of Vperiod from '2015-01-01' to '2015-01-31'
set BUDGET = 150000
where TITLE = 'BIOTECH';
```

The former state is replaced by these new three states:

TITLE	THEME	BUDGET	start	end
...	...	...	...	...
BIOTECH	Biotechnology	180000	2014-11-18	2015-01-01
BIOTECH	Biotechnology	150000	2015-01-01	2015-01-31
BIOTECH	Biotechnology	180000	2015-01-31	2015-02-27
...	...	...	...	...

Similarly, we can delete a state or a portion of a state:

```
delete H_PROJECT
for portion of Vperiod from '2015-01-01' to '2015-01-31'
where TITLE = 'BIOTECH';
```

A valid time table can be queried as a standard table, with conditions on any columns, including the temporal columns. Extracting the current state of all the employees is straightforward:

```
select CODE,NAME,Vstart
from   H_EMPLOYEE
where  Vend = '9999-12-31';
```

It can also be queried through temporal predicates based on the period of the table (they are derived from, but not identical to, Allen's relations). This query extract a snapshot of all the employees at a definite date:

```
select CODE,NAME,Vstart
from   H_EMPLOYEE
where  Vperiod contains '2016-12-31'
```

This one extract the states falling in a time slice (year 2016):

```
select CODE,NAME,Vstart
from   H_EMPLOYEE
where  Vperiod overlaps period ('2016-01-01','2016-12-31');
```

10.15.2 System-versioned temporal table

The transaction time data model of SQL:2011 is very close to that developed in this case study[18]. A system-versioned table (*transaction time table*) comprises two timestamp columns that are set and modified by the DBMS alone. They are used to define system period SYSTEM_TIME. Primary and foreign keys are defined on the current state without reference to the temporal components of the table:

```
create table H_EMPLOYEE(
   CODE    char(5)  not null,
   ...,
   PROJECT  char(20) not null,
   Sstart timestamp  not null generated always at row start,
   Send   timestamp  not null generated always at row end,
   period for SYSTEM_TIME (Sstart,Send),
   primary key (CODE),
   foreign key (PROJECT) references H_PROJECT(TITLE))
   with system versioning;
```

Update and delete queries are those of non temporal tables:

```
update H_EMPLOYEE
set PROJECT = 'BIOTECH'
where CODE = 'A237';

delete H_EMPLOYEE
where CODE = 'A237';
```

Querying a system-versioned table follows a specific syntax that is different from that of application-time tables. The current states is extracted as follows:

```
select CODE,NAME,Sstart,Send
from H_EMPLOYEE
where ...
```

18. The main difference is that the system clock must provide sufficient precision to prevent two independent transactions to get the same timestamp. A precision of a millisecond was not considered sufficient to enforce this property, hence the need of the CLOCK table.

And this is how we extract a snapshot:

```
select CODE,NAME
from H_EMPLOYEE
for SYSTEM_TIME as of '2016-12-31'
where PROJECT = 'BIOTECH';
```

A time slice can be specified by a *closed-open* interval:

```
select CODE,NAME,Sstart,Send
from H_EMPLOYEE
for SYSTEM_TIME from '2016-01-01' to '2017-01-01'
where ...
```

... or by a *closed-closed* interval:

```
select CODE,NAME,Sstart
from H_EMPLOYEE
for SYSTEM_TIME between '2016-01-01' and '2016-12-31'
where ...
```

10.15.3 Future directions[19]

The SQL:2011 standard specifies the temporal data model and language extensions to formulate elementary queries. However it does not address more complex queries such as inner and outer joins, projection (and its coalescing derivative),and aggregate functions. Their expression is the responsibility of application programmers. While the translation of an inner join is quite straightforward, thanks to function overlaps, the other complex queries require much effort that prevents programmers from implementing them, as shown in this case study.

10.16 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory SQLfast/Scripts/Case-Studies/Case_Temporal_DB. Actually, they can be run from main script **TDB-MAIN.sql**, that displays the selection box of Figure 10.48.

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to let them develop their own applications.

19. This section relies on the eponym section of reference [Kulkarni, 2012].

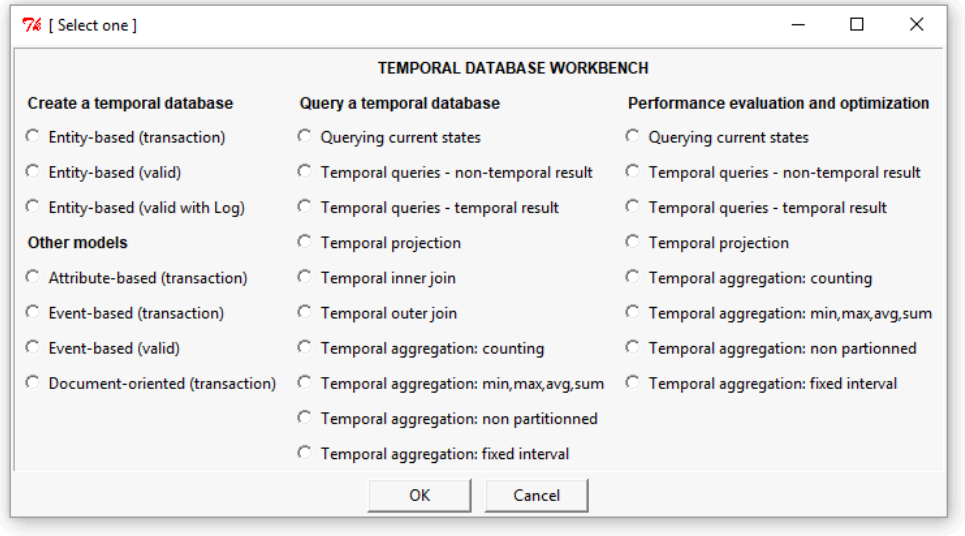


Figure 10.48 - The scripts of Part 1 and Part 2 can be run from this main panel

10.17 References

Among the many hundreds of references on temporal data management, we have chosen three of the most practical ones. They all are available on the web as free pdf documents.

Snodgrass, R., T., *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann, 2000.

This large book (528 pages) explores in much detail the concept of time and its multiple aspects. It describes various ways to model temporal data and develops SQL queries and triggers implementing temporal data management primitives (applied to base tables, not to views). Then, it analyzes and translates in SQL:1999 the main queries as well as temporal projection and joins. Finally, it describes an incremental method to build temporal databases. This text may be difficult to read and to apply practically since all the developments are based on an unnormalized temporal model, where an entity history may include gaps, overlapping states and successive value-equivalent states. This model is a generalization of the entity-based model described in this study but it makes the definition of such basic concepts as primary and foreign keys, as well as the fundamental algorithms fairly complex. This book does not address aggregation functions nor performance issues. A good reading to consolidate and extend the knowledge acquired in this case study. Also shows what lies under the hood of modern SQL:2011 DBMS.

Jensen, C., J., and Snodgrass, R., T. (Ed.), *Temporal Database Entries for the Springer Encyclopedia of Database Systems*, Springer, 2008.

A collection of 81 entries describing most of the standard and advanced concepts in temporal data management published in Springer Encyclopedia of Database Systems. Most of these entries have been been retained in the 2018 edition of this encyclopedia.

http://www.cs.arizona.edu/people/rts/sql3.html

Kulkarni, K., Michels, J., E., *Temporal features in SQL:2011*, SIGMOD Record, September 2012 (Vol. 41, No. 3)

This article describes some of the concepts of temporal databases that have been introduced in standard SQL:2011.

Finally, the reader can find in these slideshows (November 2015) an alternative view of temporal databases:

https://www.slideshare.net/torp42/temporal-databases-data-models
https://www.slideshare.net/torp42/temporal-databases-queries

Oracle (from version 12c)

valid and transaction time

```
period for <valid-time-column>[<interval>]
<interval>: (start column,end-column)]
```

`valid-time-column`: hidden system column

`start column,end-column:` date or timestamp regular columns, nullable

with [] clause: valid time

without [] clause: transaction time

```
create table H_EMPLOYEE(...,
       Tstart,
       Tend,
       period for Tvalid (Tstart,Tend));

select * from H_EMPLOYEE as of period
            for tvalid to_date('01-Jun-10','DD-MON-YYYY')
where CODE = 'www';

select * from H_EMPLOYEE versions period for Tvalid
            between to_date('01-Jun-10','DD-MON-YYYY')
                and to_date('02-Jun-10','DD-MON-YYYY')
where CODE = 'www';
```

N.B: It's worth keeping in mind this new functionality is a query transformation.

db2 11

system time and business time

```
create table H_EMPLOYEE(...,
       Tstart date not null,
       Tend   date not null,
       period business_time (Tstart,Tend));

select * from H_EMPLOYEE for business_time
         as of '2008-06-01';
select * from H_EMPLOYEE for business_time
         from '2008-06-01' to '2008-06-30';
select * from H_EMPLOYEE for business_time
         between '2008-06-01' and '2008-12-31';
```

N.B. Also by query transformation

Variant of index that checks that value-equivalent overlapping states exist

It seems that temporal join must programmed explicitly (https://stackoverflow.com/questions/50985111/business-time-with-several-joins-temporal-logic-in-db2)

SQL Server

Only transaction time

```
create table H_EMPLOYEE(...,
       Tstart datetime2(2) generated always as row start,
       Tend   datetime2(2) generated always as row end,
       period for system_time (Tstart,Tend));
```

Querying: same as db2 (system_time instead of business_time)

Case study **11**

Kings of France - Part 1

Objective. This study describes the French royal dynasty since Hughes Capet in 941. Its underlying goal is to study some properties and algorithms of widespread tree data structures. This first document of a series of two analyzes the dynasty of Kings of France, stores it in a database and extracts some simple information from it. The next study will be devoted to the derivation of more complex information.

Keywords. genealogy, tree, cyclic data structure, interval, ordering relation, temporal query, de Morgan law.

11.1 Introduction

Kings and queens generally have complicated lives, full of wars, plots, crimes, betrayal, family affairs, political alliances, unhappy marriages, mistresses, lovers, bastard and hidden children. Just to begin with.

The genealogy of Kings of France is one of the most turbulent, and therefore interesting lineages. Fortunately, the *Salic law*[1], edicted by King Clovis I in the 6th century, to which the Frankish government system was submitted, introduced important constraints in the royal inheritance principles that will substantially simplify the representation of French royal lineages in a database. One of the titles

1. http://en.wikipedia.org/wiki/Salic_law

(*rules*) of this law, the *agnatic succession*, organizes the succession of monarchs. In particular, it strictly excludes women and favours *father-son* or *brother-brother* succession, whenever possible.[2]

This means that, during nearly a millenary, the Kings of France are organized as a simple **tree**, a structure particularly straightforward to translate into data structures. It is clear that, in this organization, women would have made things much more complicated. Algorithmically speaking, I mean.

Figure 11.1 shows the members of the dynasty of French monarchs originated from **Hughes Capet** (941-996), the founder of the Capetian house. His youngest descendant, **Louis-Philippe I**, reigned until 1848.

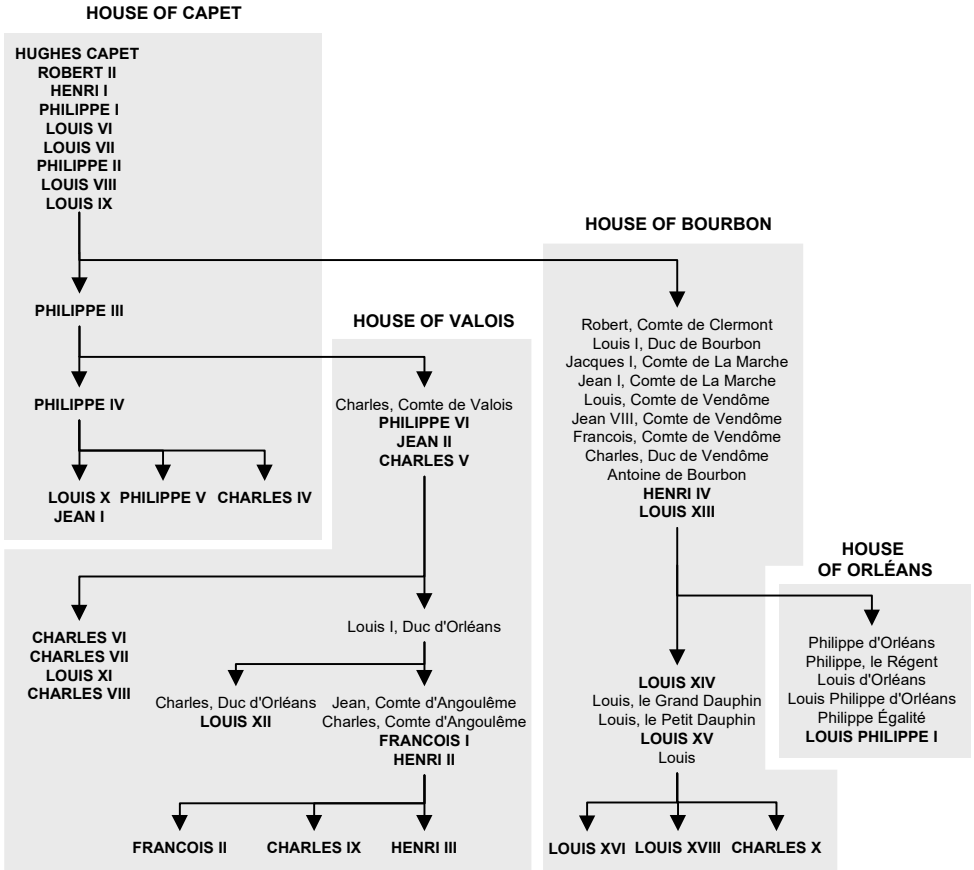


Figure 11.1 - Genealogy of Kings of France from 987 to 1848. King names are in bold upper case (*translated from http://en.wikipedia.org/wiki/French_monarchs_family_tree*)

2. The Salic agnatic succession rule still applies in several countries, notably in Belgium, where it was partly repealed in 1991 to allow female descendants of Albert II to ascend the throne (http://fr.wikipedia.org/wiki/Roi_des_Belges).

This main part of French monarchy spanned 9 centuries, thus providing a nice set of data to play with.

Note that this tree represents *kings descent*. It includes kings of course (in boldface and in upper case) but also descendants of kings who, though not having been kings themselves, have kings in their descent. OK, this seems to be a bit complex, so let us examine an example. **Charles**, Comte de Valois, was not king. However, he was the son of **Philippe III**, King of France and his son, **Philippe VI**, also was King of France. So, as a node in the link between two kings, **Charles** appears in the royal tree to ensure the continuity of this tree. This explains why kings of France seem to have so few sons![3]

11.2 Data structures

These data are structured as shown in Script 11.1. The data are stored in table **MEMBER** of database **KINGS.db**.

```
createOrReplaceDB KINGS.db;
create table MEMBER(
   PiD            char(6) not null,
   Name           char(20) not null,
   House          char(12) not null,
   Father         char(6),
   Title          char(20) not null,
   DateBirth      integer not null,
   DateDeath      integer not null,
   DateReignBegin integer,
   DateReignEnd   integer,
   primary key (PiD),
   foreign key (Father) references MEMBER);
commitDB;
closeDB;
```

Script 11.1 - Schema of the KINGS database

For each member, we record the following information:

- **PiD**, his unique code,
- **Name**, his usual name,
- **House**, the name of the house (family) he belongs to,

3. The site http://www.capedia.fr/ records the descendants of Hughes Capet, the number of which is estimated to 550,000. Most of them were not kings. Fortunately.

- **Father**, the code of his father (optional),
- **Title**, his main title; special value `'Roi de France'` (*King of France*) indicates that he reigned during a part of his life,
- **DateBirth**, (estimated) year of birth,
- **DateDeath**, (estimated) year of death,
- **DateReignBegin**, first year of his reign (optional),
- **DateReignEnd**, last year of his reign (optional).

This table is self-referencing: it includes a foreign key (Father) referencing itself, making it a *cyclic structure*. Since the father of Hughes Capet, the root of the royal tree, is not recorded, the column of this foreign key is nullable.

The loading of sample data is shown in Script 11.2. Excerpts from the database (column names have been shortened to save space) are shown below. Document Table-MEMBER.pdf, in the script directory shows the full contents of this table.

PiD	Name	House	Fath	Title	Birth	Death	RBegin	REnd
C00	HUGHES CAPET	Capet	--	Roi de France	939	996	987	996
C01	ROBERT II	Capet	C00	Roi de France	972	1031	996	1031
C02	HENRI I	Capet	C01	Roi de France	1008	1060	1031	1060
C03	PHILIPPE I	Capet	C02	Roi de France	1052	1108	1060	1108
C04	LOUIS VI	Capet	C03	Roi de France	1077	1137	1108	1137
C05	LOUIS VII	Capet	C04	Roi de France	1120	1180	1137	1180
C06	PHILIPPE II	Capet	C05	Roi de France	1165	1223	1180	1223
C07	LOUIS VIII	Capet	C06	Roi de France	1187	1226	1223	1226
C08	LOUIS IX	Capet	C07	Roi de France	1215	1270	1226	1270
C09	PHILIPPE III	Capet	C08	Roi de France	1245	1285	1270	1285
C10	PHILIPPE IV	Capet	C09	Roi de France	1268	1314	1285	1314
C11	LOUIS X	Capet	C10	Roi de France	1289	1316	1314	1316
C12	PHILIPPE V	Capet	C10	Roi de France	1291	1322	1316	1322
C13	CHARLES IV	Capet	C10	Roi de France	1295	1328	1322	1328
C14	JEAN I	Capet	C11	Roi de France	1316	1316	1316	1316
...	...	...	...	...	...	...	...	...
V03	CHARLES V	Valois	V02	Roi de France	1337	1380	1364	1380
V04	CHARLES VI	Valois	V03	Roi de France	1368	1422	1380	1422
V05	CHARLES VII	Valois	V04	Roi de France	1403	1461	1422	1461
V06	LOUIS XI	Valois	V05	Roi de France	1423	1483	1461	1483
V07	CHARLES VIII	Valois	V06	Roi de France	1470	1498	1483	1498
V08	Louis I	Valois	V03	Duc d'Orléans	1372	1407	--	--
V09	Charles	Valois	V08	Duc d'Orléans	1394	1465	--	--
...	...	...	...	...	...	...	...	...
V13	FRANCOIS I	Valois	V12	Roi de France	1494	1547	1515	1547
V14	HENRI II	Valois	V13	Roi de France	1519	1559	1547	1559
V15	FRANCOIS II	Valois	V14	Roi de France	1544	1560	1559	1560
V16	CHARLES IX	Valois	V14	Roi de France	1550	1574	1560	1574
V17	HENRI III	Valois	V14	Roi de France	1551	1589	1574	1589
...	...	...	...	...	...	...	...	...

Figure 11.2 - Some rows of the MEMBER table

Since we will do a lot of work on kings, it can be useful to denote them through view KING (Script 11.3).

The data have been collected from various sources, including,

- http://en.wikipedia.org/wiki/French_monarchs_family_tree
- http://en.wikipedia.org/wiki/Members_of_the_French_Royal_Families
- http://en.wikipedia.org/wiki/List_of_French_monarchs

```
createOrReplaceDB KINGS.db;
insert into MEMBER values('C00','HUGHES CAPET','Capet',
       null,'Roi de France',939,996,987,996);
insert into MEMBER values('C01','ROBERT II','Capet',
       'C00','Roi de France',972,1031,996,1031);
insert into MEMBER values('C02','HENRI I','Capet',
       'C01','Roi de France',1008,1060,1031,1060);
...
insert into MEMBER values('O04','Philippe Égalité','Orléans',
       'O03','Duc d''Orléans',1747,1793,null,null);
insert into MEMBER values('O05','LOUIS PHILIPPE I','Orléans',
       'O04','Roi de France',1773,1850,1830,1848);
commitDB;
closeDB;
```

Script 11.2 - Loading data about Kings of France

```
create view KING(PiD,Name,House,Father,
       DateBirth,DateDeath,DateReignBegin,DateReignEnd) as
select PiD,Name,House,Father,
       DateBirth,DateDeath,DateReignBegin,DateReignEnd
from   MEMBER
where  Title = 'Roi de France';
```

Script 11.3 - A view showing kings only

11.3 Warming up: some easy queries

Let us begin with a query *counting the kings provided by each royal house* (Script 11.4). The result set (Figure 11.3) tells us that the *Capetian house* gave the largest number of kings and that the *house of Orléans* provided only one.

House	Kings
Capet	15
Valois	13
Bourbon	7
Orléans	1

Figure 11.3 - Number of kings in each royal house

```
select House,count(*) as Kings from KING
group by House
order by Kings desc;
```

Script 11.4 - How many kings in each house?

The next query returns the average age of kings when they ascend the throne (Script 11.5). Note the use of the **round** function to avoid meaningless decimals.

```
select round(avg(DateReignBegin - DateBirth),1)
       as  'Beginning reign age'
from   KING;
```

Script 11.5 - How old (on the average) were kings when ascending the throne?

```
+---------------------+
| Beginning reign age |
+---------------------+
| 24.9                |
+---------------------+
```

Figure 11.4 - Average age of kings when ascending the throne

The next query extracts the minimum, average and maximum age of death of kings as well as their minimum, average and maximum reign length (Script 11.6). The life span of a member is given by expression `DateDeath - DateBirth` and his reign length by `DateReignEnd - DateReignBegin`. Since these rather long expressions are used several times in the query, we suggest to replace them by shorter names `life` and `reign`, defined by the first two **set** statements.

```
set life = DateDeath - DateBirth;
set reign = DateReignEnd - DateReignBegin;
select min($life$)           as MinLife,
       round(avg($life$),1)  as AvgLife,
       max($life$)           as MaxLife,
       min($reign$)          as MinReign,
       round(avg($reign$),1) as AvgReign,
       max($reign$)          as MaxReign
from   KING
where  DateBirth < DateDeath;
```

Script 11.6 - How long do kings live and reign?

Condition `DateBirth < DateDeath` may appear surprising. Actually, the royal lineage includes a singularity, King *Jean I*. This misfortunate king was born on November 16th, 1316, and died ... five days later. During his short life, he was king of France, so that `DateDeath = DateBirth = DateReignEnd = DateReignBegin`. We decide to discard him from the statistics. We get the results of Figure 11.5, showing that king is not always a safe job.

MinLife	AvgLife	MaxLife	MinReign	AvgReign	MaxReign
16	49.9	79	1	24.0	72

Figure 11.5 - Statistics on the life of kings

11.4 Rebuilding royal families

Through this series of queries, we will collect information on the local family structures of the members: just fathers and sons. The first query associates with each father, the list of his sons (Script 11.7). Expression `group_concat(PiD,',')` creates the comma-separated list of **PiD** values for each group.

```
select Father,group_concat(PiD,',') as Sons
from   MEMBER
group by Father;
```

Script 11.7 - Associating his sons with each father

As the root of the royal tree, Hughes Capet (C00) has no father but nevertheless appears as a son (of nobody) in the result (Figure 11.6).

Father	Sons
--	C00
B00	B01
...	...
B10	B11,O00
B11	B12
...	...
B15	B16,B17,B18
...	...
C08	C09,B00
C09	C10,V00
C10	C11,C12,C13
C11	C14
O00	O01

```
| ...     | ...          |
| V00     | V01          |
| ...     | ...          |
| V03     | V04,V08      |
| ...     | ...          |
| V08     | V09,V11      |
| ...     | ...          |
| V14     | V15,V16,V17  |
+---------+--------------+
```

Figure 11.6 - Fathers and sons

A variant of this query is shown in Script 11.8. It examines each member (not only fathers) and associates with each of them the list if his sons, which may be empty. This query does not make use of row grouping (`group by`) in the main query but builds the list of son's PiD through a subquery. Not surprisingly, Hughes Capet does not appear as a son in the result set.

```
select PiD, (select group_concat(PiD,',')
             from   MEMBER
             where  Father = M.PiD) as Sons
from   MEMBER M;
```

Script 11.8 - Associating his sons with each member

Showing member id's only, these results are not particularly expressive. It would be better to use names instead. However several members appear to have the same name, as shown by the following checking query:

```
select Name, count(*) as N
from   MEMBER group by Name having N > 1;
```

To solve the ambiguity, we write the *name* and *birth date* of both the father and their sons. For instance, we display `'HUGH CAPET (939)'` instead of `'C00'`. So, we replace

```
PiD
```

by[4]

```
Name||' ('||cast(DateBirth as char)||')'
```

Since this fairly complicated expression appears twice in the query, we define variable **`birth`** with value `' ('||cast(DateBirth as char)||')'`. This makes the query simpler to write and to read (Script 11.9, as an extension of Script 11.7).

4. Note that here, `Father` is the alias of the first item in the select list and not the name of the eponym source column. Not all DBMS allow this pattern. If needed, the column alias can be quoted ("Father").

```
set birth = ' ('||cast(DateBirth as char)||')';
select (select Name||$birth$ from MEMBER
        where PiD = M.Father) as Father,
        group_concat(Name||$birth$,',') as Sons
from   MEMBER M
group by Father;
```

Script 11.9 - Showing the *father-son* family composition

The result looks much better (Figure 11.7).

Father	Sons
--	HUGHES CAPET (939)
Antoine (1518)	HENRI IV (1553)
CHARLES V (1337)	CHARLES VI (1368),Louis I (1372)
CHARLES VI (1368)	CHARLES VII (1403)
CHARLES VII (1403)	LOUIS XI (1423)
Charles (1270)	PHILIPPE VI (1293)
Charles (1394)	LOUIS XII (1462)
Charles (1459)	FRANCOIS I (1494)
Charles (1489)	Antoine (1518)
FRANCOIS I (1494)	HENRI II (1519)
François (1470)	Charles (1489)
HENRI I (1008)	PHILIPPE I (1052)
HENRI II (1519)	FRANCOIS II (1544),CHARLES IX (1550),HENRI III (1551)
...	...

Figure 11.7 - Fathers and sons - A more readable presentation

If we are interested by brotherhood only, that is, by groups of at least two brothers, we could try the query of Script 11.10, which returns the list of Figure 11.8.

Brothers
LOUIS XIV (1638),Philippe I (1640)
LOUIS XVI (1754),LOUIS XVIII (1755),CHARLES X (1757)
PHILIPPE III (1245),Robert (1256)
PHILIPPE IV (1268),Charles (1270)
LOUIS X (1289),PHILIPPE V (1291),CHARLES IV (1295)
CHARLES VI (1368),Louis I (1372)
Charles (1394),Jean (1399)
FRANCOIS II (1544),CHARLES IX (1550),HENRI III (1551)

Figure 11.8 - Brotherhood

```
select group_concat(Name||$birth$,',') as Brothers
from   MEMBER
group by Father
having count(*) > 1;
```

Script 11.10 - Brotherhood

The last query of this series is a bit more *sensitive*: it asks the age of members when they had their first son[5] (Script 11.11). The query performs an auto-join that associates two members, the second one (alias S) being the son of the former (alias F).

```
select F.Name||' ('||cast(F.DateBirth as char)||')' as Father,
       count(*) as Sons,
       min(S.DateBirth - F.DateBirth) as AgeFirstSon
from   MEMBER F, MEMBER S
where  F.PiD = S.Father
group by F.PiD,F.Name
order by AgeFirstSon;
```

Script 11.11 - How old were kings when they had their first son?

The result is quite interesting (Figure 11.9)!

Father	Sons	AgeFirstSon
JEAN II (1319)	1	18
François (1470)	1	19
LOUIS XV (1710)	1	19
CHARLES VII (1403)	1	20
Louis (1661)	1	21
PHILIPPE IV (1268)	3	21
...	..	..
LOUIS XI (1423)	1	47
HENRI IV (1553)	1	48
Louis (1376)	1	50
Jean (1399)	1	60
Charles (1394)	1	68

Figure 11.9 - Age of members when they had their first son

5. As known by the database, which records the members of the royal tree only.

11.5 King succession

The database records the *father-son* tree of French members, but does not inform on the succession of kings as they ascend the throne of France. Or does it?

We know for sure that no two kings may reign at the same time. In addition, any gap between two successive reign periods must be avoided, as far as possible. More precisely,

1. Reign periods (DateReignBegin,DateReignEnd) may not overlap.
2. For each reign period, but the last one, there should be another reign period that begins when the former finishes. However, the French revolution forced a break in the succession, so that the rule must be adapted: *for each reign period, but the last one, there should be another reign period that does not begin before the former finishes.*

There are several ways to extract the king succession from table MEMBER. The simplest consists in displaying the kings in ascending order of DateReignBegin:

```
select * from KING
order by DateReignBegin;
```

This works fine except for King *Jean I*, who has the same reign beginning date as his successor, King *Philippe V.* Who succeeds whom is undefined when we only know the year of these dates. Hence the final version of Script 11.12 which discards abnormal periods.

```
select * from KING
where  DateReignBegin < DateReignEnd
order by DateReignBegin;
```

Script 11.12 - The king succession

11.6 Seeking missing kings

King history may include **breaks**, that is, periods during which no king has been reigning, for some reason (a revolution is a good one). Surprisingly, identifying these breaks is not an easy task. Indeed, we must extract from the database *something that does not exist*!

Following Figure 11.10, a royal period is missing (we will call it **K2**) if there are two periods **K1** and **K3** such that:

- **K1** and **K3** are royal reigns (we use view KING),

- **K1** finishes before **K3** starts (`K1.DateReignEnd < K3.DateReignBegin`), so, there is a gap between them,
- there is no period **K2** between **K1** and **K3** (this gap is empty). Should **K2** exist, it would be a royal period, it would begin at the earliest when **K1** terminates and it would finish at the latest when **K3** begins. We must describe **K2** in this this way, then tell that such period do not exist.



Figure 11.10 - Identifying the absence of a period

Script 11.13 translates this definition. It displays king names in **K1** and **K3** surrounding the missing period, as shown in Figure 11.11.

```
+-----------+------+------+-------------+
| Last king | From | To   | Next king   |
+-----------+------+------+-------------+
| LOUIS XVI | 1792 | 1814 | LOUIS XVIII |
+-----------+------+------+-------------+
```

Figure 11.11 - A period without king has been detected

There is only one missing period, from **1792**, when **Louis XVI** was suspended before being executed, to **1814**, when **Louis XVIII**, called by the senate, accesses the throne for a period called *Restauration*.[6]

```
select K1.Name           as 'Last king',
       K1.DateReignEnd   as 'From',
       K3.DateReignBegin as 'To',
       K3.Name           as 'Next king'
from   KING K1, KING K3
where  K1.DateReignEnd < K3.DateReignBegin
and not exists(select 1 from KING K2
               where  K2.DateReignEnd <= K3.DateReignBegin
               and    K1.DateReignEnd <= K2.DateReignBegin);
```

Script 11.13 - Finding missing kings

6. This period exhibits an interesting property. Indeed, it has been interrupted for 100 days, during Napoleon's return to France in 1815. Though this cannot be represented in our database (where dates are measured in years), the fact that several non contiguous periods can be covered by the same king means that kings and royal periods no longer are synonyms. This may make some queries and algorithms of this chapter invalid. Good exercise for the reader!

11.7 Royal conflicts

When studying king succession, we have admitted that *reign periods may not overlap*. However, kings do not always comply with such a wise rule. It is not uncommon that, in troubled times, two kings claim their legitimacy at the same time on the same country. Let us check whether the King of France lineage is devoid of such conflicts.

Checking whether two periods overlap is not as intuitive as we could think. For this, we first examine a simpler problem, that is, determining whether *two periods are disjoint*. Two periods [b1,e1] and [b2,e2] are disjoint if one of them terminates before (or when) the other starts, that is, when either

$$e1 \leq b2$$

or

$$e2 \leq b1$$

So, on the contrary, *two periods overlap* when this condition is not satisfied, that is, when[7]

$$e1 > b2 \text{ and } e2 > b1$$

We translate this condition into Script 11.14, which seeks royal conflicts (overlapping royal periods).

The result is quite comforting. It shows that French kings can been wise:

```
+------------+-------------+
| First king | Second king |
+------------+-------------+
+------------+-------------+
```

```
select K1.Name as 'First king',K2.Name as 'Second king'
from   KING K1, KING K2
where  K1.PiD <> K2.PiD
and    K1.DateReignEnd > K2.DateReignBegin
and    K2.DateReignEnd > K1.DateReignBegin
order by K1.Name,K2.Name;
```

Script 11.14 - Identifying overlapping periods

By artificially changing some values of DateReignBegin, we can create overlapping periods. Just to check the validity of the query!

7. Simple application of a *de Morgan* law: **not** (C1 **or** C2) = (**not** C1) **and** (**not** C2).

The scripts of this study are available in directory **SQLfast/Scripts/Case-Studies/Case_Kings_of_France**. They can be run from main script **Kings-MAIN.sql** (see Part 2).

Case study **12**

Kings of France - Part 2

Objective. In this chapter, we continue the exploitation of the KINGS database through more advanced tree processing applications, based notably on recursive scripts. The central concept from which most of these applications will derive is the transitive closure of table BRANCH, which comprises all direct and indirect ancestor/descendant couples. From it, we will build queries that count the descendants of a member, others that display the hierarchy of these descendants in various graphical way and a transitive reduction query that recovers the contents of table MEMBER from its closure. The last application, tree projection, extracts from table MEMBER a subset in which only kings appear.

Keywords. genealogy, tree, cyclic data structure, transitive closure, transitive reduction, tree projection, recursive CTE, recursive query, tree drawing, tree traversal, depth-first traversal, breadth-first traversal, SQLdraw.

12.1 The *ancestor/descendant* relationships

We will first build a very simple data set that tells us who is a descendant of whom. We know from the base data that ROBERT II is a descendant of HUGUES CAPET (actually his son) and that HENRI I is a descendant of ROBERT II. It is up to us to infer that HENRI I is a descendant of HUGUES CAPET (actually his grandson). We

can go on by identifying all the descendants of HUGUES CAPET, then those of ROBERT II, and so on (Figure 12.1).

The full set of these ancestor/descendant relationships is called the **transitive closure** of table MEMBER. More precisely, we should call it *the transitive closure of the set of (Father,PiD) couples extracted from MEMBER table*[1]. Drawing a transitive closure may prove particularly painful, even for a small initial set of couples. For instance, that of MEMBER comprises 850 couples!

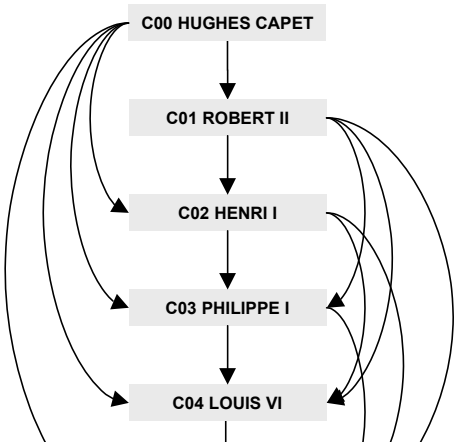


Figure 12.1 - Transitive closure: *who is a descendant of whom*?

Ancestor	Descendant
B00	B01
B00	B02
...	...
C00	C01
C00	C02
C00	C03
C00	C04
...	...
C01	C02
C01	C03
C01	C04
...	...
C02	C03
C02	C04
...	...
V14	V17

Figure 12.2 - The transitive closure of table MEMBER (excerpts)

1. In graph theory, the *transitive closure* **T** of a *binary relation* **B** (that is, a set of couples) is the set defined as follows:(1) it comprises all the couples of **B**, (2) if couples (a,b) and (b,c) are in **T**, then couple (a,c) also is in **T**.

It would be nice to *automatically* generate all these ancestor/descendant relationships from the base data recorded in table MEMBER. The result would look like the data table of Figure 12.2.

This is a good job for a recursive query, or, more precisely, a recursive CTE. In Script 12.1, CTE **PATHS** is defined by two queries:

1. the **initialization query**, that stores the couples (Father,PiD) of the members, except of those who have no father:

```
select Father,PiD from MEMBER where Father is not null
```

2. the **recursive query**, that adds to the CTE an additional subset, defined by joining the current contents of the CTE with the (Father,PiD) couples from table MEMBER:

```
select P.Ancestor, M.PiD
from   PATHS P,MEMBER M
where  P.Descendant = M.Father
```

To make the query easy to read, we have stored the definition of the CTE in variable Closure. The main query of the CTE prints all the couples of the closure.

```
set Closure = select Father,PiD from MEMBER
              where  Father is not null
                 union all
              select P.Ancestor, M.PiD
              from   PATHS P,MEMBER M
              where  P.Descendant = M.Father;

with recursive PATHS(Ancestor,Descendant) as ($Closure$)
select Ancestor,Descendant
from   PATHS
order by Ancestor,Descendant;
```

Script 12.1 - Generating the closure of table MEMBER

Despite its simplicity, the closure is a powerful concept that can be used to solve in a simple way a great variety of problems.

Before going on, we will refine it a little bit by adding two derived pieces of information: the list of members forming the **path** between the ancestor and the descendant of each couple of the closure, and the **length** of this path. For instance, the path of couple (C01,C04) is .C01.C02.C03.C04. and its length is 3.

The extended version is shown in Figure 12.3 and is generated by Script 12.2. The path is built with the technique elaborated in Chapter 19 of the Tutorial, each Id being surrounded by commas.

Ancestor	Descendant	Len	Path
B00	B01	1	.B00.B01.
B00	B02	2	.B00.B01.B02.
...	...	...	...
C00	C01	1	.C00.C01.
C00	C02	2	.C00.C01.C02.
C00	C03	3	.C00.C01.C02.C03.
C00	C04	4	.C00.C01.C02.C03.C04.
...	...	...	...
C01	C02	1	.C01.C02.
C01	C03	2	.C01.C02.C03.
C01	C04	3	.C01.C02.C03.C04.
...	...	...	...
C02	C03	1	.C02.C03.C04.
C02	C04	2	.C02.C03.C04.
...	...	...	...
V14	V17	1	.V14.V17.

Figure 12.3 - The extended transitive closure of table MEMBER (excerpts)

```
set Closure = select Father,PiD,1,'.'||Father||'.'||PiD||'.'
              from   MEMBER where Father is not null
                 union all
              select P.Ancestor,M.PiD,P.Len+1,
                     P.Path||M.PiD||'.'
              from   CTE_PATHS P,MEMBER M
              where  P.Descendant = M.Father;
with recursive
   PATHS(Ancestor,Descendant,Len,Path) as ($Closure$)
select Ancestor,Descendant,Len,Path
from   PATHS order by Len,Ancestor,Descendant;
```

Script 12.2 - Extending the closure of table MEMBER - CTE-based

Since we will use this CTE in several queries, it would be a good idea to transform it into a real SQL view.[2] We change the name of the CTE for CTE_PATHS so that the view can be named PATHS (Script 12.3).

To illustrate the exploitation of the closure, we suggest three application queries. Script 12.4 checks if a member is an ancestor of another one.

2. Or in a temporary table with appropriate indexes, if run time matters.

```
set Closure = <cfr. Script 12.2>;

create temporary view PATHS as
   with recursive
      CTE_PATHS(Ancestor,Descendant,Len,Path) as ($Closure$)
   select Ancestor,Descendant,Len,Path
   from   CTE_PATHS;
```

Script 12.3 - Transformation of the CTE into an actual SQL view

```
ask A,B = Ancestor:||Descendant:;
extract C = select count(*) from PATHS
            where  Ancestor = '$A$' and Descendant = '$B$';
if ($C$ = 1) showMessage $A$ is an ancestor of $B$;
```

```
ask A,B = Ancestor:||Descendant:;
select case when count(*) = 1
            then 'Yes, $A$ is an ancestor of $B$'
            else 'No, $A$ is not an ancestor of $B$'
       end as "Is $A$ an ancestor of $B$?"
from   PATHS
where  Ancestor = '$A$' and Descendant = '$B$';
```

Script 12.4 - Two scripts that check if member A is an ancestor of member B

Script 12.5 displays all the grandfather/grandson couples from the hierarchy.

```
select Ancestor,Descendant from PATHS where Len = 2;
```

Script 12.5 - Grandfather-grandson relationships

Finally, Script 12.6 checks whether member B is between member A and member C in the same branch without being any of them. The last condition examines whether the intermediate member (B) appears in the path from A to C. Both the first and the last characters (dots) are excluded from the search to prevent B from being found at the head or at the end of the path.

We could also check whether some paths include A, B and C in this order:

```
path like '%.$A$.%.$B$.%.$C$.%'
```

```
ask A,B,C = A:|B:|C:;
extract N = select count(*) from PATHS
            where  Ancestor = '$A$' and Descendant = '$C$'
            and    Path like '.%.$B$.%.';
write $N$;
```

Script 12.6 - Are members A, B and C in the same branch?

12.2 Loop-based computing of the transitive closure

Loop-based algorithms can be less elegant than recursive queries but they allows more complex processing on each row of the resultset. They may also be preferred by programmers less familiar with recursive CTE.

Table PATHS, that will contain the paths to build, is initialized with the *Father-Son* couples extracted from table MEMBER. The next set of paths, built by a join between table PATHS and MEMBER, will be first stored in working table LAST, then moved table PATHS. Tables PATHS and LAST have the same structure:

```
PATHS(Ancestor,Descendant,Length,Path);
LAST (Ancestor,Descendant,Length,Path);
```

However, this join is not applied to the full contents of PATHS, but only to the subset just added by the previous iteration (otherwise, the loop would run forever). This is controlled by variable length: initialization stores paths of length 1, the first iteration of the loop adds paths of length 2, the next iteration adds paths the length 3, and so on until all the *Father-Son* couples of table MEMBER have been exhausted, that is, when table LAST is empty.

Based on these principles, we build Script 12.7, which extracts the same data as Script 12.2. It is worth noticing that this iterative algorithm is quite similar to the internal mechanism of recursive CTE, though likely to be less efficient.

12.3 Counting descendants

Through view PATHS created by Script 12.3 one can easily count the descendants of each member:

```
select Ancestor, count(*) as Descent
from   PATHS
group by Ancestor
order by Descent desc, Ancestor;
```

```
insert into PATHS select Father,PiD,1,
                         '.'||Father||'.'||PiD||'.'
                  from   MEMBER where Father is not null;
for length = [1,99];
   insert into LAST
   select P.Ancestor, M.PiD, $length$+1, P.Path||'.'||M.PiD
   from   PATHS P, MEMBER M
   where  P.Descendant = M.Father
   and    P.Length = $length$;
   extract lastN = select count(*) from LAST;
      if ($lastN$ = 0) exit;
   insert into PATHS select * from LAST;
   delete from LAST;
endfor;
```

Script 12.7 - Extending the closure of table MEMBER - Loop-based

To provide a complete answer, we also have to add, through the **union** operator, members who have no descendants:

```
   select PiD as Ancestor, 0 as Descent
   from   MEMBER
   where  PiD not in (select Ancestor from PATHS)
```

This query (lines [2-9] in Script 12.8) returns a minimalist two column resultset that is not particular attractive. We will improve it by adding the name of the member and by inserting the data into a nice sentence format like that of Figure 12.4.

```
+-----------------------------------------------------------------+
| [C00] HUGHES CAPET, King of France, has 35 royal descendants    |
| [C01] ROBERT II, King of France, has 34 royal descendants       |
| [C02] HENRI I, King of France, has 33 royal descendants         |
| [C03] PHILIPPE I, King of France, has 32 royal descendants      |
| ...                                                             |
| [V02] JEAN II, King of France, has 11 royal descendants         |
| [V03] CHARLES V, King of France, has 10 royal descendants       |
| [B09] HENRI IV, King of France, has 7 royal descendants         |
| [B10] LOUIS XIII, King of France, has 6 royal descendants       |
| [B11] LOUIS XIV, King of France, has 4 royal descendants        |
| [C10] PHILIPPE IV, King of France, has 4 royal descendants      |
| [V13] FRANCOIS I, King of France, has 4 royal descendants       |
| ...                                                             |
| [V16] CHARLES IX, King of France, has 0 royal descendants       |
| [V17] HENRI III, King of France, has 0 royal descendants        |
+-----------------------------------------------------------------+
```

Figure 12.4 - An expressive presentation of the data

In addition, we will consider kings only: we count the *kings* in the descent of each *king* of France. Hence the query of Script 12.8, which deserves a bit of explanation

This query is built as a join of view KING [1] with the former expression [2-9] to extract additional information and to limit the members to those who are kings. Counting kings only in their descent is obtained by condition [4]. So, both ancestors and descendants are kings of France.

```
select '['||Ancestor||'] '||K.Name||', King of France, has
       '||Descent||' royal descendants'
from KING K,                                                    [1]
     (select Ancestor, count(*) as Descent                      [2]
      from   PATHS                                              [3]
      where  Descendant in (select PiD from KING)               [4]
      group by Ancestor                                         [5]
         union                                                  [6]
      select PiD as Ancestor, 0 as Descent                      [7]
      from   MEMBER                                             [8]
      where  PiD not in (select Ancestor from PATHS)) DC        [9]
where DC.Ancestor = K.PiD                                      [10]
order by Descent desc,Ancestor;
```

Script 12.8 - Generating the description of kings with the number of descendants

12.4 Showing the descendants of a member

In the next project, we will produce an **indented list** of the **descendants** of a member. Each son is displayed with a *two space* right shift + a dash character to make the hierarchy explicit. We would like something like the list of Figure 12.5.

```
Descent of LOUIS IX, roi de France:

- PHILIPPE III, Roi de France
  - PHILIPPE IV, Roi de France
    - LOUIS X, Roi de France
      - JEAN I, Roi de France
    - PHILIPPE V, Roi de France
    - CHARLES IV, Roi de France
  - Charles, Comte de Valois
    - PHILIPPE VI, Roi de France
      - JEAN II, Roi de France
        - CHARLES V, Roi de France
          - CHARLES VI, Roi de France
            - CHARLES VII, Roi de France
              - LOUIS XI, Roi de France
                - CHARLES VIII, Roi de France
          - Louis I, Duc d'Orléans
            - Charles, Duc d'Orléans
              - LOUIS XII, Roi de France
            - Jean, Comte d'Angoulême
```

```
              - Charles, Comte d'Angoulême
                - FRANCOIS I, Roi de France
                  - HENRI II, Roi de France
                    - FRANCOIS II, Roi de France
                    - CHARLES IX, Roi de France
                    - HENRI III, Roi de France
- Robert, Comte de Clermont
  - Louis I, Duc de Bourbon
    - Jacques I, Comte de La Marche
      - Jean I, Comte de La Marche
        - Louis, Comte de Vendôme
          - Jean VIII, Comte de Vendôme
            - François, Comte de Vendôme
              - Charles, Duc de Vendôme
                - Antoine, Duc de Vendôme
                  - HENRI IV, Roi de France
                    - LOUIS XIII, Roi de France
                      - LOUIS XIV, Roi de France
                        - Louis, Grand Dauphin
                          - Louis, Petit Dauphin
                            - LOUIS XV, Roi de France
                              - Louis, Dauphin de France
                                - LOUIS XVI, Roi de France
                                - LOUIS XVIII, Roi de France
                                - CHARLES X, Roi de France
                      - Philippe I, Duc d'Orléans
                        - Philippe II, Duc d'Orléans
                          - Louis, Duc d'Orléans
                            - Louis Philippe I, Duc d'Orléans
                              - Philippe Égalité, Duc d'Orléans
                                - LOUIS PHILIPPE I, Roi de France
```

Figure 12.5 - Indented list of the descent of LOUIS IX

The members are listed according to the *depth first traversal* described in Section 19.6 (*Recursive programming*) of the Tutorial, from which we derive Script 12.9.

```
ask pid = Member:[!select PiD from MEMBER order by PiD];
extract Nam,Tit = select Name,Title
                  from   MEMBER where PiD = '$pid$';

write-a Descent of $Nam$,$Tit$;

with recursive TREE(Ind,Father,PiD,Path) as
 (select '','$pid$',PiD,'.'||'$pid$'||'.'||PiD||'.'
  from   MEMBER where Father = '$pid$'
     union
  select T.Ind||'  ',T.PiD,M.PiD,T.Path||M.PiD||'.'
  from   TREE T, MEMBER M
  where  T.PiD = M.Father)

select Ind||'- '||M.Name||', '||M.Title as ""
from   TREE T, MEMBER M
where  T.PiD = M.PiD
order by T.Path;
```

Script 12.9 - Extracting the descent of a member through a recursive query

Alternative computing through a loop-based script

The scripts 12.10 use another technique for generating this hierarchical text: a loop scans the resultset of the CTE described above and writes, for each of its row, a line of the tree.

```
for txt =
    [with recursive TREE(Ind,Father,PID,Path) as
     (select '','$pid$',PiD,'.'||'$pid$'||'.'||PiD||'.'
      from   MEMBER where Father = '$pid$'
        union
      select T.Ind||'  ',T.PiD,M.PiD,T.Path||M.PiD||'.'
      from   TREE T, MEMBER M
      where  T.PiD = M.Father
     )
     select Ind||'- '||M.Name||', '||M.Title as ""
     from   TREE T, MEMBER M
     where  T.PiD = M.PiD
     order by T.Path
    ];
   write @s$txt$;
endfor;
```

Script 12.10 - Extracting the descent of a member through a loop-based script

12.5 Graphical representation of king genealogy

Instead of merely printing member data as a pure text as we did in the previous section, it would be nicer to **draw** the tree as we may find it in some web sites.

We can use the SQLdraw engine to present the data in a graphical way. We first generate an SQLdraw script file representing the tree, then we render it with the `showDrawing` statement.

The detail of the generation of the SQLdraw file is described in complement document SQLfast-Case-Kings-of-France-Draw.pdf available in the script directory of this case study.

Figure 12.6 shows a simple representation of the *Father-son* tree of the members. In Figure 12.7, an improved drawing allows a better understanding of the tree structure. In addition, king descriptors are written in blue, which is the natural color of (the blood of) kings of France!

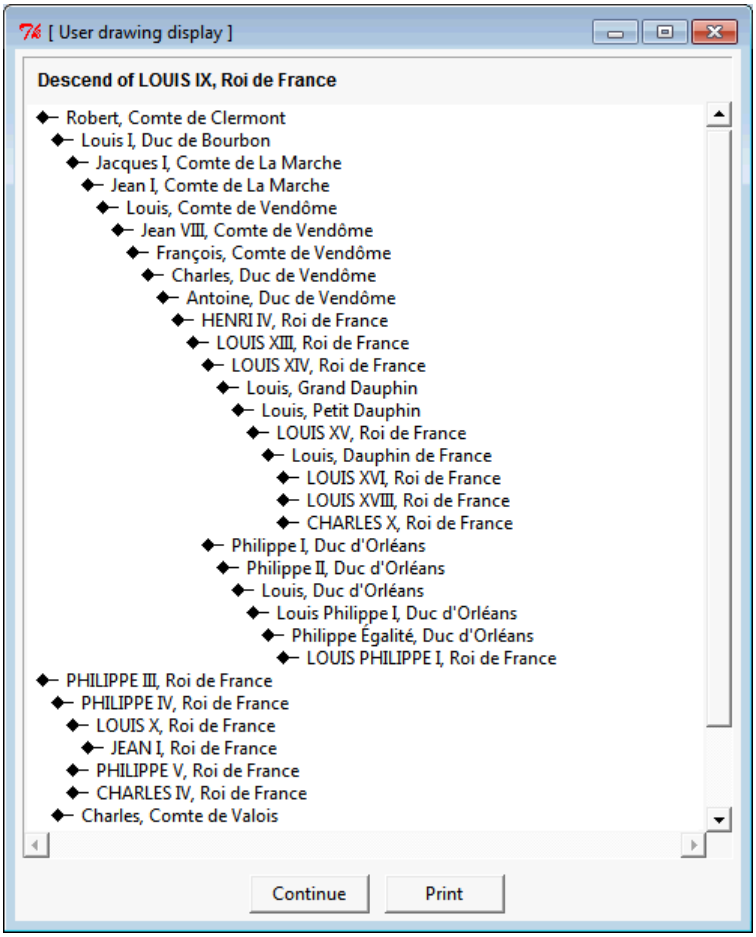


Figure 12.6 - The descent of King LOUIS IX (excerpt) - Simple version

12.6 Recovering the source data from their closure

This is a basic exercise that seems to have no immediate practical interest, but that will prove quite useful for the project discussed in the next section. The problem can be formulated as:

> *how can we extract from the 850 rows of the closure of table MEMBER the source data recorded in MEMBER?*

More precisely, we would like to select the couples (Father,PiD) of the closure that also belong to table MEMBER. This operation, that removes all the couples that can be built by composing two other couples, is called **transitive reduction**.

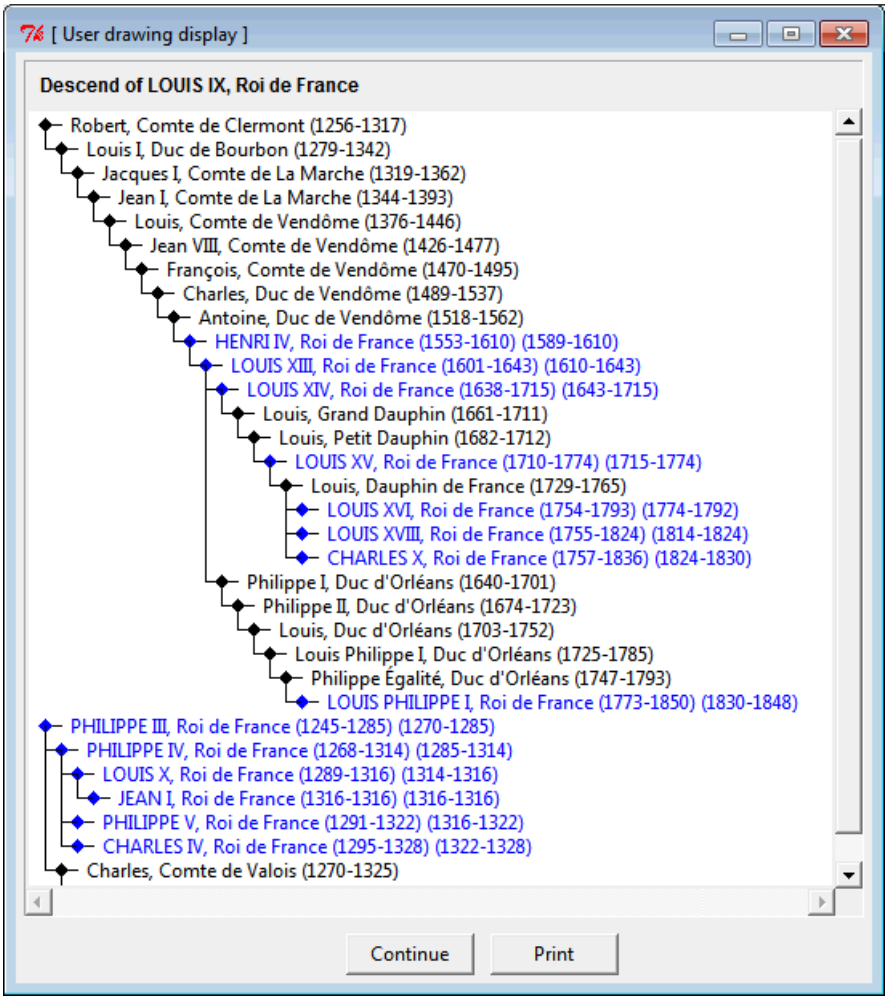


Figure 12.7 - The descent of King Louis IX - Improved version

The solution can be designed on the basis of some properties of these couples of the closure. We describe three of them:

1. If the closure is complemented with the length information (column **Len**), the selected couples satisfy condition (`Len = 1`), this is clearly the simplest and most efficient procedure.
2. A couple (M1,M3) of the closure is selected if there is **no** member M2, different from M1 and M3, such that couples (M1,M2) and (M2,M3) are in the closure.
3. A couple (M1,M3) of the closure is selected if there is **no** pair of couples (Ma,Mb) and (Mc,Md) in the closure such that M1 = Ma, Mb = Mc and Md = M2).

Query [1] of Script 12.11 implements the third property, which does not require **Len**.

```
set Closure = select PiD,PiD from MEMBER
                 union all
              select P.Ancestor, M.PiD
              from   CTE_PATHS P,MEMBER M
              where  P.Descendant = M.Father;
create view PATHS as
with recursive CTE_PATHS(Ancestor,Descendant) as ($Closure$)
select Ancestor,Descendant from CTE_PATHS
where  Ancestor <> Descendant;

select P.Ancestor,P.Descendant                                    [1]
from   PATHS P
where  not exists (select 1 from PATHS Pab, PATHS Pcd
                   where  P.Ancestor = Pab.Ancestor
                   and    Pab.Descendant = Pcd.Ancestor
                   and    Pcd.Descendant = P.Descendant)
order by P.Ancestor,P.Descendant;
```

Script 12.11 - Transitive reduction of a closure: retrieving the source Father/Son relationships

12.7 Extracting the hierarchy of kings

This project is more concrete and can find applications in various actual situations. The goal is to extract from table MEMBER a subtree that comprises *all the kings and only them*. All the members who are not king do not appear in the hierarchy. Figure 12.8 shows what we want to extract. The links no longer represent the *father/son* relationships but rather the *closest king ancestor*/*king* relationships.

Selecting the kings among the members is easy, but combining the *father/son* links between them to produce *ancestor/descendant* links seems a bit more tricky. When *non-king* members appear between two kings, they are skipped and these kings are connected by a direct link.

Let us first observe that the *ancestor/descendant* links we are looking for are in the closure of MEMBER. Moreover, if we select in this closure the couples of two kings, we get the links we are looking for, but also those links that are combinations of the latter. In short, the full set of the *ancestor/descendant* links form the *closure* of the desired set of couples.

We consider the example of path .**B11**.B12.B13.**B14**.B15.**B16**., illustrated in Figure 12.9. This path includes three kings, namely LOUIS XIV (B11), LOUIS XV (B14) and LOUIS XVI (B16).

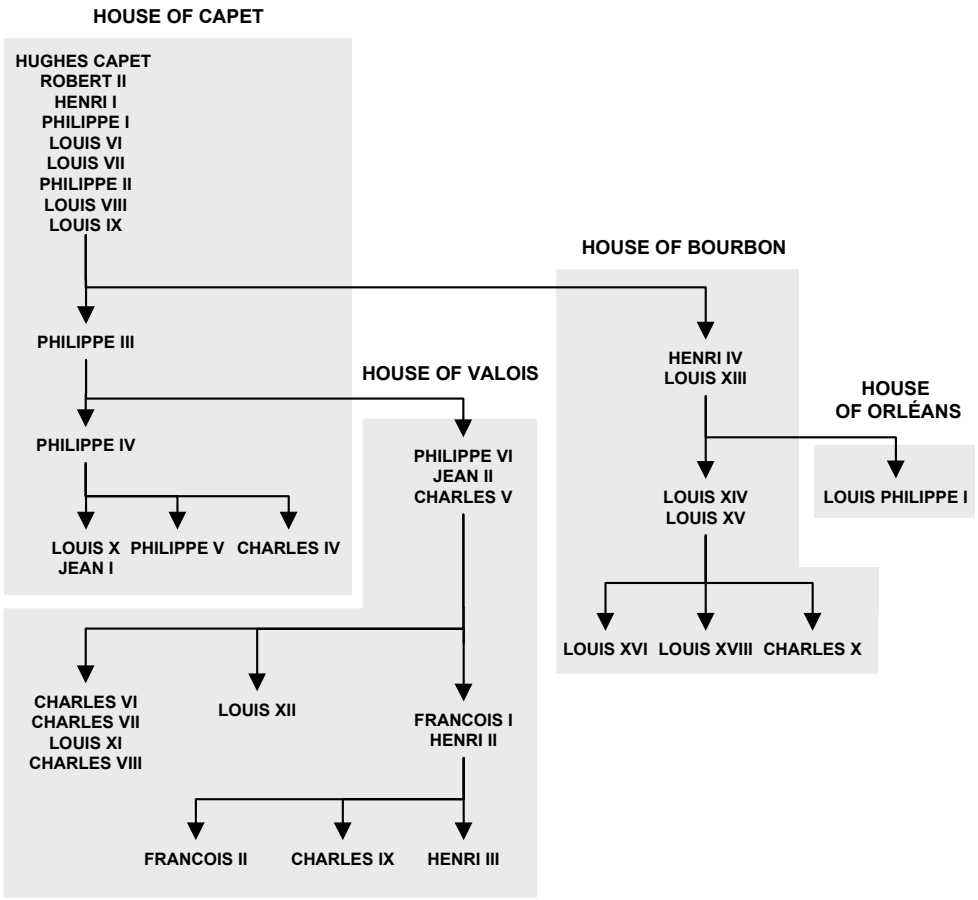


Figure 12.8 - The hierarchy reduced to the kings

The left side schema shows the fragment of the closure of MEMBER between B11 and B16. The central schema shows the kings and the links connecting them. The right side schema is the subset of these links we are looking for. We call the result the **projection** of MEMBER on its king members.

These observations provide us with a means to extract what we want:

– We select KK, the set of the couples of the closure of MEMBER in which both members are kings (central schema of Figure 12.9).

– We extract from KK the couples that cannot be built by composing two other couples (left side schema of Figure 12.9). In other words, we extract the *transitive reduction* of KK, a problem we have addressed in the previous section.

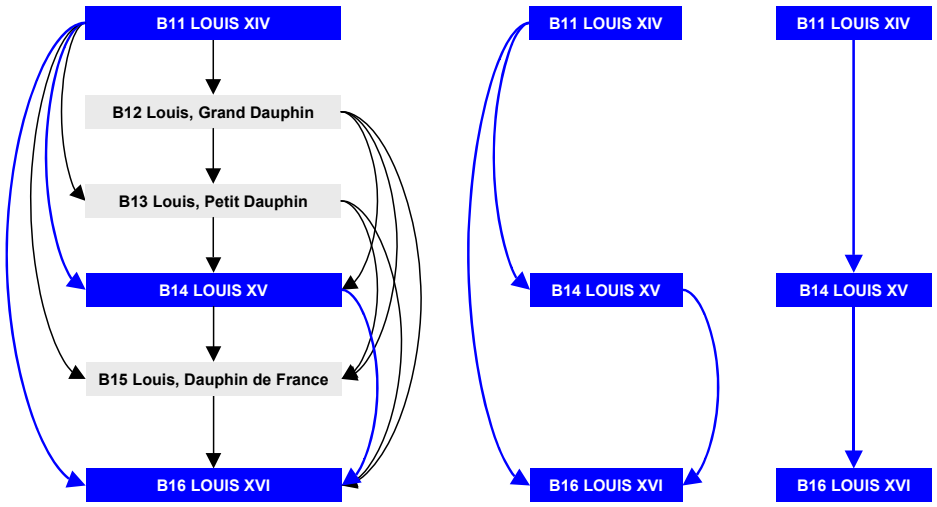


Figure 12.9 - Fragment of the closure of MEMBER (left), of its projection on kings (center) and of the transitive reduction (right)

Once again, there are several ways to solve the transitive reduction problem, each based on properties such as the following:

1. A couple (K1,K3) of KK, with path P13, is selected if there is **no** king K2, different from K1 and K3, that appears in P13. This is the simplest and the more efficient procedure, but it requires the recording of the paths.
2. A couple (K1,K3) of KK is selected if there is **no** pairs of couples (Ka,Kb) and (Kc,Kd) in the closure of MEMBER such that K1 = Ka, Kb = Kc and Kd = K2).
3. A couple (K1,K3) of KK is selected if there is **no** king K2, different from K1 and K3, such that couples (K1,K2) and (K2,K3) are in the closure of MEMBER.

Script 12.12 is based on the first idea. The main query joins each component of the couples of PATHS with its respective MEMBER [1,3]. Condition [2] discards the initialization couples. Conditions [4] force the elements of the couples to be kings. Condition [5] discards the couples, the path of which includes another king.

Figure 12.10 shows the result of the execution of this script while Figure 12.11 presents this information in a graphical way.

```
set Closure = select PiD,PiD,'.'||PiD||'.' from MEMBER
                 union all
              select P.Ancestor, M.PiD, P.Path||M.PiD||'.'
              from   PATHS P,MEMBER M
              where  P.Descendant = M.Father;
with recursive PATHS(Ancestor,Descendant,Path) as ($Closure$)
select P.Ancestor,K1.Name,P.Descendant,K3.Name,P.Path
from   MEMBER K1, PATHS P, MEMBER K3                              [1]
where  P.Ancestor <> P.Descendant                                 [2]
and    P.Ancestor = K1.PiD and P.Descendant = K3.PiD              [3]
and    K1.Title = 'Roi de France'                                 [4]
and    K3.Title = 'Roi de France'                                 [4]
and not exists (select count(*) from KING K3                      [5]
                where  P.Path like '_%.'||K3.PiD||'.%_')
order by P.Ancestor,P.Descendant;
```

Script 12.12 - Extracting the royal hierarchy

Ancestor	Name	Descendant	Name	Path
B09	HENRI IV	B10	LOUIS XIII	.B09.B10.
B10	LOUIS XIII	**B11**	**LOUIS XIV**	.B10.B11.
...				
B11	LOUIS XIV	**B14**	**LOUIS XV**	**.B11.B12.B13.B14.**
B14	LOUIS XV	**B16**	**LOUIS XVI**	**.B14.B15.B16.**
B14	LOUIS XV	B17	LOUIS XVIII	.B14.B15.B17.
B14	LOUIS XV	B18	CHARLES X	.B14.B15.B18.
C00	HUGHES CAPET	C01	ROBERT II	.C00.C01.
C01	ROBERT II	C02	HENRI I	.C01.C02.
C02	HENRI I	C03	PHILIPPE I	.C02.C03.
C03	PHILIPPE I	C04	LOUIS VI	.C03.C04.
C04	LOUIS VI	C05	LOUIS VII	.C04.C05.
C05	LOUIS VII	C06	PHILIPPE II	.C05.C06.
C06	PHILIPPE II	C07	LOUIS VIII	.C06.C07.
C07	LOUIS VIII	C08	LOUIS IX	.C07.C08.
...				
C08	LOUIS IX	C09	PHILIPPE III	.C08.C09.
C09	PHILIPPE III	C10	PHILIPPE IV	.C09.C10.
C09	PHILIPPE III	V01	PHILIPPE VI	.C09.V00.V01.
C10	PHILIPPE IV	C11	LOUIS X	.C10.C11.
C10	PHILIPPE IV	C12	PHILIPPE V	.C10.C12.
C10	PHILIPPE IV	C13	CHARLES IV	.C10.C13.
C11	LOUIS X	C14	JEAN I	.C11.C14.
V01	PHILIPPE VI	V02	JEAN II	.V01.V02.
V02	JEAN II	V03	CHARLES V	.V02.V03.
...				
V13	FRANCOIS I	V14	HENRI II	.V13.V14.
V14	HENRI II	V15	FRANCOIS II	.V14.V15.
V14	HENRI II	V16	CHARLES IX	.V14.V16.
V14	HENRI II	V17	HENRI III	.V14.V17.

Figure 12.10 - The royal hierarchy produced by Script 12.12

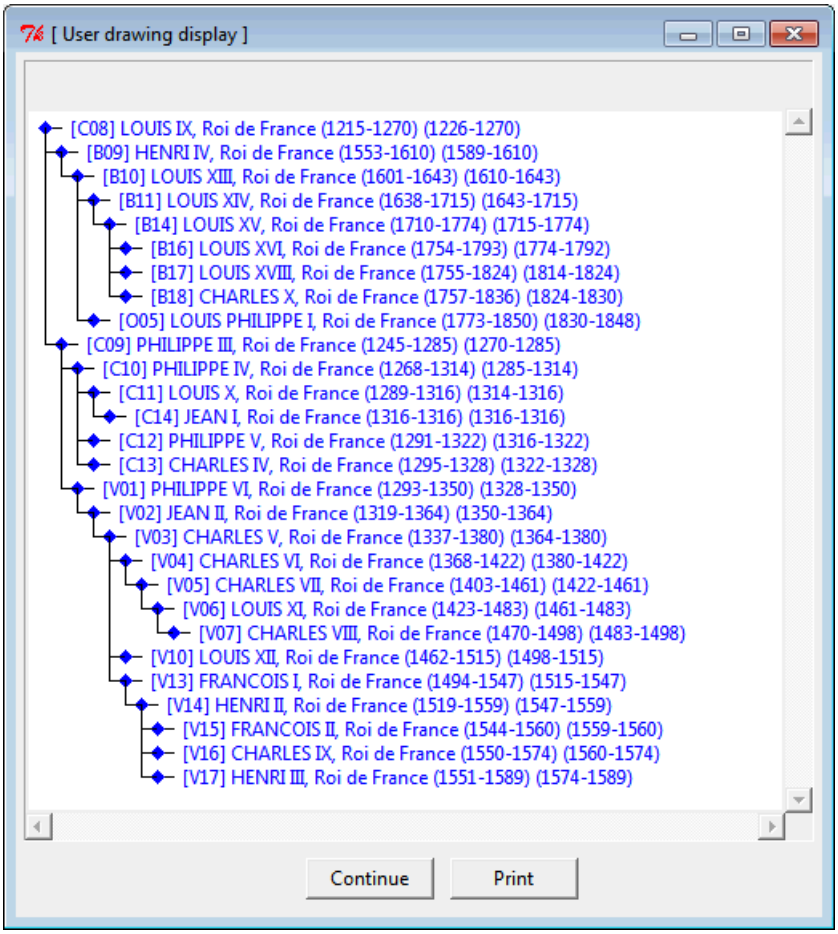


Figure 12.11 - The royal hierarchy based on ancestor/descendant relationships

12.8 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Kings-of-France**. Actually, they can be run from main script **Kings-MAIN.sql**, that displays the selection box of Figure 12.12.

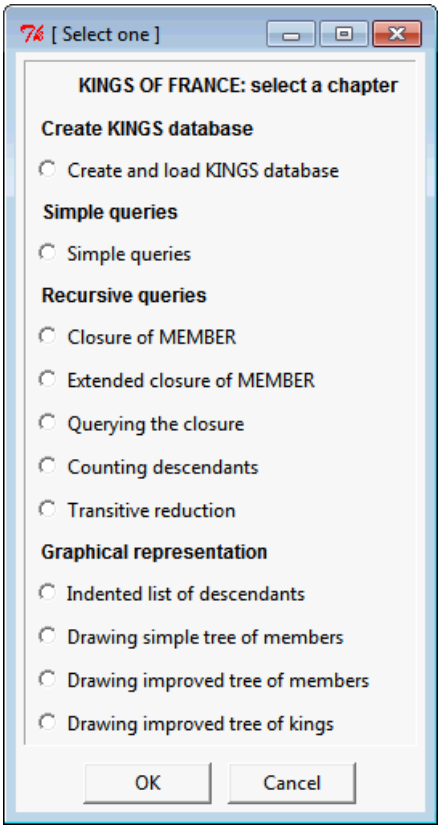


Figure 12.12 - Kings of France: selecting an application

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

Case study 12

Complement on Kings of France

Objective. This document is a complement to case study *Kings of France - Part 2*. It details the development of graphical representation of the data of KINGS.db database with the vector graphics engine of SQLfast.

1. Graphical representation of king genealogy - version 1

Instead of merely printing member data as a pure text as we did in the previous section, it would be nicer to **draw** the tree as we may find it in some web sites.

We will use the SQLdraw engine to present the data in a graphical way. We first generate an SQLdraw script file representing the tree, then we render it with the `showDrawing` statement.

Let us begin with a first draft such as that of Figure 12.2. We can build it very simply by drawing, for each member, two points located at the same position:

– an *ornemental* point represented by a small black diamond, without label; a point model 31 is fine[1] (Figure 12.1),

– a point represented by a simple dash, followed by the name and title of the member; here, we choose a point model 68.

1. A point is defined by its coordinates, in horizontal and vertical pixels, by its type and by a label, possibly empty. The illustrated catalogue of SQLdraw points is presented in Chapter 21 of the Tutorial.

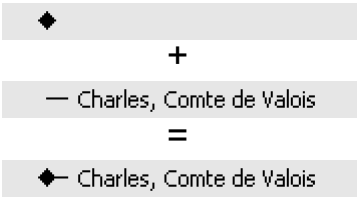


Figure 12.1 - Detail of the representation of a member

The first five members will be drawn by the following script.

```
area 370,780
points " ",1,"black","black",68
10,10,"Robert, Comte de Clermont"
points " ",1,"black","black",31
10,10,""
points " ",1,"black","black",68
20,25,"Louis I, Duc de Bourbon"
points " ",1,"black","black",31
20,25,""
points " ",1,"black","black",68
30,40,"Jacques I, Comte de La Marche"
points " ",1,"black","black",31
30,40,""
points " ",1,"black","black",68
40,55,"Jean I, Comte de La Marche"
points " ",1,"black","black",31
40,55,""
points " ",1,"black","black",68
50,70,"Louis, Comte de Vendôme"
points " ",1,"black","black",31
50,70,""
```

SQLdraw directive **area** specifies the dimensions of the drawing space. The (x,y) coordinates are measured in pixels and the origin of this space is (0,0) at the top-left corner. The first son of Louis IX, Robert, Comte de Clermont, is positioned at (10,10), the next members being drawn 10 point to the right of their father and 15 point downward from the previous one. Directive **points** starts a group of points (only one point here) by defining its thickness (1), its color (black) and its type (68 and 31). A group is assigned a label, here a simple space character " ", which can be ignored. An individual point is defined by its x, y coordinates and its label.

Script 12.1 builds and displays the descent of a selected member. Its principle derives from that of Script 12.2.

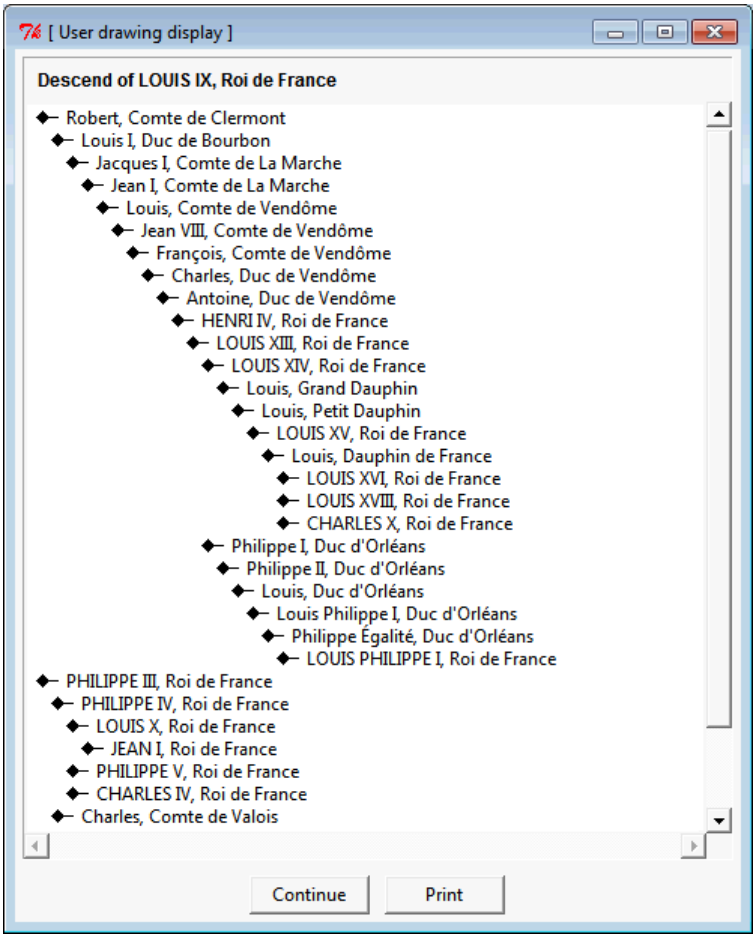


Figure 12.2 - The descent of King LOUIS IX (excerpt) - First version

As it is used in several other queries, TREE is transformed into an independent view. For a much larger source MEMBER table, building a temporary table would probably be more efficient (*created once, used many times*).

The text of the SQLdraw script is generated into variable draw [3], then stored in file KINGdrawing.draw [6,7]. We choose to generate the script through an SQL query [5]. To avoid column names and tabular layout, we apply the elementary List output format [2]. This query, which produces four SQLdraw lines for each row, appears to be fairly complicated. To simplify its writing, we define a series of shorthands [4], through variables defining the size of the tree, in pixels (maxX, maxY), the expression of the position of each member in the tree, in pixels (coordX, coordY), and, for each member, the four lines defining the two points (header68, point68, header31, point31). Finally, the SQLdraw script is rendered graphically [8].

```
ask pid = member:[!select PiD from MEMBER order by PiD];
extract Nam,Tit = select Name,Title
                  from   MEMBER where PiD = '$pid$';

create temp view TREE as
  with recursive PATHS(Len,Father,PiD,Path) as
  (select 0,'$pid$',PiD,'.'||'$pid$'||'.'||PiD||'.'
   from   MEMBER where Father = '$pid$'
      union all
   select P.Len+1,P.PiD,M.PiD,P.Path||M.PiD||'.'
   from   PATHS P,MEMBER M
   where  P.PiD = M.Father
   )
  select Len,Father,PiD,Path from PATHS;
```

```
execSQL UTIL-SELECT-parameters-for-Lists.sql;                    [2]
outputOpen draw.var;                                             [3]
-- Some shorthands --                                            [4]
set maxX = max((Len+1)*10);
set maxY = max((select count(1) from TREE T2
                where  T2.Path < T.Path)*15)+10;
set coordX = cast((Len+1)*10 as char);
set coordY = cast((select count(1) from TREE T2
                   where  T2.Path < T.Path)*15+10  as char);
set header68 = 'points " ",1,"black","black",68@n';
set point68  = $coordX$||','||$coordY$||',"'||M.Name||',
               '||M.Title||'"@n';
set header31 = 'points " ",1,"black","black",31@n';
set point31  = $coordX$||','||$coordY$||',""';
select 'area '||cast($maxX$+200 as char)||','
       ||cast($maxY$+50 as char) as Area from TREE T;
-- The generation query --
select $header68$||$point68$||$header31$||$point31$              [5]
from   TREE T,MEMBER M where T.PiD = M.PiD order by Path;
outputOpen KINGdrawing.draw;                                     [6]
write $draw$;                                                    [7]
showDrawing KINGdrawing.draw = [/x450/y750Descent of $Nam$];     [8]
```

Script 12.1 - Extracting the descent of a member (simple graphical view)

2. Graphical representation of king genealogy - version 2

Now, we will improve the drawing in three aspects. First, we add the *life period* of the members and, for kings, the *reign period*. Secondly, we color the king lines in blue (nobles and royal family members were said to have *blue blood*!). Thirdly, we draw connectors between members in order to make *father-son* relationships more explicit. To summarize, we propose to generate the drawing of Figure 12.4, a detail of which is shown in Figure 12.3.



Figure 12.3 - Detail of the representation of members, their status and their father-son relationships

Though the procedure could be developed as an extension of the former procedure, we will implement it in a slightly different way. First, view TREE will be transformed into a real (temporary) table. This table already includes all the coordinates of the graphic points, to avoid further computation at generation time. Secondly, the central query will not rely on variables but will be given a layout that evokes the list of eight lines to generate for each member. In addition, the first line must be prepared in a special way, since it has no parent line.

Script 12.2 creates table TREE, with five new columns: **Seq**, a sequence number generated automatically, **X** and **Y** the current coordinates and **FX** and **FY** the coordinates of the father. Note that the `order by` clause is necessary to let the values of **Seq** follow the *depth first* traversal of the tree in construction. Column **X** is computed at load time as (Len + 1) * 10 . Column **Y**, **FX** and **FY** are set through an `update` query. Column **Y** is derived from the value of **Seq**, while the values of columns **FX** and **FY** are extracted from the row of the father.

The main query is shown in Script 12.3. The sequence of elements that build each of the eight SQLdraw lines are identified by number [1] to [8]. The big `case-end` block noted [*] controls the generation of the *first* and the *following* lines.

Reign period and blue color: in sequences [1], [2] and [3], they are generated when the title of the member is `'Roi de France'`. The choice is expressed by three `case-end` function.

Graphical links. The link between a father and one of his son is drawn as a *polyline*, that is, sequence of connected segments. The SQLdraw directive **`polyline`** defines a label (ignored here), the thickness of the segments (here **1**), the color of the segments (**black**) and the fill color (empty) if the polyline also is a polygon, which is meaningless here. The polyline comprises a vertical segments and a horizontal segment, defined by three points. To draw them, we must know the coordinates of the current member (columns **X**, **Y**) and those of his father (columns **FX**, **FY**).

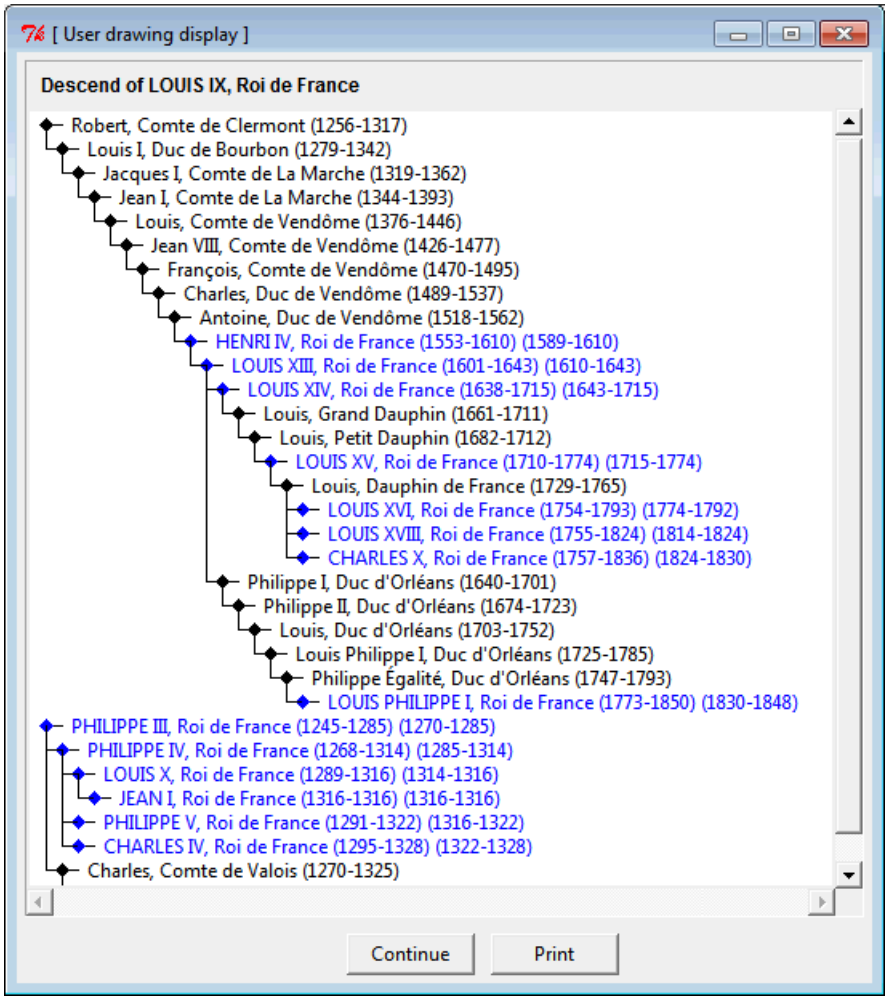


Figure 12.4 - The descent of King Louis IX

The following SQLdraw script fragments result from the execution of Script 12.2 and 12.3.

```
points " ",1,"black","black",68
30,40,"PHILIPPE IV, Roi de France (1268-1314) (1285-1314)"
points " ",1,"black","black",31
30,40,""
polyline " ",1,"black",""
20,25
20,40
30,40
points " ",1,"blue","blue",68
```

```
      30,115,"Charles, Comte de Valois (1270-1325)"
      points " ",1,"black","black",31
      30,115,""
      polyline " ",1,"black",""
      20,25
      20,115
      30,115
```

```
create temporary table TREE (
     Seq integer primary key autoincrement,
     Len integer, Father char(6),
     PiD char(6), Path varchar(1024),
     X integer, Y integer, FX integer, FY integer);
with recursive PATHS(Len,Father,PiD,Path) as ($Closure$)
insert into TREE (Len,Father,PiD,Path,X)
select Len+1,Father,PiD,Path,(Len+1)*10
from PATHS order by Path;
update TREE
set Y = Seq*15 - 5,
   FX = (select X from TREE T where PiD = TREE.Father),
   FY = (select Seq*15-5 from TREE T where PiD = TREE.Father);
extract maxX,maxY = select max(X)+320,max(Y)+40 from TREE;
```

Script 12.2 - Descent of a member: preparing table TREE

A procedural approach

The main query in Script 12.3, which packs in one statement all the logic of text generation may appear large and complex (it is, but basically because the text itself is complex), and therefore prone to reading and writing errors. It could be decomposed into a series of `select` queries, each addressing a specific pattern: first line, following lines, kings, other members.

Some readers may find a more procedural expression better suited. Script 12.4 is such a proposal. It replaces the large query of Script 12.3.

The procedure is structured as a large loop, traversing the TREE table.

The case *first/non-first* member is described through variable **first** (value 1 for *first* and 0 for *non-first*). This variable is initialized to 1, then is set to 0 at the end of the first (actually of all) iteration of the loop.

Whether a member is a king or not is described by variable **king** (value 1 for *king* and 0 for *non-king*), set by the SQL query of the loop header.

```
write area $maxX$,$maxY$;
select case when M.Title = 'Roi de France'                    [1]
            then 'points " ",1,"black","blue",68@n'
            else 'points " ",1,"black","black",68@n'
       end
     ||cast(X as char)||','||cast(Y as char)||','"'           [2]
                      ||M.Name||', '||M.Title
                      ||' ('||M.DateBirth
                           ||'-'||M.DateDeath||')'
                      ||case when M.Title = 'Roi de France'
                             then ' ('||M.DateReignBegin
                                      ||'-'
                                      ||M.DateReignEnd||')"@n'
                             else '"@n'
                        end
     ||case when M.Title = 'Roi de France'                    [3]
            then 'points " ",1,"blue","black",31@n'
            else 'points " ",1,"black","black",31@n'
       end
     ||cast(X as char)||','||cast(Y as char)||',""'           [4]
     ||case when T.FX is null then ''                         [*]
            else '@n'
                 ||'polyline " ",1,"black",""@n'             [5]
                 ||cast(FX as char)||','                     [6]
                 ||cast(FY as char)||',""@n'
                 ||cast(FX a s char)||','                    [7]
                 ||cast(Y as char) ||',""@n'
                 ||cast(X as char) ||','                     [8]
                 ||cast(Y as char) ||',"'
       end
from   TREE T,MEMBER M where T.PiD = M.PiD order by Path;
```

Script 12.3 - Descent of a member: generating the script of the full graphical view. [KING-20-Members-Full-Drawing.sql]

Both variants provide the same result, but a slight difference can be observed, notably for the longest descent, that of Hughes Capet. The procedural script takes more time to build the SQLdraw script: **1.125** seconds for the SQL loop in the procural script against **0.031** seconds for the pure SQL query. Such a difference is not significant for small problems but may be an issue for larger ones. However, these figures depends on technical options of the SQLfast interpreter, options that may evolve in the future.

Note

Script **King-Full-Drawing-with-Root.sql** is a minor extension of Script **King-Full-Drawing.sql** that also displays the root of the tree and the PiD value of each member. Considering that the *House* is represented by the first character of PiD,

the picture produced (available in file **Capetian-dynasty.jpg**) can be seen as the graphical equivalent of the source data.

```
for Seq,Len,Father,PiD,Path,X,Y,FX,FY,
    nam,tit,datB,datD,dateRB,dateRE,king
    = [select Seq,Len,M.Father,M.PiD,Path,X,Y,FX,FY,
           Name,Title,DateBirth,DateDeath,
           DateReignBegin,DateReignEnd,
           case when Title = 'Roi de France' then 1 else 0 end
       from TREE T,MEMBER M
       where  T.PiD = M.PiD order by Path];
   set color = black;
   if ($king$ = 1) set color = blue;
   write points " ",1,"black","$color$",68;
   set line = $X$,$Y$,"$nam$, $tit$ ($datB$-$datD$);
   if ($king$ = 1) set line = $line$ ($dateRB$-$dateRE$);
   write $line$";
   write points " ",1,"$color$","black",31;
   write $X$,$Y$,"";
   if ($first$ = 0);
       write polyline " ",1,"black","";
       write $FX$,$FY$,"";
       write $FX$,$Y$,"";
       write $X$,$Y$,"";
   endif;
   set first = 0;
endfor;
```

Script 12.4 - Descent of a member: generating the script of the full graphical view through an SQLfast loop-based script

Case study **14**

The book of which you are the hero

Objective. Game books are traditional text-based adventure games made up of a collection of pages (episodes) connected by references (branches). An episode comprises a text that describes a situation or an action and one or several branches that allow the gamer to jump to other episodes. Many of them are now available as *pdf* or *html* documents. In this study, we implement a simple game engine that automates such game books. This engine is based on a game database that can also be used to automatically generate stories.

Actually, this project is a nice opportunity to examine in some detail the concept of graph (a game book basically is a set of nodes and a set of edges) and to develop exploration and transformation algorithms. In particular, we study the structure of a game graph, we identify its anomalies, we extract its circuits, we build and count the different possible runs from the starting episode to an exit episode, we search for unreachable episodes and dead-end branches and we identify episodes that can be merged.

A representative heroic fantasy game book has been encoded and all the algorithms developed in the study are provided as SQLfast scripts.

Keywords. computer game, game engine, story generation, graph, cyclic graph, acyclic graph, graph transformation, Marimont algorithm, reachability, circuit, elementary circuit, transitive closure, cyclic kernel, set comparison.

14.1 Interactive adventure books and computer games

Those of us *over a certain age* should certainly fondly remember their first computer game, *Colossal Cave Adventure* (1977) or *Zork* (1979), both released on the PDP-10 mini-computer. These text-based adventure games relied on a primitive human-computer conversation: the computer writes a short description of a place or of actions taking place, then lets the user type commands, such as `go west`, `open door`, `take sword` or `kill troll with sword` (Figure 14.1).[1]

These games were then ported on personal computers. Some of them included simple graphics: instead of (or in addition to) being told that `In front of you an old house with a door and two broken windows`, the player can see a sketch of this house in B/W or in 16 colors (Figure 14.2). In addition, to make these games accessible to casual players, the available commands were written below the descriptive text, so that typing their sequence number, moving the cursor or, later, clicking on them was sufficient to move to the next step. In Figure 14.3, the user clicks on a verb, then on an object, to form a short `action-object` command.

In the 80's, descriptive texts and their list of commands (together forming an *episode*) were printed on paper and sold in bookshops. The reader was invited to follow the adventure by moving from episode to the next one, the page of which was mentioned in the command selected by the reader. These non-linear interactive books, where the reader plays an active role, were called *game books* or *books of which you are the hero*.

This is where things are getting interesting!

The structure of adventure games or game books as a set of episodes connected by commands submitted to the player is simple enough to think of some kind of automation through a database. Episodes and commands (we will call them *branches* from one episode to the next one) are stored in a database. Episodes and branches are extracted by SQL queries and presented to the player through a dialogue box.

14.2 Choosing sample game books

Quite surprisingly, nearly forty years after their first appearance, text-based adventure games still are very popular, gathering a worthy audience of retro-game amateurs[2]. Therefore, it shouldn't be too difficult to find one or two games for the sake of demonstration.

The site *Chronicles of Arborell* (http://www.arborell.com) provides a rich collection of small to large *Heroic Fantasy* game books written by Wayne Densley and available as pdf or html documents.

1. See http://classicgames.about.com/od/computergames/p/Adventures-In-Text-The-Origin-Of-Adventure-Games.htm
2. See http://textadventures.co.uk for instance.

```
You are facing the north side of a white house. There is no door here,
and all the windows are barred.

> go east

You are behind the white house. In one corner of the house
there is a window which is slightly ajar.

> open window

With great effort, you open the window far enough to allow passage.

> enter window

You are in the kitchen of the white house. A table seems to have
been used recently for the preparation of food. A passage leads to
the west, and a dark staircase can be seen leading upward. To the
east is a small window which is open.
On the table is an elongated brown sack, smelling of hot peppers.
A clear glass bottle is here.
The glass bottle contains:
A quantity of water.

> take sack

Taken.

> take bottle

Taken.

> go staircase

I don't understand that.

>
```

Figure 14.1 - Excerpt from Zork (1979)



Figure 14.2 - Mystery house (by On-Line Systems, Apple II, 1980)



Figure 14.3 - Maniac Mansion (by Lucasfilm Games, Commodore 64, 1987)

Amongst this collection, we have chosen two games,

– **TheWatchtower**[3], a small game book of 16 pages, 9 of which devoted to the description of 39 episodes (called *sections*) and 136 branches. The author modestly qualifies it *Micro-gamebook*, though we will show later that its structure may be quite complex.

– **WindhammeR**[4]: a large game book of 618 pages, 599 episodes and 993 branches.

In *TheWatchtower*, episode 15 (*section* 15 in the game) is written as follows (Figure 14.4):

> 15
>
> Moving towards the table in the south-west corner you find a collection of scrolls, journals and small paper packets filled with a grey powder. The scrolls are written in a dialect of the Hordim and you recognize them quickly for what they are.
>
> What you have found are research notes, and you understand now what the Hordim have been doing here. The grey powder is a poison and from the notes you can see that it is being altered so that it is specific to Humans alone. Luckily the Hordim have not yet perfected its chemistry and you cannot let them succeed. This is no simple raiding party. They are experimenting on human subjects and for you that presents a greater problem. Research is not a function of the Hresh for they serve as warriors alone. There is a Mutan Overseer here and he will not be far from his experiments.
>
> If you have not already done so and would begin a general search of this level *turn to section 22*.
>
> If you would make directly for the row of beds against the northern wall *turn to section 20*.

3. http://www.arborell.com/thewatchtower_text.html. "*This is a freeware title and can be copied and distributed free of charge, however all rights are reserved by the author.*"
4. http://www.arborell.com/windhammer.pdf. [same copyright as above]

If you would instead make for the lone bed in the south-eastern corner *turn to section 13*.

If there is good cause to make for the stairway to the battlements above *turn to section 17*.

Figure 14.4 - The contents of episode 15

This text comprises a short description of what appears in this episode, or, for action episodes, what happens, followed by a series of four possible decisions. Each one formulates a proposition of action and indicates the next episode number to move to, to continue the adventure.

14.3 Designing a text-based interactive adventure game

Our primary goal is to store the content of an interactive game book in a game database and to develop an SQLfast script (pompously called *game engine*!) that allows us to play this game.

We store the description of episodes in table EPISODE and the definition of branches in table BRANCH.

An episode is defined by its unique id (EPID) and the text to display (TEXT). A branch is described by the id of its source episode (EPID, a foreign key to EPISODE), its sequence number (SEQNBR) in the source episode, the text describing the decision (PROMPT) and the id of the target episode to jump to (TARGET, a foreign key to EPISODE). Branches are identified by the values of (EPID, SEQNBR).[5]

Script 14.1 shows the definition of the tables and Script 14.2 the loading of some representative objects of *TheWatchtower*.

The data have been extracted from the source text of the game and manually formatted into SQL statements. They have been remastered as follows:

– the text of some decisions has been adjusted to make them look more natural when they appear on the screen,

– artificial exit episode 0 has been added to collect all failure executions; similarly, exit episode 99 terminates all successful execution,

5. This representation is a faithful representation of the structure of the game as a graph, according to the mathematical definition of a graph as a set **V** of vertices (or nodes) and a set **E** of edges (or arcs), such that $E \subseteq \mathbf{V} \times \mathbf{V}$. Surprisingly, the textbooks and encyclopedia entries on *Graph theory* mention several data structures for representing graphs and developing algorithms, including the (elegant) incidence and adjacency matrices and the (awkward and irregular) incidence and adjacency lists, but never (as far as I know) the natural and efficient structure used in this study. This is unfortunate since it allows a more declarative expression of the main graph properties and algorithms.See [en.wikipedia.org/wiki/Graph_theory] for example.

```
create table EPISODE(
  EPID   integer not null primary key,
  TEXT   varchar(4096));
create table BRANCH(
  EPID   integer not null,
  SEQNBR integer not null,
  PROMPT varchar(256),
  TARGET integer not null,
  primary key (EPID,SEQNBR),
  foreign key (EPID)   references EPISODE on delete cascade,
  foreign key (TARGET) references EPISODE on delete cascade);
```

Script 14.1 - Data structures for the game

```
insert into EPISODE values (1,'\nIt is a truth of every
Ranger''s life that theirs'' must be a solitary existence.
...that it is time to begin.
\n\nWith the suns of Arborell ...
\n\nOutside the remains of the day flee westwards and with
their passing ... It is cold, it is dark, and without
hesitation you begin your mission.');
...
insert into EPISODE values (15,'\nMoving towards the table in
the south-west corner you find ... There is a Mutan Overseer
here and he will not be far from his experiments.');
...
insert into BRANCH values (15,1,'You have not already done so
and would begin a general search of this level',22);
...
insert into BRANCH values (15,4,'There is good cause to make
for the stairway to the battlements above',17);
...
```

Script 14.2 - Loading the description of episodes and branches

– episode id's have been transformed into purely numeric values: **?a** ⟶ **?01** and **?b** ⟶ **?02**; for example, **2b** is replaced by **202**,

– elementary formatting has been added, notably by insertion of *new lines* (`\n`).

The game engine displays the text of the current episode, followed by the list of possible decisions and lets the user select one of them. Then, the engine jumps to the selected episode and so on until an exit episode `0` or `99` has been reached.

The suggested dialogue box is shown in Figure 14.5. It is made up of two elementary boxes: a text window (`showText`) and a list of radio buttons (`selectOne`).

The player reads the text of the episode, selects one of the decisions then clicks on button OK, or Cancel to abort the game.

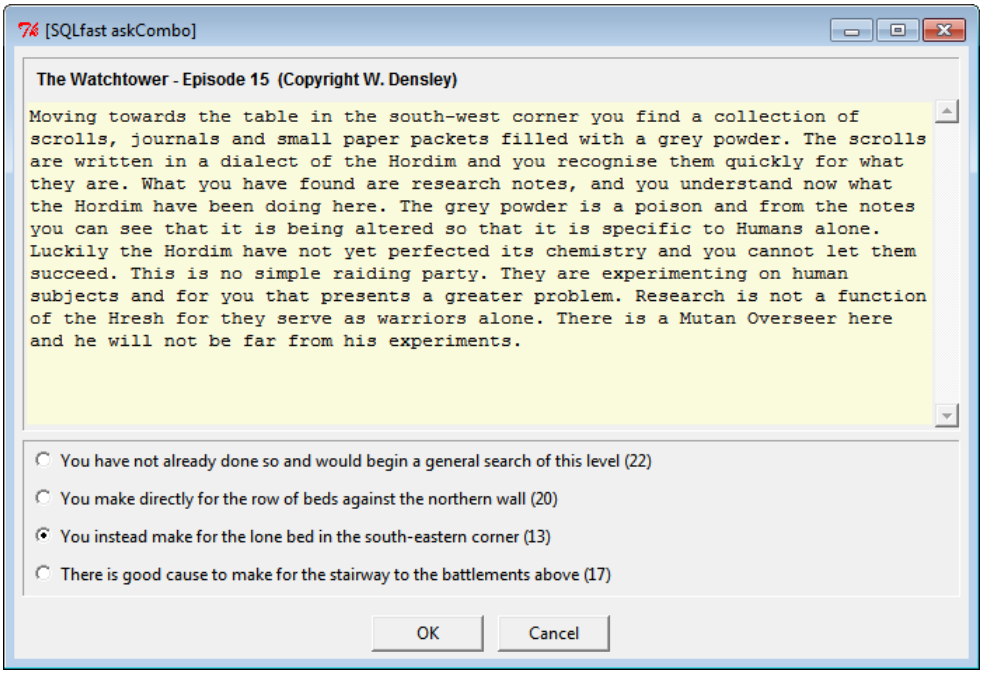


Figure 14.5 - The game dialogue box: visiting episode 15

An elementary game engine is proposed in Script 14.3. The current episode number is in variable E. The script extracts the text of the episode [1] and builds the list of decision items through a `group_concat` aggregation function [2].

```
set E = 1;
while ('$Ep$' <> '');
  extract txt = select TEXT from EPISODE where EPID = $E$;   [1]
  extract P   = select group_concat(PROMPT,'|') from BRANCH
                where  EPID = $E$ order by SEQNBR;           [2]
  askCombo [showText txt = [/b/x80/y14The Watchtower - Episode
                            $Ep$  (Copyright W. Densley)]]
           |[selectOne next = $P$];                          [3]
  if ('$DIALOGbutton$' = 'Cancel' or $Ep$ in (0,99)) exit;  [4]
   extract E = select TARGET from BRANCH
               where  EPID = $Ep$ and SEQNBR = $next$;       [5]
endwhile;
```

Script 14.3 - The game engine

Then, it displays the dialogue box [3]. It stops the game if requested to do so or if the current episode is an exit one (0 or 99) [4]. Finally, it extracts in variable E the id of the next episode [5], which becomes the new current episode. The loop of the engine comprises no more than **five statements**, among which **three SQL queries**.

This game is ready to play. Enjoy!

14.4 Story telling

Automatic novel writing is an old dream of computer scientists (those who have no talent for writing novels) and of lazy writers. Thanks to interactive game books, this dream comes true, practically at no cost.

When a game terminates, the sequence of the episodes visited forms a **story** driven by the decisions of the player. In fact, these decisions can be carried out by a script, which, for each episode, selects the output branch randomly.

For reasons we will analyze later, the probability that two runs would produce the same story is extremely low. So, each run of the game generates a new story. Running TheWatchtower game one thousand times probably will produce one thousand different stories!

Script 14.4 implements a *story generating procedure* that writes stories in external file A-TheWatchTower-Story.txt (to write in a file, the standard output is redirected to this file, through statement `outputOpen`).

```
outputOpen A-TheWatchTower-Story.txt;
set E = 1;
while ($E$ not in (0,99));
  extract txt = select TEXT from EPISODE where EPID = $E$;  [1]
  replace txt = \\n,@n;                                     [2]
  write $txt$;                                              [3]
  extract NB = select count(*) from BRANCH where EPID = $E$; [4]
  compute B = 1 + abs(random()) % $NB$;                     [5]
  extract B,PR,E = select SEQNBR,PROMPT,TARGET from BRANCH
                   where  EPID = $E$ and SEQNBR = $B$;      [6]
  if ('§PR§' <> 'Continue') write-b @n$PR$.;                [7]
endwhile;
outputAppend window;
```

Script 14.4 - The story generator

The current episode number is in variable E. The script extracts its text [1] in variable txt. In this text, new lines commands (**\n**)) are replaced by control characters **@n** [2], so that the **write** statement can manage them when writing in the external file [3]. The number of output branches of the current episode is computed and stored in

variable NB [4]. A random number in range [1,NB] is drawn and stored in variable B [5]. The branch of the current episode with sequence number B is extracted [6]. Its text, stored in variable PR, is written in the external file [7] if it is worth to (different from Continue). The generation is stopped when an exit episode (0 or 99) has been processed.

Just seven statements embedded in a loop for several million exciting stories![6]

The stories generated from *TheWatchtower* game look like this one (excerpt):

. . . .

The common area remains dark and quiet, the growing storm a rumbling backdrop that masks your footfalls upon the cold flagstones. In the shadows you consider what you should do next.

You would rather leave this floor and take to the stairs in the north-west corner.

Before you is a stairway that connects the Ground Floor to the First Level above. You listen intently but can hear nothing beyond the howl of the winds and the clatter of wooden shutters loose upon their mountings.

You would take these stairs and make for the First Level.

In the gloom of the First Level of the Watchtower you stand at the top of a stairway that leads down to the Ground Floor below. All about you is darkness, the First Floor veiled in shadows. It appears that this level served as a barracks, the eastern wall lined with a series of sleeping alcoves, the floor littered with broken furniture and discarded food. In the south-western corner you can also see the dark shadow of a stairway leading to the levels above.

You wish to explore this First Level.

Surveying the First Floor from the stairs you see evidence of a long occupation here. The Hordim have used the tower as a base of operations and it is an incursion that must be brought to an end. Outside a peal of thunder crashes against the stone walls about you and in its rolling aftermath you see a form rise from one of the sleeping alcoves. Immediately the Hresh sees you and starts to cry out, but its attempt at raising an alarm is smothered by another shuddering roll of thunder. Before the creature can fill its lungs with air again you charge it down, crashing into its side and splaying it bodily upon the floor. The warrior does not remain still. In one swift movement the Hresh regains its feet and attacks.

Clothed in a loose-fitting uniform and wielding a scimitar the Hordim closes upon you. The Hresh Warrior has a combat value of 14 and an endurance of 10. If you win this combat continue with your mission. If not, it will be here that your adventure will end.

You loose this combat.

Here ends your adventure.
It will be in a latter life that you will have to strive for greater success '

6. Let's remain modest: to generate *real novels* we should consider other aspects of the game, such as fighting (a random process) and acquiring resources!

14.5 Structure of a game

Beyond the fun of playing adventure games, there is the fun of analyzing and changing the structure of these games[7]. Basically, as we observed it above, games can be modeled as **graphs**, so that the concepts and techniques of graph theory can be applied, at least to some extent. However, we will mainly reason on game concepts instead of on abstract graph concepts. Let us first examine how games are structured.

14.5.1 Episodes, branches, paths and runs

The link from current episode E1 to episode E2, referenced by a decision of E1, is called a **branch**. E1 is the *source episode* of this branch while E2 is its *target episode*. E2 is a *successor* of E1 and E1 is a *predecessor* of E2.

Normally, the successors of an episode are distinct. However, some games may propose several branches to the same target episode for a source episode, as can be observed in WindhammeR. For the sake of simplicity, we will postulate the uniqueness hypothesis. If two decisions do lead to the same target episode, they are merged through an **or** logical connector.

The excerpt above shows that episode **15** is the source of four branches, the targets of which are episodes **22**, **20**, **13** and **17**. Episode **15** is also the target of five branches, namely from episode **17**, **20**, **22**, **26** and **201** (formerly **2a**).

The graph of Figure 14.6 depicts episode **15** with its predecessors and its successors. Let us call a **path** a suite of connected branches. For instance, the suite **26.15.13**, obtained by composing branch (**26**,**15**) with branch (**15**,**13**), is a path from episode **26** (the source of the path) to episode **13** (the target of the path). The **length** of a path is the number of branches it is made up of. The length of **26.15.13** is 2.

A **run** of the game is a path that starts at episode 1 and ends at exit episodes 0 (*fail*) or 99 (*win*).

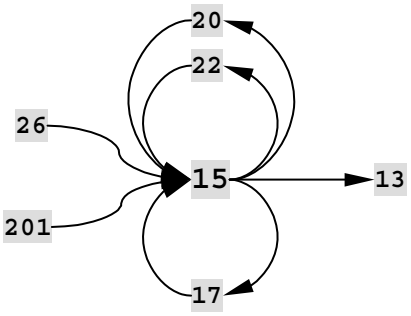


Figure 14.6 - Episode 15 together with its environment

7. Not to mention the fun of writing them.

14.5.2 Starting and ending episodes

A game book has one starting point, which is the first episode to visit. In *TheWatchtower* and *WindhammeR*, the starting point is episode 1. The game has at least one successful exit point and one failure exit point. They define the end of the game and the result of the run: *win* or *fail*.

14.5.3 Circuits

In most games, an episode can be visited more than once. There are several reasons for this:

- the episode is the source of several branches, each on them worth to be followed,
- some successors of an episode are made reachable only when the player has acquired some definite resources, such as a key or a special skill,
- the *mischief* of the game designer, who find it funny to lead the player to an episode already visited,
- or the player is a bit absent-minded or disoriented ...

A path the source and target of which are the same episode is called a **circuit**. Since there is no branch from an episode to itself, a circuit comprises at least two branches.

An episode can be visited several times in the same run if it appears in a circuit. Figure 14.6 makes it clear that *TheWatchtower* includes circuits. Actually, a circuit has no natural starting point, so that the paths E1.E2.E3, E2.E3.E1 and E3.E1.E2 all define the same circuit. However, we will find it convenient to designate one of its episode as its (arbitrary) starting point. This way, by choosing E1 as the starting point of the circuit, we can unambiguously denote the latter by path E1.E2.E3.

We could conclude from the observation of the graph of Figure 14.6 that it comprises three circuits of length 2, namely **15.20.15**, **15.22.15** and **15.17.15**.

However, it also comprises circuit **15.20.15.17.15** of length 4, as well as **15.20.15.20.15**. In fact, the graph includes an infinite number of circuits! As a consequence, there are quite a lot of paths (exactly an *infinity*) from episode **26** to episode **13**. Note that **26.15.20.15.13** is not a circuit but a path that includes a circuit.

We call **elementary** a circuit the episodes of which appear only once, except for its starting point. Circuit **15.20.15** is elementary while path **15.20.15.17.15** is not. The graph of Figure 14.6 includes three elementary circuits. We observe that an episode can be part of more than one circuit, as evidenced by episode **15**.

Two final observations: in a circuit, at least one episode must have two or more predecessors, otherwise the player would not be given the opportunity to enter the circuit! Similarly, at least one episode must have two or more successors, otherwise there would be no way to exit the circuit.

14.6 Analyzing a game

In this section, we will examine two properties of games: whether they are well-formed and whether they include circuits.

14.6.1 Searching for game abnormalities

A well-formed game must satisfy some structural properties. We will mention some of them, together with the SQL queries that can be used to check them.

a) The game has one and only one starting point

Checking: we count the episodes that have no predecessors. There should be only one.

```
select EPID,count(*) from EPISODE
where  EPID not in (select TARGET from BRANCH);
```

b) The game has at least one successful exit point

Checking: we count the branches the target of which is episode 99.

```
select count(*) from BRANCH where TARGET = 99;
```

c) The game has at least one failing exit point

Checking: we count the branches the target of which is episode 0.

```
select count(*) from BRANCH where TARGET = 0;
```

d) There is no isolated episode

Checking: every episode must appear in some branch.

```
select EPID from EPISODE
where  EPID not in (select EPID from BRANCH
                      union
                    select TARGET from BRANCH);
```

e) There is no circuit of length 1

Checking: we count the branches where the source is the target.

```
select count(*) from BRANCH where EPID = TARGET;
```

f) Every episode is reachable from the starting point

Checking: this one is a bit more complicated. We leave it for further investigation.

g) Each episode contributes to a run

Checking: same comment. We leave it for further investigation.

TheWatchtower appears to be devoid from these abnormalities.

14.6.2 Does the game include circuits?

We want to know whether the game includes circuits. If it doesn't, then it is called *acyclic*. It may be good news since then we are sure that we can finish the game in a finite number of steps, preventing us from spending the whole night to complete it. To cope with this question, we will first reason on two small games.

The first game is acyclic (Figure 14.7). It comprises 8 episodes, that cannot be visited more than once. Therefore, before examining the structure of the game, we are guaranteed that it can be completed in at most 7 steps (8 - 1, because only one exit episodes can be reached). Actually, a more thorough examination shows that it can be completed in 3, 4 or 5 steps.

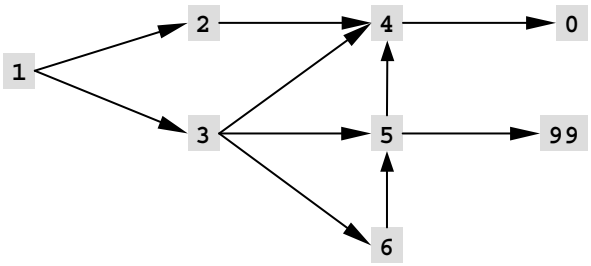


Figure 14.7 - An acyclic game

The second game, on the contrary, includes a circuit comprising episodes 2, 4 and 3 (Figure 14.8). The difference with acyclic games is that some episodes can be visited more than once. For this, they are called *cyclic episodes* and the game is said *cyclic* as well.

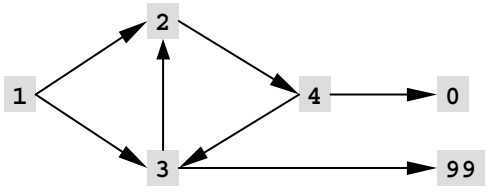


Figure 14.8 - A game including a circuit

Let us try to *prune* (i.e., to eliminate episodes from) the acyclic game as follows[8]:

– *step 1*: we remove all the entry episodes (if any), that is, episodes that have no predecessor

– *step 2*: if the remaining game still comprises entry episodes, we apply step 1.

Figure 14.9 shows what happens to acyclic game of Figure 14.7 when we apply this procedure. First, entry episode 1 is removed. Then, entry episodes 2 and 3 disappear, followed by episode 6, then 5, then 4 and 99. Finally, episode 0, which has become an entry episode, is removed. The result is an empty game.

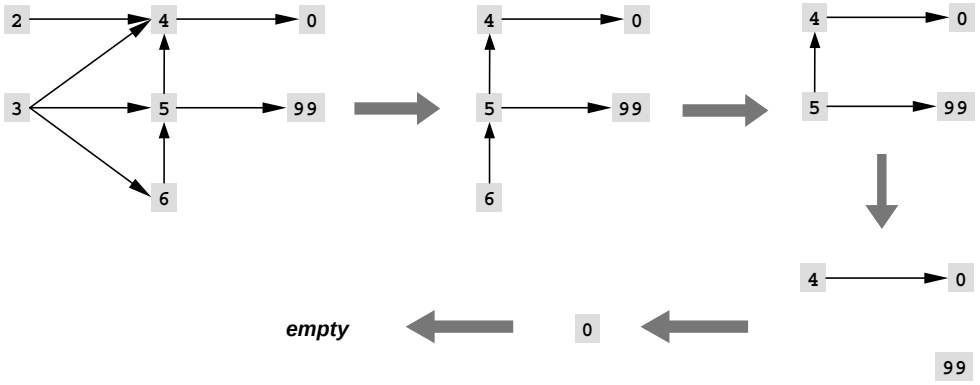


Figure 14.9 - Iteratively eliminating **entry** episodes until the game vanishes

The same result can be achieved by iteratively removing **exit** episodes. The following episodes are removed: first 0 and 999, then 4, then 2 and 5, then 6, then 3, and finally 1 to yield an empty game.

This experiment illustrates an interesting property of acyclic games: when we iteratively remove entry episodes, or exit episodes, or both, we finally remove all the episodes of the game.

How about applying this procedure to games with circuits? Let us try it with the game of Figure 14.8. First we remove episode 1, then ... things are blocked! Indeed, episodes 2 and 3 both have a predecessor and cannot be removed. Starting from the exit episodes does not help: we can remove episodes 0 and 99, but the remaining game now has no exit episode any more. Removing both entry and exit episodes yield the graph of Figure 14.10 (which is no longer a game since it has no entry nor exit episodes) which is cyclic.

8. This procedure is an application of *Marimont algorithm*, based on the property that a non empty graph that has no entry points and no exit points includes circuit(s).

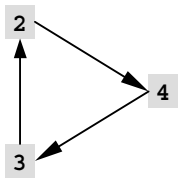


Figure 14.10 - Iteratively eliminating **entry** and **exit** episodes reduces the game to its cyclic episodes

This procedure is simple and intuitive. It allows us to discover whether a game is acyclic. Its SQL translation in particularly straightforward. The *entry episodes* of the game are defined by the following SQL expression,

```
select EPID from EPISODE
where  EPID not in (select TARGET from BRANCH)
```

and its *exit episodes* by,

```
select EPID from EPISODE
where  EPID not in (select EPID from BRANCH)
```

Therefore, the *entry* and *exit* episodes can be removed through these SQL queries:

```
delete from EPISODE
where  EPID not in (select TARGET from BRANCH);

delete from EPISODE
where  EPID not in (select EPID from BRANCH);
```

Hence the procedure of Script 14.5. The loop is controlled by variables **Nin** counting the *entry* episodes [1] and **Nex** counting the *exit* episodes [2]. When both of them have value zero, it is time to exit the loop [3]. The game is acyclic if BRANCH is empty, otherwise, this table contains all the branches of the circuits of the game. Applied to the game TheWatchtower, this procedure removes 9 episodes and 26 branches.

It is worth noticing that, though all the cyclic episodes belong to the remaining set, not all the remaining nodes can be qualified cyclic. Indeed a node that belongs to no circuit but that *bridges* two circuits cannot be eliminated and therefore belongs to the remaining set. In a further case study, *From data bulk loading to database book writing*, we will also exploit this elimination algorithm. This set of remaining nodes will be called the **cyclic kernel** of the graph.

```
while (True);
   extract Nin = select count(*) from BRANCH
           where EPID not in (select TARGET from BRANCH);   [1]
   extract Nex = select count(*) from BRANCH
           where TARGET not in (select EPID from BRANCH);   [2]
   if ($Nin$ = 0 and $Nex$ = 0) exit;                       [3]
   delete from EPISODE
   where  EPID not in (select TARGET from BRANCH);
   delete from EPISODE
   where  EPID not in (select EPID from BRANCH);
endwhile;
```

Script 14.5 - Reducing a game to its cyclic kernel

14.7 Extracting circuits

Identifying all the circuits of a game may appear a non trivial task. Actually, it involves some interesting challenges but, once carefully analyzed, step by step, it proves quite tractable. The basic idea is simple:

> *we build all the paths of the game by iteratively composing branches then we select those the first and last episodes of which are identical.*

This is a nice example of recursive CTE, expressed in Script 14.6. To simplify the construction, we represent a path by a sequence of episode id's separated by dots as we did before, but we also add a leading dot and a trailing dot so that each id is surrounded with dots. In addition, we consider that episode id's are character strings of length 2 (or more for very large games).[9]

The initialization query inserts in CTE PATHS columns EPID and TARGET of the BRANCH rows and builds from them the first element of the paths. The recursive query extends the paths in PATHS with the rows of BRANCH that match on PATHS.TARGET = BRANCH.EPID. They also add a new branch to existing paths. The paths built in this way are stored in table CIRCUIT defined as

```
CIRCUIT(Circuit)
```

Actually, this query does not work, and this for several reasons.

9. Otherwise, we would have to transform integers into fixed length strings through expression `cast(EPID as char(2))`, or, in SQLfast `frame(cast(EPID as char),2,'>','0')`. This would inevitably obscure the queries. The scripts of game analysis associated with this study use this convention.

```
with recursive PATHS(EPID,TARGET,Path) as
  (select EPID,TARGET,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH
      union
   select P.EPID,B.TARGET,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   )
insert into CIRCUIT
select Path from PATHS where EPID = TARGET;
```

Script 14.6 - Extracting the circuits of a game - First attempt

First, the graph of the game can be so large that computing the set of possible paths would take hours, if not days. So we build a safety mechanism that limits the length of the paths through CTE column Length. Its value is initialized to 1 and incremented by the recursive query. We also add column Length to CIRCUIT.

Secondly, when a circuit has been built in PATHS, new paths will be built by extending it with additional branches, which is useless. So, we freeze such paths by allowing extension of paths only if their first and last episodes are different.

Finally, since we are interested in *elementary circuits* only, we do not allow paths to be extended when they already include the node we intend to add, except of course when the latter is also the first one (in which case we are building a circuit). So, we add a constraint like this one:

```
P.Path not like '.%.'||B.TARGET||'.%'
```

What does this condition says? P.Path is the current path we intend to extend. B.TARGET is the candidate new episode. This episode may not appear in P.Path:

```
P.Path not like '%.'||B.TARGET||'.%'
```

except if it appears in the first position. So, to discard this case, we add a dot in front of the pattern to force the condition to ignore the first episode of the path:

```
'.%.'||B.TARGET||'.%'
```

The improved version of the algorithm is shown in Script 14.7.

Now, table CIRCUIT includes all the elementary circuits of the game. However, it also includes all the **permutations** of these circuits. Applied to the graph of Figure 14.8, that includes a single circuit, the query produces the content shown in Figure 14.11.

```
set len = 16;
with recursive PATHS(EPID,TARGET,Length,Path) as
  (select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH
      union
   select P.EPID,B.TARGET,P.Length+1,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   and    P.EPID <> P.TARGET
   and    P.Path not like '.%.'||B.TARGET||'.%'
   and    P.Length < $len$
   )
insert into CIRCUIT
select Path,Length from PATHS where EPID = TARGET;
```

Script 14.7 - Extracting the circuits of a game - Next try

```
+---------------+--------+
| Path          | Length |
+---------------+--------+
| .02.04.03.02. | 3      |
| .04.03.02.04. | 3      |
| .03.02.04.03. | 3      |
+---------------+--------+
```

Figure 14.11 - Permutations of the same circuit

So, the next problem we have to solve is to reduce all the permutations of a circuit to a single one. This does not seem obvious at first glance. Fortunately, this problem is a special case of a common, more general problem that we may encountered in many other situations: *comparing sets*. Indeed, the common property of these permutations is that they are defined on the same **set** of episodes.

Since SQL basically is a *set-oriented language*, we could express set comparison by an SQL query, for instance by checking that each element of the first one belongs to the second one and that they have the same size. However, such a query would be terribly inefficient in this situation, in which we have to compare hundreds of millions of pairs of sets. We will develop another technique based on a character string derived from the path and that uniquely denotes the set of its episodes, for instance this one, in which the episode id's are sorted:

```
02.03.04
```

This character string is common to all the permutations of a circuit, so that we can gather them with a simple `group by` and select, say, the permutation with the lowest rank in the alphanumeric order (`min(Path)`), that is, here, `.02.04.03.02.`. We suggest to add column EpSet (episode set) to table CIRCUIT, in which we will store the denotation of the set of episodes of each path:

```
+---------------+--------+----------+
| Path          | Length | EpSet    |
+---------------+--------+----------+
| .02.04.03.02. | 3      | 02.03.04 |
| .04.03.02.04. | 3      | 02.03.04 |
| .03.02.04.03. | 3      | 02.03.04 |
+---------------+--------+----------+
```

There is a last problem to solve: building this set denotation. We will derive it from the expression of the circuit. First, we remove the first and last dots:

```
trim(Path,'.')
```

Now, we have a dot-separated list of episode id's. We sort this list with SQLfast function itemSort. This function requires four parameters: the list to sort, the sorting direction (0 = ascending), the uniqueness (0 = duplicates allowed; 1 = duplicates discarded) and the separator:

```
itemSort(trim(Path,'.'),0,1,'.')
```

The completed code is shown in Script 14.8 and its result in Figure 14.12.

```
set len = 16;
with recursive PATHS(EPID,TARGET,Length,Path) as
  (select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH
      union
   select P.EPID,B.TARGET,P.Length+1,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   and    P.EPID <> P.TARGET
   and    P.Path not like '.%.'||B.TARGET||'.%'
   and    P.Length < $len$
   )
insert into CIRCUIT
select min(Path),Length,
       itemSort(trim(Path,'.'),0,1,'.') as EpSet
from   PATHS
where  EPID = TARGET group by EpSet;
```

Script 14.8 - Extracting the circuits of a game - Final version

Circuit	Length	EpSet
.06.18.06.	2	06.18
.06.21.06.	2	06.21
.06.28.06.	2	06.28
.06.30.06.	2	06.30
...	...	...
.02.24.29.02.	3	02.24.29
.02.25.29.02.	3	02.25.29
.02.27.29.02.	3	02.27.29
.03.09.14.03.	3	03.09.14
...	...	...
.02.12.24.29.02.	4	02.12.24.29
.02.24.25.29.02.	4	02.24.25.29
.02.24.27.29.02.	4	02.24.27.29
.02.24.34.29.02.	4	02.24.29.34
...	...	...
.02.12.24.25.29.02.	5	02.12.24.25.29
.02.12.24.27.29.02.	5	02.12.24.27.29
.02.12.24.34.29.02.	5	02.12.24.29.34
.02.24.25.27.29.02.	5	02.24.25.27.29
...	...	...
.02.12.24.25.27.29.02.	6	02.12.24.25.27.29
.02.12.24.25.34.29.02.	6	02.12.24.25.29.34
.02.12.24.27.23.29.02.	6	02.12.23.24.27.29
.02.12.24.34.27.29.02.	6	02.12.24.27.29.34
...	...	...
.02.12.07.11.37.34.29.02.	7	02.07.11.12.29.34.37
.02.12.24.25.27.23.29.02.	7	02.12.23.24.25.27.29
.02.12.24.25.34.27.29.02.	7	02.12.24.25.27.29.34
.02.12.24.34.27.23.29.02.	7	02.12.23.24.27.29.34
...	...	...

Figure 14.12 - Table CIRCUIT: elementary circuits of game TheWatchTower

The Watchtower game comprises 437 elementary circuits made up of 2 to 15 branches.

14.8 Counting runs

One of the important questions about a new game is whether the player would like to play it again. This refers to the variety of the runs, that is, to the number of the different paths starting at episode 1 and reaching exit episode 0 (fail) or 99 (win).

To answer this question for a definite game, we generate all the paths starting at episode 1, then we store the runs among them in new table RUN, defined as follows:

```
RUN(EXIT,Length,Path)
```

Column EXIT specifies the result of the run: *win* (99) or *fail* (0), Length the number of branches of the run and Path the sequence of episodes.

The general structure of the query that computes the runs (Script 14.9) is similar to that of Script 14.8. The main difference is that we initialize the process with the branches starting from episode 01 only.

A run can be *acyclic*, in which case all the episodes are distinct, or *cyclic* if the run includes one or several circuits. If we allow unconstrained runs, allowing them to follow any number of circuits, then the answer to the counting problem is clear and inexpensive: *infinite*.

So, we must control the number of circuits allowed in the runs computed by the query. The control mechanism is a condition symbolically called **<cyclic>** in Script 14.9.

```
set Len = 20;
with recursive PATHS(EPID,TARGET,Length,Path) as
  (select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH where EPID = '01'
      union
   select P.EPID,B.TARGET,P.Length+1,P.Path||B.TARGET||'.'
   from   PATHS P, BRANCH B
   where  P.TARGET = B.EPID
   and    P.EPID <> P.TARGET
   and    <cyclic>
   and    P.Length < $Len$
   )
insert into RUN
select TARGET,Length,Path
from   PATHS where TARGET in ('00','99');
```

Script 14.9 - Computing the runs of a game

This condition is similar to that coded in Script 14.8. For linear runs, it tells that the path we intend to extend *may not include the new episode*, i.e.,

```
P.Path not like '%.'||B.TARGET||'.%'
```

For cyclic runs, we specify the number of episode instances allowed in runs. For instance, if we allow **one instance** of each elementary circuit,[10] the pattern becomes:

```
'%.'||B.TARGET||'.%.'||B.TARGET||'.%'
```

... for **two instances**,

```
'%.'||B.TARGET||'.%.'||B.TARGET||'.%.'||B.TARGET||'.%'
```

10. This is not quite exact. When a circuit has been followed, all the other circuits that share some episode with it will not be followed.

... and so on.

Linear runs

Let us first examine the result of this path generation procedure applied to the small acyclic game of Figure 14.7. It identifies 6 runs, 2 of which are successful and the other 4 fail (Figure 14.13).

EXIT	Length	Path
00	3	.01.02.04.00.
00	3	.01.03.04.00.
99	3	.01.03.05.99.
00	4	.01.03.05.04.00.
99	4	.01.03.06.05.99.
00	5	.01.03.06.05.04.00.

Figure 14.13 - The runs of acyclic game of Figure 14.7

Among the linear runs of the cyclic game of Figure 14.8, 2 are successful and 2 fail (Figure 14.14).

EXIT	Length	Path
99	2	.01.03.99.
00	3	.01.02.04.00.
00	4	.01.03.02.04.00.
99	4	.01.02.04.03.99.

Figure 14.14 - The linear runs of cyclic game of Figure 14.8

Cyclic runs

Now, let us allow runs to include at most **one instance** of each circuit. The application of the procedure to the cyclic game of Figure 14.8 generates the eight runs of Figure 14.14.

EXIT	Length	Path
99	2	.01.03.99.
00	3	.01.02.04.00.
00	4	.01.03.02.04.00.
99	4	.01.02.04.03.99.
99	5	.01.03.02.04.03.99.
00	6	.01.02.04.03.02.04.00.
00	7	.01.03.02.04.03.02.04.00.
99	7	.01.02.04.03.02.04.03.99.

Figure 14.15 - Acyclic and single cyclic runs of cyclic game of Figure 14.8

Analysis of TheWatchTower

Now, we consider a real but small game, *TheWatchTower*. Computing all the *linear paths* from episode 01, be they complete (i.e., runs) or incomplete, gives impressive results:

- 1,004,389,004 paths, that is, more than one billion.
- 213,508,282 runs
- 126,344,842 failing runs
- 87,163,440 successful runs
- the shortest run comprises 3 episodes (01.28.00)
- the longest run is made up of 38 episodes; there are 46,080 of them
- the most frequent run length (26,947,948) is 30 episodes.

One can guess that this result comes at a price: **more than 9 hours** of computing at full speed.[11] To get faster but more modest data we must limit the maximum length of the runs. The numbers of Figure 14.16 have been computed for runs from 2 to 20 branches. Failing runs are much more frequent than winning ones. An unfair habit of most games! One can fail after just two episodes but to win, one must visit at least 13 episodes.

Length	Win	Fail
2	0	1
3	0	3
4	0	11
5	0	31
...	...	...
12	0	9,341
13	1	22,567
14	13	51,870
15	95	113,343
16	514	236,366
17	2,281	471,998
18	8,712	905,093
19	29,367	1,670,646
20	88,753	2,975,483

Figure 14.16 - The runs of TheWatchTower, from 2 to 20 episodes

If you can complete a game in an average of 10 minutes, then *TheWatchtower* can give you 20,218 **years** of pleasure! Full time.

However, we are not done yet. So far, we have allowed *linear games* only, which is fairly unrealistic. We can use Script 14.9, allowing episodes to be visited twice,

11. Stored in a 105 GB database

three or four times (several episodes even have 5 or 6 successors). This will augment the number of runs considerably.

14.9 Are all episodes reachable?

We have left this question unanswered in Section 14.6.1 because we had not yet developed the necessary algorithm to solve it. Now we are ready to address this issue.

Basically, an episode is **reachable** if it belongs to at least one path (not necessarily a run) starting from episode 01. A first idea would be to identify the episodes that have no predecessor, that is, that do not appear as a target in any branch:

```
select EPID from EPISODE
where  EPID not in (select TARGET from BRANCH);
```

However, this query will not catch all unreachable episodes. For instance, all the episodes of a circuit *with no entry point* are targets but they are unreachable.

Then, we could adapt the algorithm of Script 14.9 a little bit, so that it generates all the paths of the game (that start at episode 1) and not only its runs. The TARGET components of these paths form the set of the reachable episodes. Therefore, the elements of EPISODE that are not in this set are not reachable and constitute abnormal episodes. Unfortunately, this technique is very expensive: it generates more than *one billion paths* in about *9 hours* for a game that comprises 39 episodes only. So, we must find a far more efficient algorithm.

When the CTE of Script 14.9 builds a path of the game, its last episode is a *reachable episode*. We can stop the extension of any other path with it since this would bring us information we already know. Practically, if episode E is known to be reachable, we can delete all the branches where TARGET = E.

Script 14.10 translates this idea. Since we will modify data in the process, we cannot express it through a CTE. So, we will simulate it with a loop as we already did in Section 12.2 of case study *Kings of France - Part 2*.

Table PATH, that will contain the paths to build, is initialized with the branches starting from episode 1. The next set of paths will be stored in table LAST which has the same structure as PATH. Table BBRANCH is a copy of BRANCH, the rows of which will be progressively deleted. The algorithm works as follows.

– Variable NE stores the number of episodes, which is the maximum number of iterations.

– Each iteration considers the *last paths* inserted in PATH (those with length = $length$) and extends them with the matching rows from BBRANCH. The extended paths are stored in table LAST then moved (`insert` + `delete`) to PATH.

```
insert into PATH(START,END,Length,Path)
   select EPID,TARGET,1,'.'||EPID||'.'||TARGET||'.'
   from   BRANCH where EPID = '001';
create table LAST as select * from PATH where 1 = 2;
insert into RBRANCH
   select EPID,TARGET from BRANCH where EPID <> '001';
extract NE = select count(*)-1 from EPISODE;
for length = [1,$NE$];
   insert into LAST
      select P.START, B.TARGET,
             $length$ + 1, P.Path||B.TARGET||'.'
      from   PATH P, RBRANCH B
      where  P.END = B.EPID
      and    P.Length = $length$
      and    P.Path not like '%.'||B.TARGET||'.%';
   extract N = select count(*) from LAST;
   if ($N$ = 0) exit;
   insert into PATH select * from LAST;
   delete from LAST;
   delete from RBRANCH where TARGET in (select END from PATH);
endfor;
```

Script 14.10 - Finding all the reachable episodes

– *Critical step*: at the end of each iteration, the rows of BBRANCH, the target of which is one of the episodes just added, are deleted.

– The reachable episodes are in column END of PATH:

```
select distinct END from PATH
```

The episodes of EPISODE that are not in PATH are unreachable. If the game is consistent, only episode 1 is unreachable!

The performances of this modified algorithm are very impressive. For game TheWatchTower (41 episode, 136 branches), table PATH is filled with 58 paths[12] in **0.02 s**. For WindhammeR (599 episodes, 993 branches), 64,289 paths have been generated in **0.92 s**.

12. We could wonder why this number is higher than that of episodes. Indeed, though no iteration identifies an episode that has already been identified by preceding iterations, the same episode may be identified more than once within the same iteration.

14.10 Do all episodes contribute to a solution?

An episode is said to *potentially contribute to a run* if it appears in at least one run. If it doesn't, even though it is reachable, then it belongs to a dead-end path and is useless. Just like the reachability problem, this one can be solved by a standard, but very expensive, algorithm based on the set of all the paths of the game. However, we can transform it into a known problem, by observing that this question is the inverse of the reachability problem. If we **inverse all the branches**, swapping EPID and TARGET values, we just have to prove that all episodes can be reached from episode 0 or episode 99.

14.11 Merging episodes

When an episode **E1** has only one successor **E2**, and if **E1** is the only predecessor of **E2**, we can think of simplifying the game by merging **E1** and **E2** into single episode **E12**. The text of **E12** is the text of **E1** followed by the text of **E2,** the predecessors of **E12** are those of **E1** and the successors of **E12** are those of **E2**. We call branch (**E1**,**E2**) *one-to-one*.

Figure 14.17 shows how episodes **7** and **11**, connected by *one-to-one* branch (**7**,**11**), are merged to form new episode **711**.

We discard artificial exit episodes **0** and **99** from this restructuring. In addition, **E1** cannot be the target of **E2**, otherwise, the merging would create a *reflexive* branch (here, from **711** to itself).

TheWatchtower includes 3 such patterns: (**7**,**11**), (**13**,**31**), and (**39**,**19**). They have been identified by Script 14.11, that works as follows. Episodes **E1** and **E2**, linked by a branch, can be merged if:

- **E2** is not an exit episode [1],
- **E1** has only one successor [2],
- **E2** has only one predecessor [3],
- **E1** is not a target of **E2** [4].

Merging these couples reduces the size of the game by 3 episodes and 3 branches.

In WindhammeR, 93 couples to merge have been found, among them 24 form 12 double chains (two successive one-to-one branches), 6 form 2 triple chains and 8 form 2 quadruple chains.

The writing of the transformation algorithm is not particularly difficult. It is left as an exercise.

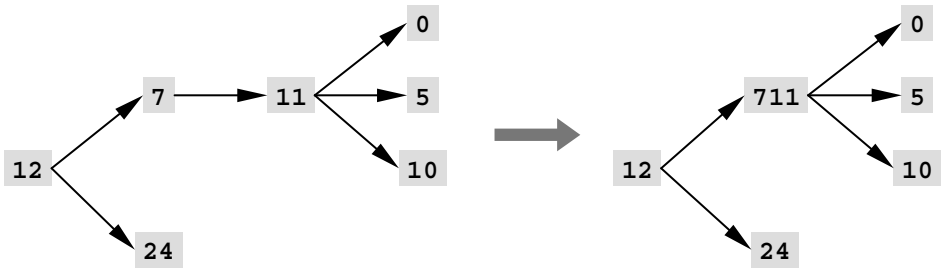


Figure 14.17 - Merging episodes 7 and 11

```
select EPID,TARGET
from   BRANCH B
where  TARGET not in ('00','99')                                    [1]
and    (select count(*) from BRANCH
        where  EPID = B.EPID) = 1                                   [2]
and    (select count(*) from BRANCH
        where  TARGET = B.TARGET) = 1                               [3]
and    (TARGET,EPID) not in (select EPID,TARGET from BRANCH); [4]
```

Script 14.11 - Identifying episodes that can be merged

14.12 Toward a better game engine

The game engine we have built is a very basic prototype. It can be improved in several ways. Let us begin with some goodies easy to develop.

- The database stores one game at a time. It would be nice to allow it to contain several games and to let the user choose which one she wants to play with. *The* user? Why not several users?
- There is no way to suspend the execution of a game, then to resume it later. This requires a *game saving* function, that saves the state of the game (aka a *save point*) and that restores it when the player wants to continue the game. A system allowing several saves of the game would be better.
- Why not show pictures or drawings in the episode display box (as in Figure 14.2)?
- Combat resolution requires the player to roll dices to determine who, among the player and the enemy, wins the combat. Dice rolling is simple to implement through SQLfast random() function.

– A game recording function would allow the player to replay, to undo (roll-back), to examine or to share a run.

– Some game books include an important descriptive part related to the world in which the game takes place including the actors, the arms, the resources, the language, the society rules and the history of this world. Large game books sometimes comprise a glossary of the specific terms. These resources can easily be implemented as help documents that can be opened from the dialogue box of the game.

Some improvements would require more effort but will make the games more fun to play.

– The player, and sometimes other characters of the game, can be assigned specific attributes or resources: strength, agility, endurance, weapons, health, energy, money, etc. These attributes generally are used to decide the way the player behaves when facing some problematic situations, such as combat, hunger or wound. These attributes can evolve: decreasing when used and increased when resources are encountered in some episodes (the ubiquitous *health packs* for example).

– What about a map of the episodes already visited?

– When the player can take objects found in some episodes, then an inventory system is required. These objects can be used to obtain some desired results, such as using a key to open a door or giving money to buy food. Figure 14.3 shows a nice way, easy to implement with two `selectOne` elementary boxes, to let the player perform an action on an object of the inventory.

– When the game model gets richer, storing its definition in the database through SQL `insert` queries may become more tricky. Why not design a GDL (*game definition language*)? And, of course, the program that translates it into SQL `insert` queries!

14.13 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Game_Book**. Actually, they can be run from main script **GameBook-MAIN.sql**, that displays the selection box of Figure 14.18.

Figure 14.19 shows the dialogue box that sets the parameters of the analysis of a game.

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

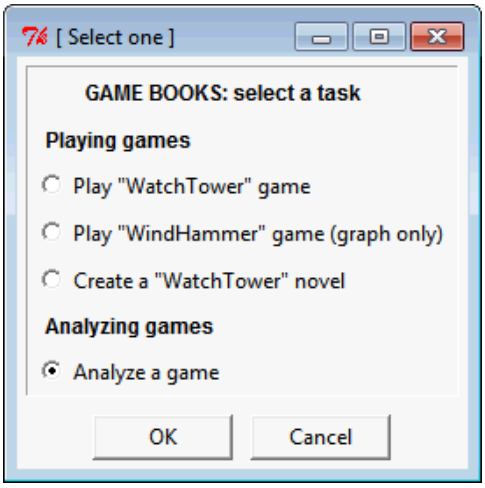


Figure 14.18 - Selecting an activity

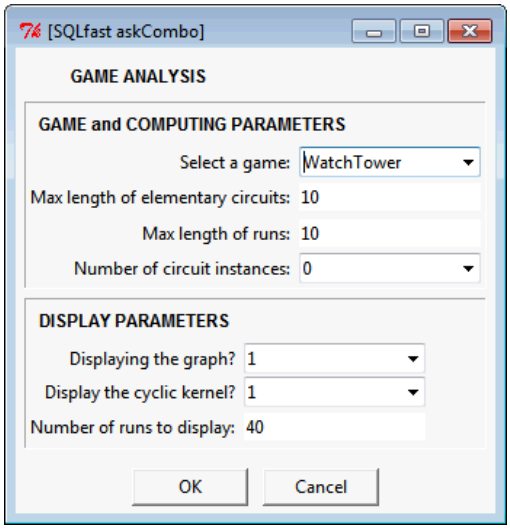


Figure 14.19 - Setting the parameters for game analysis

Case study **15**

Directory management

Objective: The contents of storage media, such as hard disks and flash disks, both internal and external, are organized into a hierarchical structure made up of directories and files.

This chapter shows that, when such structures are stored in a database, processes can be designed easily to examine directories, to analyze their contents, to describe their evolution and to discover potential problems. In particular, small applications will be developed to extract statistics, to display the structure and contents of a directory, do identify and describe potentially duplicate files and directories within a root directory or between two directories.

The problem of fast clone detection, that is, of set of files that have *exactly* the same contents, is also analyzed and solved.

Keywords: directory structure, tree modeling, tree analysis, statistics, tree evolution, duplicate files, clone detection, secure hashing, SHA256, database performance, CTE, recursive queries

15.1 Of files and directories

In this chapter, we will deal with the way the contents of storage media (internal/ external disks and Flash memory for example) are organized. Each of these media provides a space in which *files* of any kind are permanently stored. When their number are getting larger, we find it convenient to collect these files in *directories* according to some logical rules. A directory generally contains files but may also contain directories (of which the former is the *parent*), that, in turn, may contain other files and directories. The storage medium can be seen as a special directory.

Directory or folder?

Directories, the kind of container we just discussed about may also be called *folders*. Actually, Unix and Unix-like taxonomies favor the abstract term *directory* (as well as MS-DOS) while Windows and Mac OS have long introduced the more practical term *folder*. In this chapter, the terms *folder* and *directory* will be used interchangeably.[1]

Seen as a whole, the contents of a directory is structured as a *tree*, of which the latter is the root. Each node is either a directory or a file. Each edge represents the inclusion relation between a directory and each of its children node. The root of the tree is always a directory and its leaves are either files or empty directories.

Figure 15.1 shows a typical aspect of the standard file explorer of MS Windows. File DIRECTORY.sql is selected in folder Chapter-48. This folder is located in parent folder SQLfast-Tutorials, itself contained in folder Scripts of parent folder SQLfast-std, at the root of internal disk D:/. The graphical representation of the left panel shows clearly the tree structure of the directories.

Character string 'DIRECTORY.sql' is the *local name* of this file, that is, its name within its immediate parent folder. Its *full name* also comprises the names of all its direct and indirect parents:

```
D:/SQLfast-std/Scripts/SQLfast-Tutorials/Chapter-48/DIRECTORY.sql
```

Files have some interesting properties, such as their *extension*, their *size*, the time they were *last updated* or *last accessed*.

15.2 Examining directories

Directories surely are a convenient and intuitive way to organize the many files we store on our disk. However, after a while, they also become a real headache when the number and the variety of these files are getting unmanageable. Files are modified, their successive versions are kept, sometimes saved in more than one copy, quite often scattered across one or several disks. Directories and files are not always

1. https://msdn.microsoft.com/en-us/library/bb513869.aspx

named in a pertinent way, so that these names may give little hints about their contents.

Hence the need to better control the collection of our files distributed among the internal hard disk(s) and all these external hard disks and USB flash keys. In particular, displaying and analyzing their contents is the first task, for which we will develop some useful but simple tools.

Since we are going to examine and query trees, we could just reuse the scripts we created for the *Kings of France* application (Chapters 35 and 36). However, directory trees have quite different properties and suggest specific processes, that make it worth to develop a new approach.

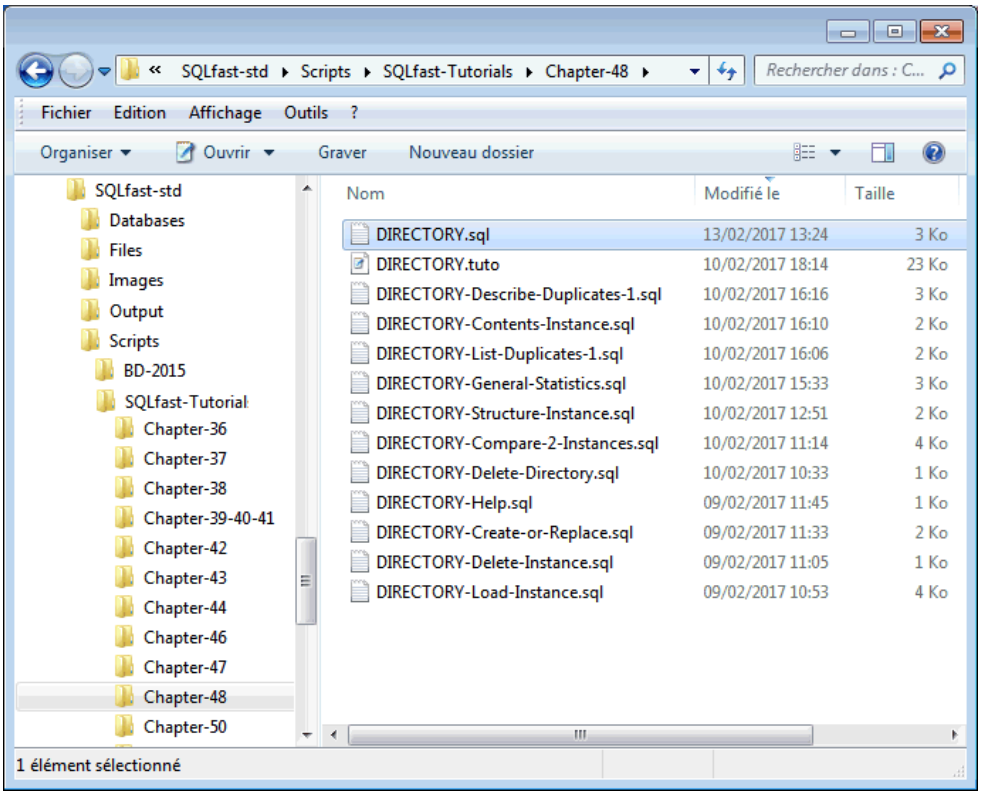


Figure 15.1 - Screen shot of Windows explorer

15.3 Modeling the problem

We intend to store in a database the description of the nodes of a directory in order to query it in various ways. However, further examination of the problem shows that some of its aspects may be more complex than thought at first glance. In particular,

1. considered statically, a directory may contain duplicate files (and directories)
2. directories generally evolve in time: files are updated, deleted, created, renamed, moved from one place to another one and duplicated
3. file duplication also appears among several directories, particularly if we create backup versions on different devices.

Addressing these spatial and temporal aspects of directories leads us to identify three main concepts:

1. *Directory*: a directory is uniquely identified by its name throughout all the devices we intend to manage. This concept is time independent: directory D:/SQLfast on our internal hard disk has been created several years ago and still exists, though with different contents. To distinguish these directories from those found in their composition, we call them *root directories*.
2. *Instance of directory*: the state of a root directory at a definite point in time (a *date-time*); in other words, a *snapshot* of a directory. All the instances of a directory have distinct date-time.
3. *Node*: the state of an element, either a directory or a file, in an instance of a directory. A node belongs to one instance. Therefore, a file that has been captured in two instances is represented by two different nodes. Deciding whether two nodes describe the same file must be done based on their properties (name, size, last update time) and position in the tree of their instances.

Let us illustrate and refine these concepts with an example.

We want to describe two root directories: **SQLfast** and **SQLfast-std**, located at the root of partition D of our internal hard disk. We decide to name them Local-D:/SQLfast and Local-D:/SQLfast-std. Why not just D:/SQLfast? Because this name alone may prove ambiguous if, later, we decide to record the contents of another computer in which a directory named D:/SQLfast also exists. So, we distinguish the real system name of a root directory (D:/SQLfast) from its *nickname* we assign to it to resolve potential name ambiguities (Local-D:/SQLfast). Nicknames are unique in the scope we define while system name are may not be.

We record the state of SQLfast-std on January 6th (twice), 9th and 10th, therefore creating four instances of it. Same for SQLfast, to which three instances are associated.

For each instance, we record (the description of) all the directories and files that composed its root directory at the time the instance was captured. These directories (root included) and files constitute the nodes of this instance. Two successive instances of a directory are likely to have a different composition.

15.4 Data structures

The implementation of these three concepts is straightforward. Root directories are recorded in table DIRECTORY, their instances are recorded in table INSTANCE and their components in table NODE. In each table, the rows are assigned a unique arbitrary number.

Table DIRECTORY records the system name (as their full path) of root directories (DirName), their unique nicknames (DirNick) and assigns them a technical primary key (DirID).

Table INSTANCE records all the instances of each root directory (column DirID) at a time point (InstDate and InstTime, in ISO format), down to the second. Each instance also is assigned a technical primary key (InstID).

Table NODE records the description of all the components of each instance (column InstID). A component is either a directory (NType = 'D') or a file (NType = 'F'). The root directory is itself a component of all its instances. Each node is assigned a unique Id (NodeID).

Each node has a parent node (ParentID), which is the directory node in which it is located, except root directories, which have no parent (this is their distinctive property).

A node has a full name (FullName), that is, the unique path starting from the device to its local name. The local name of a node (LocalName) is its name within its immediate parent directory. The prefix of a node (Prefix) is the full name of its immediate parent directory. Local name and prefix derive from the decomposition the full name (FullName).

Four properties are specific to file nodes:

- their extension, derived from their local name (Extension)
- their size, in Bytes (Size)
- the last time they were updated (LastUpdate)
- the last time they were accessed without modification (LastAccess)

Two derived properties synthesize the overall structure of instances:

- the level of a node within its instance tree (Level); root directory nodes are assigned level 0 and the level of a node is that of its parent + 1
- the suite of nodes from the root directory to each node, those included (Path)

Column Hash, that will be useful to identify clone files, will be explained later. These data structures are translated in SQL-DDL in Script 15.1.

Figures 15.2 and 15.3 show excerpts of the data that represent the example described above.

- Table DIRECTORY stores (the description of) root directories D:/SQLfast (nickname Local-D:/SQLfast) and D:/SQLfast-std (nickname Local-D:/SQLfast-std).

– Table INSTANCE stores three instances of Local-D:/SQLfast and four instances of Local-D:/SQLfast-std.

– The contents of table NODE is fairly large, so that one node only is shown in a `showData` box (Figure 15.3).

```
create table DIRECTORY(
       DirID      integer not null primary key autoincrement,
       DirName    varchar(1024) not null),
       DirNick    varchar(1024) not null unique);

create table INSTANCE(
       InstID     integer not null primary key autoincrement,
       DirID      integer not null
                  references DIRECTORY on delete cascade,
       InstDate   date not null,
       InstTime   time not null,
       unique (DirID,InstDate,InstTime));

create table NODE(
       NodeID     integer not null primary key autoincrement,
       InstID     integer not null
                  references INSTANCE on delete cascade,
       NType      char(1) not null,
       ParentID   integer
                  references NODE on delete set null,
       FullName   varchar(2048),
       Prefix     varchar(2048),
       LocalName  varchar(256),
       Extension  varchar(32),
       Size       integer,
       LastUpdate char(19),
       LastAccess char(19),
       Path       varchar(2048),
       Level      integer,
       Hash       char(64);
```

Script 15.1 - The tables of database DIRECTORY.db

This node describes a file (NType = F) named DIRECTORY.sql (LocalName), located in directory D:/SQLfast-std/Scripts/SQLfast-Tutorials/Chapter-48 (Prefix). Its node Id (NodeID) is 15813 and it belongs to instance 8 of root directory D:SQLfast (ParentID).
It is at level 4 from its root directory (column Level) and it can be reached (column Path) from the root directory node (NodeID = 14400) by following intermediate nodes 14897, 15163 and 15801. The prefix letter, D or F, is a reminder of the nature of the node. Column Hash is not shown.

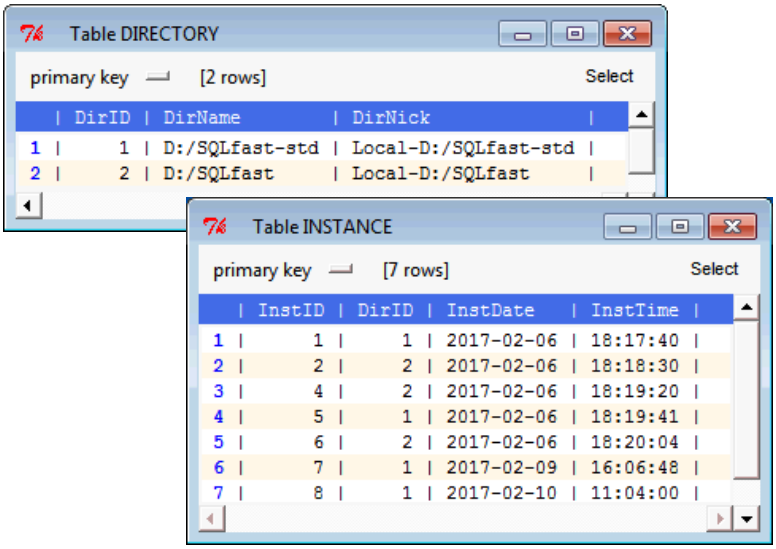


Figure 15.2 - DIRECTORY and INSTANCE tables

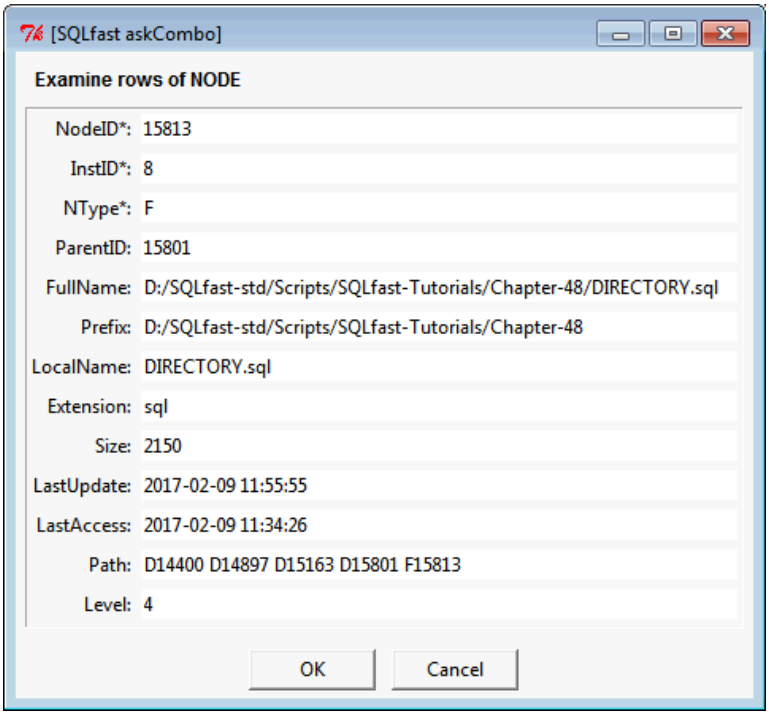


Figure 15.3 - Composition of a node (here script file DIRECTORY.sql)

15.5 Loading a directory instance

Statement `dirFileNamesAll-r` is used to extract recursively (suffix `-r`) the contents of a directory. Script fragment 15.2 lets the user select a root directory (`selectDirectory`) and stores its content in file names.txt.

```
set path = D:/;
selectDirectory path = $path$;
dirFileNamesAll-r $fileDirectory$/names.txt = $path$;
```

Script 15.2 - Selecting a root directory and recording its contents in file names.txt

Then, the user is invited to assign a nickname to the root directory, either by entering a new name (the path of the directory just selected is suggested) in which case both a new root directory and its first instance are created, or by selecting one among those already recorded, in which case a new instance is created for this root directory (Script 15.3).

```
set rnn = $path$;
ask-u rnn = [/bSelect/Enter the nickname of the root directory]
             Root directory nickname:[select DirNick
                                      from   DIRECTORY];
```

Script 15.3 - Getting the nickname of the selected root directory

15.5.1 Creating a directory

If this root directory already exists, its Id (dirID) is extracted, otherwise, a new DIRECTORY row is inserted (Script 15.4).

```
extract dirID = select DirID from DIRECTORY
                where  DirNick = '§rnn§';
if ('$SQLdiag$' = 'NONE');
   insert into DIRECTORY(DirName,DirNick)
          values ('§path§','§rnn§');
   extract dirID = select DirID from DIRECTORY
                   where  DirNick = '§rnn§';
endif;
```

Script 15.4 - Creating a root directory if needed

15.5.2 Creating the current instance

Now, we have enough information to record the information on the current instance in a row of table INSTANCE (Script 15.5). Variables dat and tim have preliminarily been set to the current date and time.

```
insert into INSTANCE(DirID,InstDate,InstTime)
             values ($dirID$,'$dat$','$tim$');
extract instID = select InstID from INSTANCE
                 where  DirID = $dirID$
                 and    InstDate = '$dat$'
                 and    InstTime = '$tim$';
```

Script 15.5 - Creating an instance of the root directory

15.5.3 Creating the nodes of the current instance

The next step is to store the nodes of this instance (Script 15.6). Their names are extracted from file names.txt. Each line is decomposed by function splitPath of library LFile. It returns five values: the *full name*, the *prefix*, the *local name* for a directory (empty for a file), the *local name* for a file (empty for a directory) and the *extension* (empty for a directory).

For files, additional informations on *size*, *last update time* and *last access times* are collected through function infoFile of library LFile.

```
for line = [file $namesFile$];
  function ful,pre,dir,fil,ext = LFile:splitPath {$line$};
  if ('§fil§' = '');
    insert into NODE(InstID,NType,FullName,Prefix,LocalName)
             values($instID$,'D','§ful§','§pre§','§dir§');
  else;
    function siz,cre,upd,acc = LFile:infoFile {$line$};
    insert into NODE(InstID,NType,FullName,Prefix,LocalName,
                     Extension,Size,LastUpdate,LastAccess)
             values($instID$,'F','§ful§','§pre§','§fil§',
                    '§ext§',$siz$,'$upd$','$acc$');
  endif;
endfor;
```

Script 15.6 - Creating the nodes of the current instance of the root directory

Filling column ParentID is straightforward: the full name of the parent of a node is the prefix of this node in the same instance. Hence the `update` query of Script 15.7.

The level of a node could be computed by a recursive query, but it is more simply derived from the number of symbol `'/'` in its full name, relative to that of its root directory (Script 15.7, last part).

```
update NODE
set    ParentID = (select NodeID from NODE
                   where  FullName  = NODE.Prefix
                   and    InstID  = $instID$)
where  InstID = $instID$;

compute NL = countInst('§path§','/');
update NODE set Level = countInst(FullName,'/') - $NL$
where  InstID = $instID$;
```

Script 15.7 - Computing the values of columns ParentID and Level

15.5.4 Computing the path of a node

The value of column Path of a node is the list of node Id's from the root directory (included) to this node (included). Computing paths in graphs in general, and in trees in particular, is a nice application of recursive queries also known as *recursive CTE* in SQL (see Chapter 36 for example).

In Script 15.8, CTE TREE *extracts* couples (ParentID, NodeID) from NODE and recursively computes the path of a node (starting from the root directory) from that of its parent, to which it appends its own node Id.

```
with TREE(FromID,ToID,Path) as (
   select ParentID,NodeID,cast(NodeID as char)
   from   NODE where ParentID is null
      union all
   select T.ToID,N.NodeID,T.Path||' '||cast(N.NodeID as char)
   from   TREE T, NODE N
   where  T.ToID = N.ParentID
   )
update NODE
set Path = (select Path from TREE where ToID = NODE.NodeID);
```

Script 15.8 - Computing node paths - Recursive version

The way node id's are formatted in the path is a bit raw. For example, the sequence <937,1038,11847,32> will be expressed as

```
'937 1038 11847 32'
```

which is not particularly elegant and, more important, will not serve the depth-first ordering role we intend to assign to it (see below). So, we suggest to replace expression

```
cast(NodeID as char)
```

by[2]

```
NType||frame(NodeID,$nd$,'>','0')
```

which obviously deserves some explanation!

Function `frame(s,nd,a,p)` returns string `s` framed in a `nd`-long string; `a` is the alignment ('<' = left, '>' = right, '^' = center) of `s` in the frame; unused positions are padded with characters `p`. The value of `nd` is computed as the number of digits of the largest node Id:

```
extract nd = select length(cast(max(NodeID) as char))
             from   NODE;
```

In addition, each node Id is prefixed with the nature of the node (D or F). The result becomes quite regular:

```
'D00937 D01038 D11847 F00032'
```

This algorithm can also be translated into an iterative script. Its logic is quite similar to that of the recursive version:

- first, the path of the root node is set to its own node Id,
- then, each iteration computes the path of the children of each node of the preceding level, until all levels have been processed (Script 15.9).

```
extract maxLevel = select max(Level) from NODE;
update NODE set Path = cast(NodeID as char)
where  Level = 0;
for iLev = [1,$maxLevel$];
   update NODE
   set Path = (select Path from NODE
               where  NodeID = NODE.ParentID)
                              ||' '||cast(NodeID as char)
   where Level = $iLev$;
endfor;
```

Script 15.9 - Computing node paths - Iterative version

Note that, for simplicity, both algorithms have been designed for a table NODE that stores one directory instance only. The fully functional scripts must restrict the processing to the current instance, that is, to those NODE rows that satisfy condition

```
where InstID = $instID$
```

2. Function `frame` transforms numeric values into character strings, so that casting them into `char` type is unnecessary.

15.6 Applications

The value of a database lies in the quality of the data but also on the problem solving application portfolio. We will develop some simple applications that focus on the analysis of a collection of directories in order to understand its structure and its contents on one hand, and to search them for potential duplicates on the other hand, a frequent problem of people who use computers for several years!

The applications described below just are suggestions that are intended to be adapted and completed to meet specific readers needs.

15.7 Application 1: general statistics

Some global statistics will give us a general picture on our directories (Script 15.10). Script 15.11 provides more detailed statistics on the instances of each root directory, notably providing the time range of these instances.

```
select
  (select count(*) from DIRECTORY) as "Root directories",
  (select count(*) from INSTANCE)  as Instances,
  (select count(*) from NODE)      as Nodes,
  (select count(*) from NODE where NType = 'D') as Directories
  (select count(*) from NODE where NType = 'F') as Files;
```

Script 15.10 - Displaying the size of the tables

```
select D.DirNick                    as "Root directory",
       count(*)                     as Instances,
       min(InstDate||' '||InstTime) as "First recording",
       max(InstDate||' '||InstTime) as "Last recording"
from   DIRECTORY D, INSTANCE I
where  D.DirID = I.DirID
group by D.DirID;
```

Script 15.11 - Statistics for each root directory

Excerpts of the result are shown in Figure 15.4, top two tables. This screen shot also shows the hierarchical instance description produced by Script 15.12. For each directory and for each of its instances, it provides the instance time, the number of directories, the number of files, the total file size and the most recent file update time.

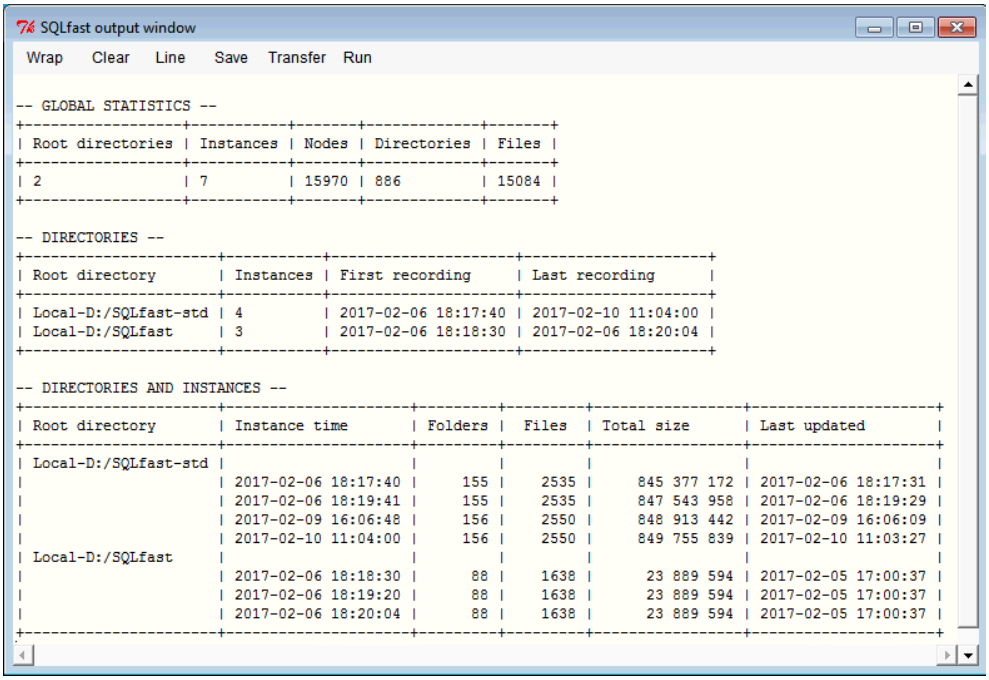


Figure 15.4 - General statistics

This script is a bit more complex than the queries developed so far. It comprises three main parts:

1. the **`from`** clause builds the union of two results sets, one producing the identification of root directories and the second one the description of instances. Since they must comprise the same number of columns, those of them that are not significant are left empty (string `''`).
2. the **`order by`** clause creates the hierarchical sequence that presents each directory immediately followed by its instances. To do so, we define special computed column **`sort`**. Its values are that of DirID in both tables, so that all DIRECTORY and INSTANCE rows are grouped, followed by a character that forces the DIRECTORY row to precede its INSTANCE rows: we naturally have chosen `'D'` for DIRECTORY and `'I'` for INSTANCE.
3. the **`select`** clause formats the values from the combined result sets and assigns them expressive names. Function `thousandSep(n,sep)` returns the character string formed by number `n`, in which character `sep` separates groups of three digits in the integral part of `n`. E.g., `thousandSep(1234567.45,',')` returns `'1,234,567.45'`.

```
select DirNick                    as "Root directory",
       Time                       as "Instance time",
       frame(Folders,7,'>',' ') as Directories,
       frame(Files,7,'>',' ')   as " Files",
       frame(thousandSep(Size,' '),15,'>',' ') as "Total size",
       Last                       as "Last updated"
from
 (select DirNick,'' as Time,
         '' as Folders, '' as Files,'' as Size,
         '' as Last, cast(DirID as char)||'D' as Sort
  from DIRECTORY D
    union
  select '' as DirNick,InstDate||' '||InstTime as Time,
         (select count(*) from NODE N where NType = 'D'
          and  N.InstID = I.InstID) as Folders,
         (select count(*) from NODE N
          where  NType = 'F' and N.InstID = I.InstID) as Files,
         (select case NType when 'F' then sum(Size) else ''
                 end from NODE N
          where  N.InstID = I.InstID) as Size,
         (select max(LastUpdate) from NODE N
          where  NType = 'F' and N.InstID = I.InstID) as Last,
          cast(DirID as char)||'I' as Sort
  from INSTANCE I)
order by Sort;
```

Script 15.12 - Displaying the directory/instance hierarchy

15.8 Selecting a directory instance

All the following applications will carry out an analysis of a definite directory instance (or two instances for some of them). In order to alleviate the code of these applications, we could create specific scripts to select this instance and to get the nickname of the directory. We could also use a lighter technique: storing common patterns as code fragments in SQLfast variable. These patterns are shown in Script 15.13. Selecting an instance is now easy:

```
ask inst = Instance:[!$listInstances$];
```

and extracting the directory name of instance `$inst$` is quite straightforward as well:

```
extract iName = $instName$;
```

```
set listInstances
    = select D.DirNick||' ('||I.InstDate
             ||' '||I.InstTime||')' as Name,
             I.InstID
      from   DIRECTORY D,INSTANCE I
      where  D.DirID = I.DirID
      order by Name;

set instName
    = select DirNick||' ('||InstDate||' '||InstTime||')'
      from   DIRECTORY D, INSTANCE I
      where  InstID = $inst$ and D.DirID = I.DirID;
```

Script 15.13 - Two useful common patterns to select instances

15.9 Application 2: Structure of a directory instance

We call *structure of an instance* the list of internal directories that shows, through indentation, the inclusion relationships. It represents graphically the tree structure of these directories (Figure 15.5).

Script 48.14 that produces this structure is straightforward. The generation of the sequences of dots that create the indentation is based on column Level of table NODE (function `repeat(s,n)` returns a character string formed by **n** time string **s**). To make directory names distinguishable from those of files[3], the former are surrounded by square brackets.

```
ask inst = Instance:[!$listInstances$];
extract iName = $instName$;
write-ab -- Directory structure of directory "$iName$" --;
select NodeID,
       repeat('.... ',Level)
     ||case when NType = 'F'
            then LocalName
            else '['||LocalName||']'
       end as Name
from  NODE
where InstID = $inst$ and NType = 'D'
order by Path;
```

Script 15.14 - Displaying the directory structure of an instance

3. Not particularly striking in this case, but will be useful in the next report!

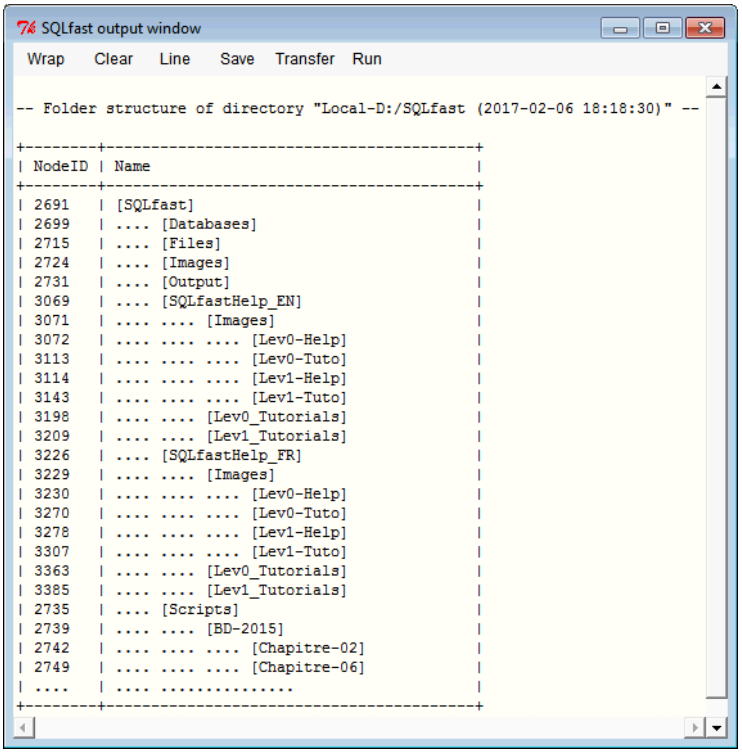


Figure 15.5 - Structure of directory SQLfast-std (excerpts)

The correct tree structure of the report is defined by the `order by` clause, that implements, through the values of column Path, the *depth-first* traversal order of the node tree, according to which the children of a node N are *visited* before visiting the next sibling of N and its children.

15.10 Application 3: Examining the contents of an instance

This application is an extension of the previous one. The file and directory local names are presented with the same indentation technique while an additional column indicates the prefix of each node.

The contents of the directory can be limited to nodes of a certain type: directories only, files only, both (as in the previous report) or files with a definite extension. Figure 15.6 shows a report on *.sql scripts of the instance selected (Figure 15.7).

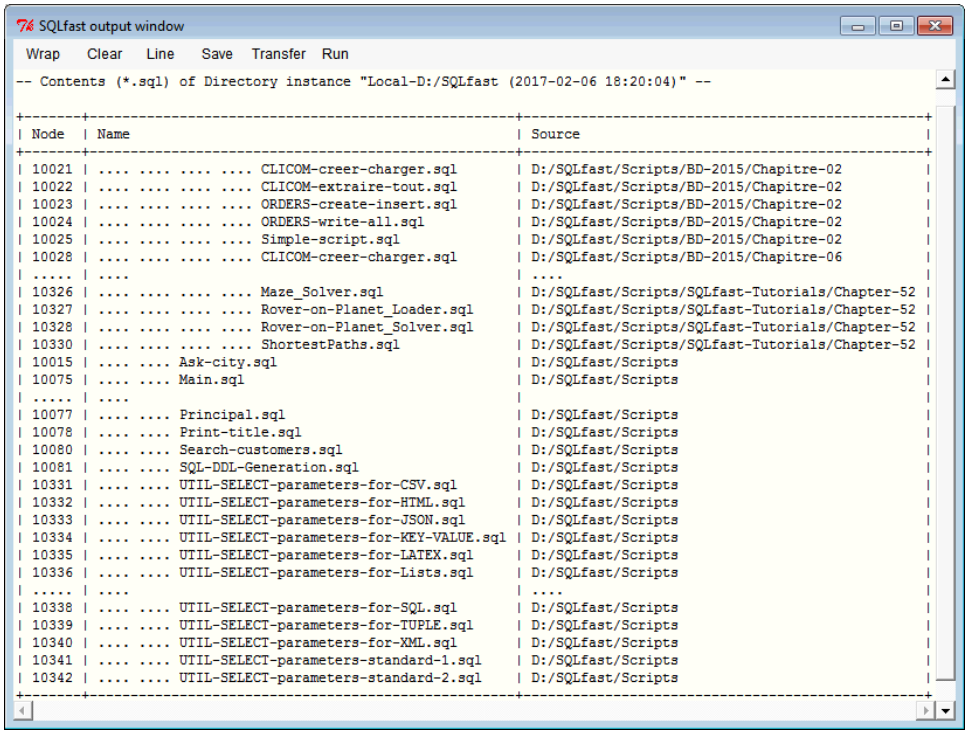


Figure 15.6 - List of all the sql scripts of the instance selected

This report has been produced by Script 15.16, that comprises four main parts:

1. the dialogue box (Figure 15.7)
2. the sequence of `if ('$ext$'...)` statements that translate the node type into an SQL condition stored in variable extCond
3. extracting the name of the instance and writing the report title
4. the SQL query that produces the report itself.

To make the script simpler, we define pattern `listTypes`, that creates the list of the types of the file nodes in table NODE (Script 15.15).

15.11 Application 4: Duplicate nodes in a directory instance

Finding nodes that denote identical files in a root directory is one of the most important problem when managing a large collection of files. We first address the search of potentially duplicate files and directories in a root directory.

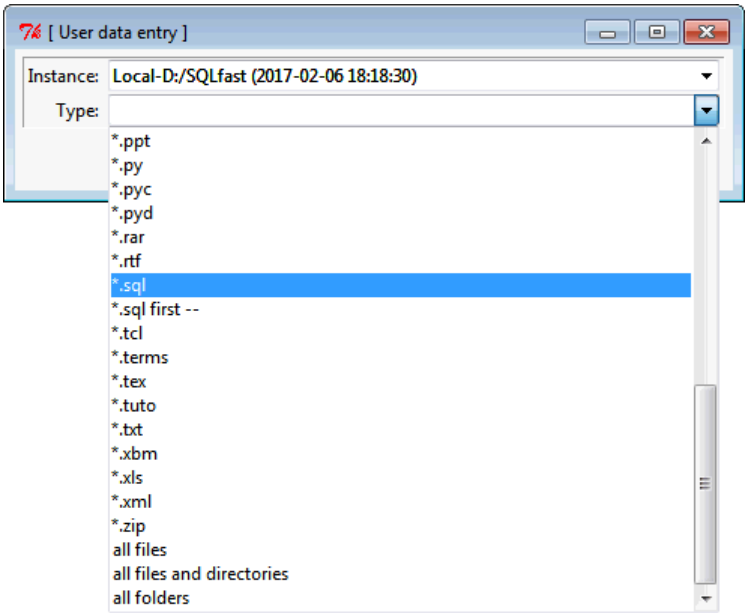


Figure 15.7 - Selecting the type of contents to display

```
set listTypes
    = select 'all files,all folders,...,'
             ||group_concat('*.'||Extension)
      from   (select distinct lower(Extension) as Extension
              from   NODE
              where  Extension <> ''
              order by lower(Extension));
```

Script 15.15 - Pattern that extracts the types of all the files recorded in table NODE

15.11.1 List of duplicate nodes in a directory instance

Through this small application, we want to know which nodes have the same name in a selected root directory. This indicator is not sufficient to declare them identical, but it is good starting point to evaluate redundancies in a directory.

Script 15.17 asks the user to select a directory and to specify the minimum multiplicity for which duplicate names must be reported.

Figure 15.8 shows excerpts of the result of its execution for directory Local-D:/SQLfast for a multiplicity of 2. This report is just a short summary of the duplication phenomenon, but it is worth being examined to decide whether a more in-depth analysis may be useful. The latter is the objective of the next section

```
ask inst,ext = Instance:[!$listInstances$]
               |Type:[!$listTypes$];
if ('$ext$' = '') set ext = - all files and directories;
set extCond = ;
if ('$ext$' = 'all files and folders')
   set extCond = ;
if ('$ext$' = 'all files')
   set extCond = and NType = 'F';
if ('$ext$' = 'all directories')
   set extCond = and NType = 'D';
if (startswith('$ext$','*.'))
   set extCond = and Extension = substr('§ext§',3,99);
extract iName = $instName$;
write-ab -- Contents ($ext$) of Directory instance "$iName$";
select NodeID as Node,
       repeat('.... ',Level)
     ||case when NType = 'F'
            then LocalName
            else '['||LocalName||']'
       end as Name,
       Prefix as Source
from  NODE
where InstID = $inst$ $extCond$
order by Path;
```

Script 15.16 - Displaying the contents of a directory

```
set inst = ;
set mul  = 2;
ask-u inst,mul = Instance:[!$instName$] | Min multiplicity:;
if ('$DIALOGbutton$' = 'Cancel' or '$inst$' = '') goto EXIT;
extract iName = $instName$;
write-ab Duplicates in Directory instance "$iName$";
select case when NType = 'F'
            then LocalName
            else '['||LocalName||']'
       end as Nodes,
       count(*) as Nbr
from   NODE where InstID = $inst$
group by Nodes
having Nbr >= $mul$
order by Nbr desc, Nodes;
```

Script 15.17 - Identifying duplicate files and directories

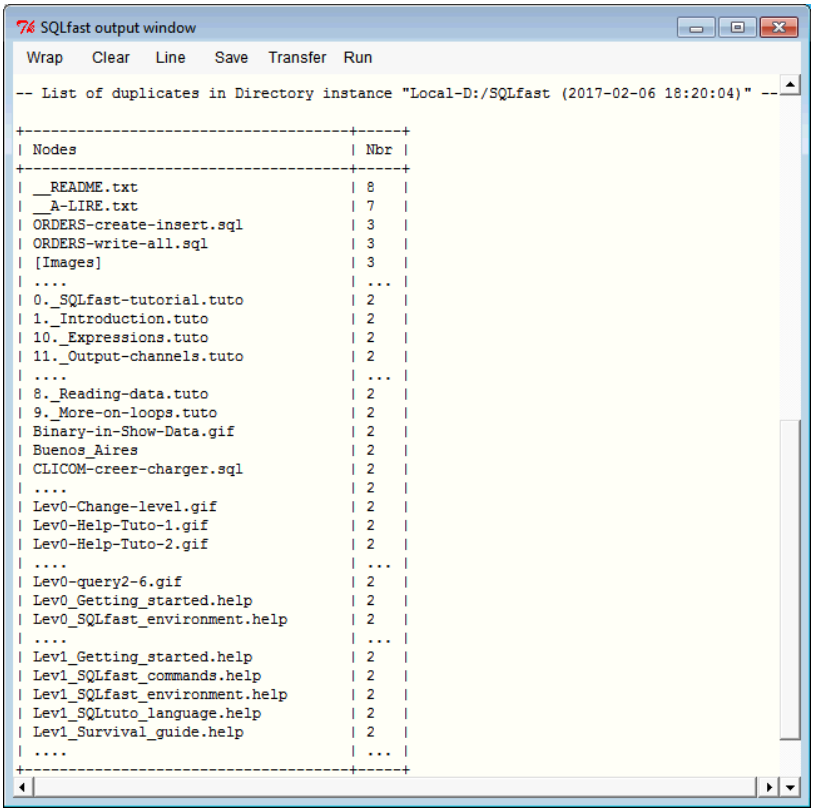


Figure 15.8 - Duplicate nodes in a root directory instance (excerpts)

15.11.2 Description of duplicate nodes in a directory instance

This application extends the previous report by organizing duplicate nodes in a tree structure and by adding size and update time information to better compare potentially duplicate files (Figure 15.9).

We give the report a hierarchical structure in which, for each local name that appears more than once, we list the full names of the directories in which they are located, together with additional properties size and last update time.

The SQLfast code is given in Script 15.18. As usual, the hierarchy is created by the union of the rows providing the two kinds of lines in the result set: the common local names and the parent directories of these names.

The correct depth-first order is defined by the sort key **Sort** built by the common local name suffixed with letters `'A'` and `'B'` respectively.

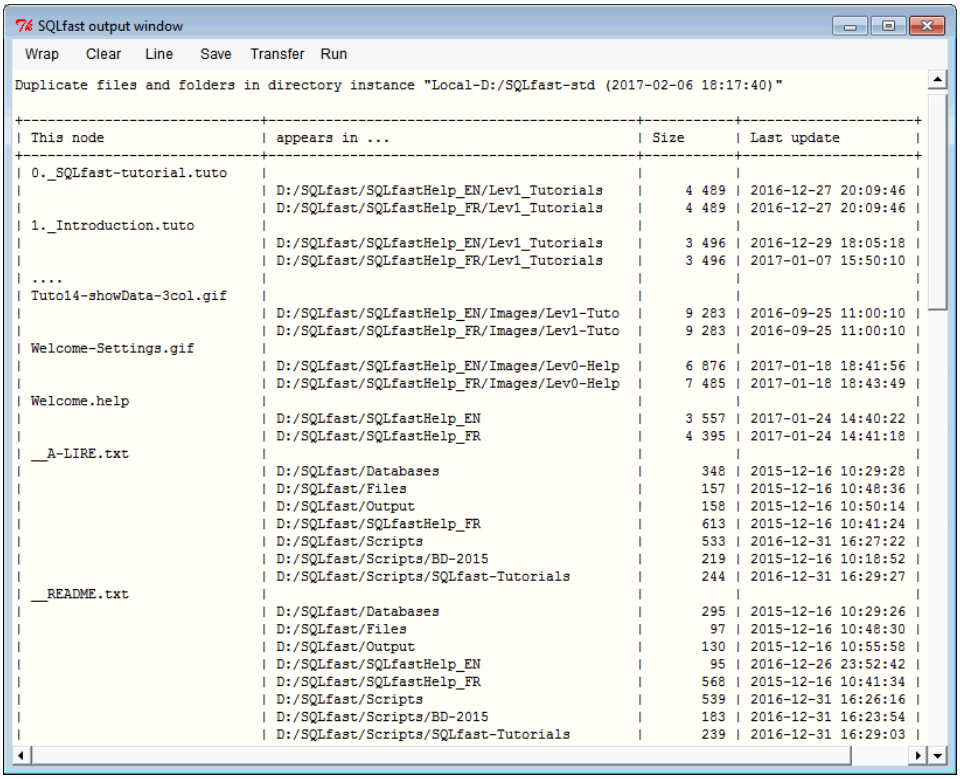


Figure 15.9 - Detailed report of potentially duplicate files

15.12 Application 5: Comparing two directory instances

This application compares the contents of two directory instances by identifying the components of the first one that are absent from the second one, and conversely.

This comparison particularly makes sense when the instances have similar structures, for example when we examine the evolution of two instances of the same directory or when we compare two backup external disks that are supposed to be identical.

Nodes are compared on their name and their type, but a more strict comparison can be chosen, that adds the *size* and *last update time* properties to the operands. In addition, the comparison can be limited to two subdirectories of the selected instances. This is shown in the parameter selection box of Figure 15.10.

```
set inst = ;
set mul  = 2;
ask-u inst,mul = Instance:[!$instName$] | Min multiplicity:;
if ('$DIALOGbutton$' = 'Cancel' or '$inst$' = '') goto EXIT;
select
       "Node name" as "This node",
       Source as "appears in ...",
       Size,
       LastUpdate as "Last update"
from
   (select distinct
           case when NType = 'F'
                then LocalName
                else '['||LocalName||']'
           end as "Node name",
           '' as Source,
           '' as Size,
           '' as LastUpdate,
           LocalName||'A' as Sort
    from   NODE
    where  InstID = $inst$
    and    LocalName in (select LocalName from NODE
                         where  InstID = $inst$
                         group by LocalName
                         having count(*) >= $mul$)
        union
    select
           '' as "Node name",
           Prefix as Source,
           frame(thousandSep(Size,' '),13,'>',' ') as "Size",
           LastUpdate,
           LocalName||'B' as Sort
    from   NODE
    where  InstID = $inst$
    and    LocalName in (select LocalName from NODE
                         where  InstID = $inst$
                         group by LocalName
                         having count(*) >= $mul$)
   )
order by Sort;
```

Script 15.18 - Generating the hierarchical report of Figure 15.9

To clarify the algorithms, we decide to work on two temporary tables, NODE1 and NODE2, in which we store the sets of nodes to compare in both instances. The comparison can then be translated into the set theoretic difference of these tables (SQL operator **except**).

The fragment of Script 15.19 displays the local name nodes of NODE1 that are absent from NODE2, based on their values of FullName and NType.

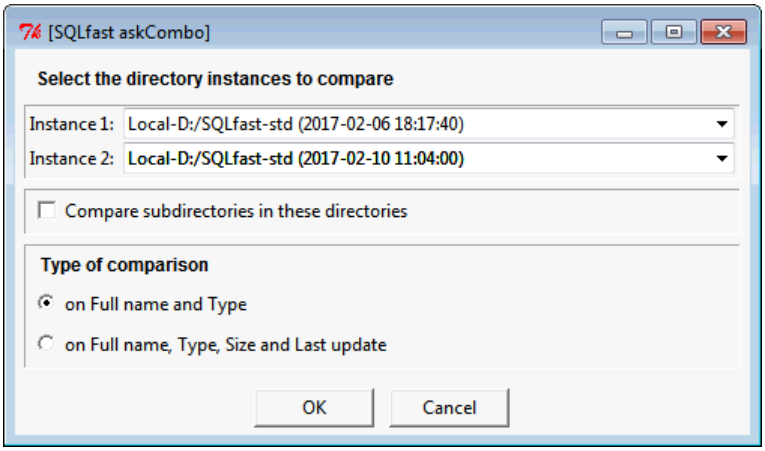


Figure 15.10 - Comparing two directory instances

Expression `case-end` performs a pretty formatting of the node names according to their type. Variables **dir1** and **dir2** contains the full names of the directories to compare.

```
select NType as "Type",
       case when NType = 'F'
            then ' '||Name
            else '['||Name||']'
       end as "In Directory (1) but not in Directory (2)"
from  (select replace(FullName,'§dir1§/','') as Name,NType
       from   NODE1
          except
       select replace(FullName,'§dir2§/','') as Name,NType
       from   NODE2);
```

Script 15.19 - Displaying the nodes in NODE1 but not in NODE2

15.13 Application 6: Searching for clone files

So far, the concept of duplicate has been based on external properties of two files: same local name, same size and same last update time. This approach is far from reliable in practice since it may lead to two kinds of errors:

– *false positive*: two files happen to have the same external properties, though their contents are quite different

– *false negative*: two files have different names and last update time while their contents are identical.

15.13.1 The problem

To go beyond the mere comparison of external properties, we have to carry out a more in-depth examination of the files to identify those of them that are real *clones*, that is, that have the same contents. Let us adopt the following reasoning:

1. clones may have different local names (ORDERS.db and ORDERS-old.db) and different extension (ORDERS.db, ORDERS.back); so, comparing names and extensions may help detecting some clones but surely not in all cases
2. clones should have the same last update time but copying or downloading a file may change the update time of the copy
3. clones have the same size
4. clones have the same contents, compared byte by byte.

If we agree on these rules, then only the last two ones should be used to detect clones. If checking whether two files have the *same size* is fairly easy, the *same contents* rule is likely to be harder to implement, specially for large files (small file can be preloaded in RAM).

Considering **S** = {f1, f2, ..., fn-1, fn} a set of **n** candidate files. A procedure that compares two files at a time by reading them in parallel, has a best case cost of **2(n-1)** sequential readings if the files are real clones. Indeed, we have to compare the following pairs: (f1, f2), (f2, f3), ..., (fn-2, fn-1), (fn-1, fn); f1 and fn are read once, all the others twice. On the contrary, if the files of **S** are not clones, each pair of files must be evaluated, that is, **n(n-1)** file comparisons. Each of them requires two sequential readings, for a total cost of **2n(n-1)** sequential readings. However, these readings will be partial, since they will stop once a pair of different bytes have been read. If, on the average, these bytes are encountered in the middle of the files, then the estimated total cost will be **n(n-1)** full sequential readings.

If a new file is added and is reported in a further instance, we will have to carry out the comparison process again. To save expensive file processing, we could think of a way to record the comparison result. This would require a new table CLONE_SET in which each row denotes a set of clones already evaluated and dependent table CLONE_MEMBER each row of which references a NODE row member of this set. When a set of clones has been identified, it is recorded in these tables. This way, introducing a new file will entail less effort, since it must be compared with one member of each clone set.

To be honest, all this seems both very expensive and complicated.

15.13.2 Saved by (computer) science

Fortunately, computer science (and SQLfast as well!) provides us with a nice technique to compare the contents of files of any size: *hash functions*, and more particularly, *secure hash functions*.

This technique has been described in Section 15.7.3 (Secure hashing). It consists in deriving from a byte string **B**, here the contents of a file, another, short, fixed

length, character string (the *hash value*) that can be used as a *fingerprint* or *signature* of **B**. With good hash functions, the probability of two different byte strings producing the same signature is extremely low. Therefore, we are allowed to consider that the hash value of a file (reasonably) identifies it among all the files, past, present and future, we intend to manage. SQLfast **hashFile** function and statement return, for any file, whatever its type and its length, a *hash value* consisting of a string of 256 bits converted into a string of 64 hexadecimal characters.

Now, the problem of searching for clones can be solved quite simply. We compute the hash value of each candidate file and we store it along with the other properties of these files. Clones are files with the same *hash value*.

15.13.3 Hashing files

First, we add a new column to table NODE[4]:

```
alter table NODE add column Hash char(64);
```

and we fill it for all the rows in instance **inst**:

```
update NODE set Hash = hashFile(FullName)
where  InstID = $inst$ and NType = 'F';
```

Function **hashFile** returns the hash value of the file denoted by its argument. However, it returns *null*[5] if this argument does not refer to an existing file (e.g., a directory, a link or a missing file).

An alternative iterative version, based on **hashFile** statement, is proposed in Script 15.21.

Performance

The hash functions and statement of SQLfast are based on SHA256, one of the best algorithms to generate near collision[6] free hash value distribution. What about its performance? Each call reads the sequence of bytes of a file and applies to them complex binary computation. On a standard, average, machine (laptop, i7, 7200 rpm HD), a directory instance of 2,700 files totalling 850 MB is processed in 8.5 s. So, the reading-computation speed is of 100 MB/s. Not bad, considering that this operation has to be performed only once per instance. Not surprisingly, the iterative version is a slower: 21.5 s. instead of 8.5 s.

4. Actually, it has already been defined in table NODE.
5. Constant `None` in Python, interpreted as `null` in SQLite. SQLfast statement **hashFile** returns an empty string instead of `None`.
6. *collision*: undesirable event of two different source files generating the same hash value, which could lead to false positive.

```
update NODE set Hash = hashFile(FullName)
where  InstID = $inst$ and NType = 'F';
```

Script 15.20 - Updating column Hash of table NODE: pure SQL version

```
for node,name = [select NodeID,FullName from NODE
                 where  InstID = $inst$ and NType = 'F'];
   hashFile hash =  $name$;
   update NODE set Hash = case when '$hash$' = ''
                               then null
                               else '$hash$'
                          end
   where  NodeID = '$node$';
endfor;
```

Script 15.21 - Updating column Hash of table NODE: iterative version

15.13.4 Identifying clone candidates in a directory instance

Basically, exploring directory instances to find clones is quite similar to the procedures based on local names, as we did so far, where we substitutes Hash for LocalName.

Let us first define **cloneHash**, an SQL pattern that denotes the set of Hash values that exist more than once in instance **inst** (Script 15.23, top). Condition `Hash is not null` copes with missing files and condition `Size > 0` discards empty files.

Extracting the list of clone files is now straightforward (Script 15.22).

```
select Hash,LocalName,Prefix
from   NODE where Hash in ($cloneHash$)
order by Hash;
```

Script 15.22 - List of clone files in instance **inst** (raw version)

Since the values of Hash are a bit long, we display excerpts of them:

```
substr(Hash,1,8)||'...'||substr(Hash,56,8) as "Hash"
```

We define pattern **hashDigest** with this expression to make queries more readable (Script 15.23, bottom).

```
set cloneHash = select Hash from NODE
                where  InstID = $inst$ and NType = 'F'
                and    Hash is not null and Size > 0
                group by Hash having count(*) > 1;

set hashDigest = substr(Hash,1,8)||'...'||substr(Hash,56,8);
```

Script 15.23 - Two patterns: set of the Hash values of clones in instance **inst** and condensed version of hash values

Following the sorting trick of Script 15.12, Script 15.24 produces the nice hierarchical report of Figure 15.11. The group name (alias Group) is meaningless, its only role being to create a two-level hierarchy.

The structure of this query is fairly simple. The first subquery of the `from` clause creates the heading of the clone groups. The second subquery creates one row per clone file.

```
select Hash as Group,LocalName as Clone,Prefix as Source
from (select $hashDigest$ as Hash,
             '' as LocalName,'' as Prefix,Hash||'C' as Sort
      from   ($cloneHash$)
         union
      select '' as Hash,
             LocalName,Prefix,Hash||'F' as Sort
      from   NODE
      where  Hash in ($cloneHash$) and InstID = $inst$
     )
order by Sort;
```

Script 15.24 - List of clone files in instance **inst** (hierarchical view)

15.13.5 Identifying clone candidates among two directory instances

The problem seems quite similar to that of finding clones in a single instance. One important distinction though: these instances should be of *different directories*. Indeed, two instances of the same directory will most of the time exhibit little differences, so that most files will be clones!

The approach we suggest consists in finding clones in the **union** of two source instances. This will work nicely provided (1) each instance has been cleaned from its own clones, or at least if these clones have been reduced as far as possible and (2) the instances have not too many common files. Otherwise a too large number of files will be declared clones, making the report of little interest.

To make the main query clearer, we first define pattern **`cloneHash2`** that computes the set of hash values common to both instances, denoted by variables inst1 and inst2 (Script 15.25). These values are those of candidate clones.

The main query is shown in Script 15.26. It creates a hierarchical report in which each set of clones coming from both instance are presented as a group. The main difference with the report of Figure 15.11 is that the source directory of each clone (its Prefix) is indexed with symbol <1> or <2> according to this source.

The query sorts three kinds of rows so that they appear in this order for each group of clones:

1. the group id (the concise Hash value (sort key = `Hash||'C0'`)
2. the clones from instance <1> (sort key = `Hash||'F1'`)
3. the clones from instance <2> (sort key = `Hash||'F2'`)

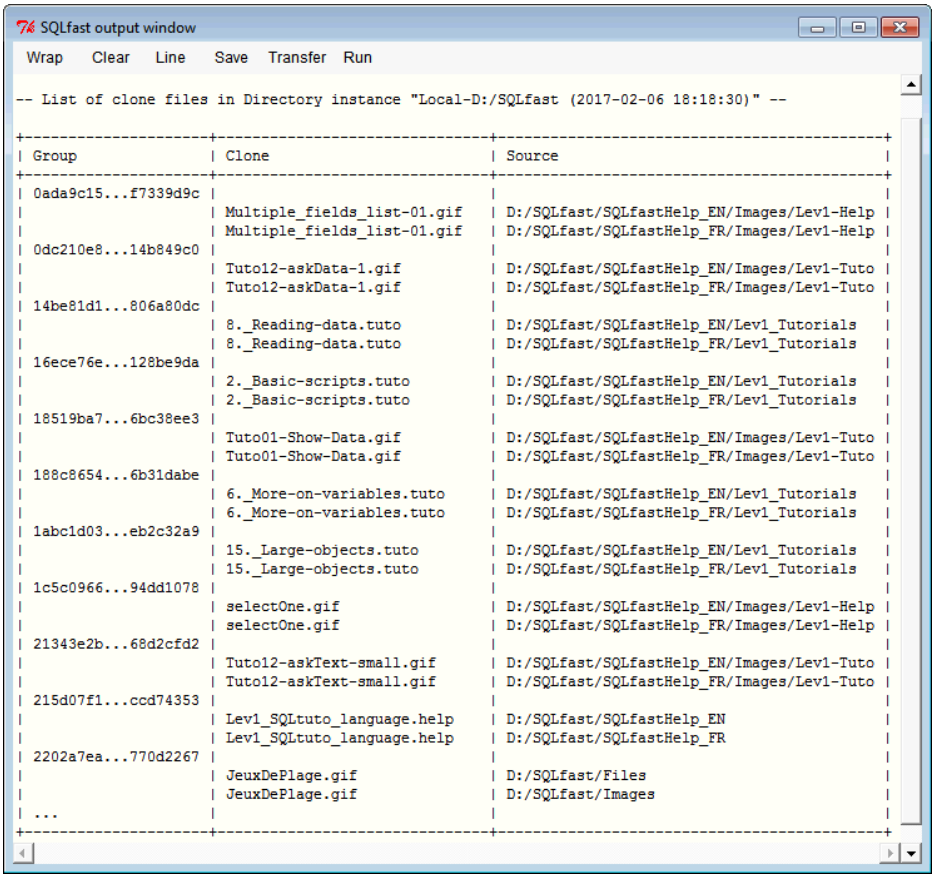


Figure 15.11 - Hierarchical view of the clone files of a directory instance

```
set cloneHash2 = select Hash
                 from (select Hash from NODE
                       where  InstID = $inst1$ and NType = 'F'
                       and    Hash is not null and Size > 0
                          intersect
                       select Hash from NODE
                       where  InstID = $inst2$ and NType = 'F'
                       and    Hash is not null and Size > 0);
```

Script 15.25 - Pattern computing the set of the hash values common to instances inst1 and inst2

```
select Hash as "Group",LocalName as Clone,Prefix as Source
from (select $hashDigest$ as "Hash",
             '' as LocalName,'' as Prefix,
             Hash||'C0' as Sort
      from   NODE
      where  Hash in ($cloneHash2$)
         union
      select '' as "Hash",
             LocalName,'<1>'||Prefix as Prefix,
             Hash||'F1' as Sort
      from   NODE
      where  Hash in ($cloneHash2$) and InstID = $inst1$
         union
      select '' as "Hash",
             LocalName,'<2>'||Prefix as Prefix,
             Hash||'F2' as Sort
      from   NODE
      where  Hash in ($cloneHash2$) and InstID = $inst2$
     )
order by Sort;
```

Script 15.26 - Identifying clones between two directory instances

15.14 A touch of optimization

No indexes have been created yet, apart from those automatically associated with primary keys (and unique keys, if any) by most RDBMSs.

Creating non unique indexes may accelerate some queries. However, deciding which ones will really be useful is not an easy task. Indeed, the benefit of an index depends on the size of the table, its selectivity (which measures the proportion of the table that can be accessed through it), the frequency of the queries that will use it

and the cost of its updating. In addition, a missing index can be compensated by the access strategy of the DBMS, which can decide to dynamically create an index on the fly for the execution of a query.

Due to their small size, tables DIRECTORY and INSTANCE can be left without additional indexes.

Concerning table NODE, additional indexes will generally be useless for a population of a few thousands rows. Moreover, reducing execution time from 1 second to 100 milliseconds is of little interest when we want to produce, once in a while, a report that will require one minute to analyze.

A common sense rule suggests to index foreign keys, as these indexes will support such operations as joins with the parent table and delete/update in the parent table. This is quite justified in this case since most queries operate on the nodes of a definite instance. So, we create an index on InstID (Script 15.27). As to ParentID, we observe that only Script 15.8, that computes Path, includes a condition `is null` on this column. We suggest to discard this foreign key index.

An index on NType, though this column is mentioned in many conditions (Scripts 15.10, 15.12, 15.14, 15.16, 15.20, 15.21, 15.23 and 15.25), will be useless due to its low selectivity (two values only). Same argument and conclusion for an index on Level.

Several queries include `where`, `order by` or `group by` clauses on columns Hash, Path, FullName, LocalName and Extension, all in combination with InstID. Indexes on these columns may decrease the execution time of report generation queries but not by far.[7] Script 15.7, that computes ParentID values could make use on an index on (InstID, FullName). However, this operation already is very fast without such index.

```
create index ndxNodeInstID on NODE(InstID);
```

Script 15.27 - We just need one additional index

A last argument to be conservative about index creation is that each index on NODE entails a loading time penalty. So, the lesser, the best for this operation.

Of course, reporting on *very large* directory instances may change our conclusion.

7. On the example shown in this chapter, most reporting queries require less than 1 second with an index on InstID only.

15.15 Building the DIRECTORY application

The functions developed above have been integrated into a global application. The main program, called **DIR-MAIN.sql**, in directory **Scripts/Case_Studies/Cases_Directory_Manegement**. Its structure is that of a simple loop that displays the control panel of Figure 15.12 and executes the function selected.

Beside the analysis functions (right side column), some basic management functions have been integrated (left side column):

- creating a new DIRECTORY.db database
- loading a new instance of an existing or a new root directory
- inserting the hash values of an instance
- deleting a root directory and all its instances
- deleting an instance of a root directory and all its nodes
- opening the help window

The latter function display a small tutorial document (DIRECTORY.tuto) that describes the main functions of the application.

The mapping between the button checked and the script to execute is implemented through item list **modList**, which is made up of the names of the scripts separated by semi-colons. The number of the button checked by the user is returned in variable **op**. This value is used to get the module name through expression

```
itemS('$modList$',$op$)
```

that returns the **op**th item in list **modList**.[8]

The main script is shown in Script 15.28. Some comments on its main parts:

- [1]: sets the current script directory **scriptPath**
- [2]: loads the common patterns, collected in script DIRECTORY-Common-Patterns.sql
- [3]: builds the module name list in **modList**
- [4]: sets the default operation
- [5]: main loop
- [6]: creates and displays the control panel; returns operation number **op**
- [7]: extracts the selected module name in variable **module**
- [8]: executes the selected module.

The main script also comprises ancillary operations that are not shown here.

8. Suffix **S** refers to the standard separator, which is the semi-colon.

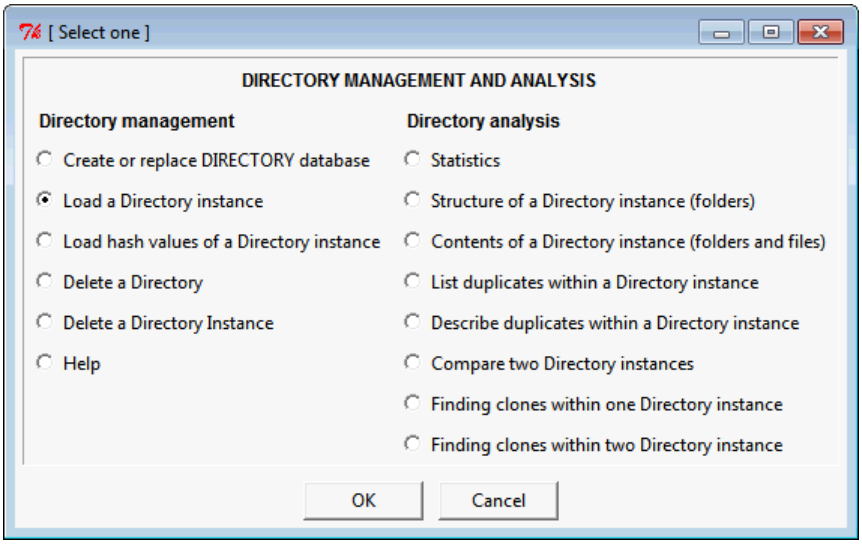


Figure 15.12 - Control panel of the DIRECTORY application

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

15.16 Suggestions

This chapter has shown that simple tree structures can provide much interesting information on the contents of storage media. On the analysis side, a lot of other reporting queries can be developed, such as the following:

- comparing two media, to check whether they have the same contents, despite different structures
- comparing the structure of two instances
- identifying the files that have been renamed (comparing two successive instances)
- computing the size of each directory
- synthesizing the activity of some files across successive instances (based on varying update and access time); identifying dead (or archived) files.

On the other hand, directory structure modification can be developed, such as copying, deleting and renaming files.

```
set scriptPath = SQLfast-Tutorials/Chapter-48;                [1]
execSQL $scriptPath$/DIRECTORY-Common-Patterns.sql;           [2]
set modList = Create-or-Replace;Load-Instance;                [3]
set modList = $modList$;Load-Hash;Delete-Directory;
...
set modList = $modList$;Select-Clone-Candidates-2;            [3]
set op = 2;                                                   [4]
while (True);                                                 [5]
   selectOne-u op =                                           [6]
      [/b @t@tDIRECTORY MANAGEMENT AND ANALYSIS]
      {Directory management}
    | Create or replace DIRECTORY database
    | Load a Directory instance
    | Load hash values of a Directory instance
    | Delete a Directory
    | Delete a Directory Instance
    | Help
   || {Directory analysis}
    | Statistics
    | Structure of a Directory instance (folders)
    | Contents of a Directory instance (folders and files)
    | List duplicates within a Directory instance
    | Describe duplicates within a Directory instance
    | Compare two Directory instances
    | Finding clones within one Directory instance
    | Finding clones within two Directory instance;
   if ('$DIALOGbutton$' = 'Cancel') exit;
   compute module = itemS('$modList$',$op$);                  [7]
   execSQL $scriptPath$/DIRECTORY-$module$.sql;               [8]
endwhile;                                                     [5]
```

Script 15.28 - Main script of DIRECTORY application

Case study **24**

Agent-based modeling and simulation

Part 1

Objectives. Agent-based modeling aims to simulate the behavior of complex real-world systems made up of a population of interacting autonomous entities. These elementary (microscopic) interactions define the overall (macroscopic) behavior of the system. For example, the expansion and reduction of a pandemic depends on the behavior of each member of the affected population. In agent-based modeling, each entity of the system is represented by a software component, called an agent, which models its main properties and behavior rules. Starting from a set of initial parameters, each execution of this model simulates a history of the complex system. The analysis of the data generated by a series of simulations enables us to better understand and predict the evolution of the complex system.

This study experiments with the development of agent-based models using database concepts. It is shown that these concepts apply naturally to the specification of the complex data used and generated by simulators. It is also shown that the *trigger* mechanism perfectly translates the behavioral rules of agents and their interactions.

In this first part of the study, we build a series of models of increasing complexity. Each is the subject of a parametric implementation enabling the global behavior of the modeled system to be visualized and analyzed.

Keywords. agent-based modeling, large system simulation, moving agent, agent life cycle, time management, trigger, UDF, active database.

Table of content

24.1 Introduction
A first illustration
Agent-based modeling
What is an agent made up of?
Agent-based simulation
ABMS applications
ABMS technologies and tools
Why use database concepts to build agent-based models?

24.2 Architecture of a simulation
24.2.1 The controller
24.2.2 The world
24.2.3 SQL implementation

24.3 The controller
24.3.1 Principles
24.3.2 SQL code
24.3.2 Firering order

24.4 Agent behavior
24.4.1 SQL modeling

24.5 Graphical representation of the world

24.6 Five (not so) easy pieces

24.7 Agents are moving: single-agent worlds
24.7.1 Trajectory styles
A. The random trajectory (Linear 0)
B. The linear trajectory (Linear 5)
C. Intermediate trajectories (Linear 1 to 4)
24.7.2 SQL modeling
A. Data structures
B. Analysis of the agent behavior
24.7.3 SQL code
24.7.4 Experimentation: the *Lawnmower*
A. The *Basic* simulator
B. The *Extended* simulator

24.8 A first multi-agent world: *Aggregation*
24.8.1 SQL modeling
A. Space structure
B. Analysis of the agent behavior
24.8.2 SQL code
24.8.3 Experimentation: the *Aggregation* simulator

24.9 Attractors and repellers

24.10 Agents and attractors
24.10.1 SQL modeling of attractors
A. Data structures
B. Analysis of the agent behavior
24.10.2 SQL code for attractors
A. SQL code for close attractors
B. SQL code for distant attractors
C. Extension of the T_AGENT trigger
24.10.3 Experimentation: the *Attractors-Repellers* simulator

24.11 Agents and repellers
24.11.1 SQL modeling of repellers
A. Data structure
B. Analysis of the agent behavior
24.11.2 SQL code for repellers
A. SQL code for close repellers
B. SQL code for distant repellers
24.11.3 Experimentation: the *Attractors-Repellers* simulator

24.12 Agent life cycle
24.12.1 Towards autonomous agents
24.12.2 SQL modeling
A. World structure
B. Agent structure
C. Analysis of the agent behavior
24.12.3 SQL code
24.12.4 Experimentation: the *LifeCycle* simulator

24.13 Agent life cycle with birth control
24.13.1 The LifeCycle model is unstable
24.13.2 Adding birth control to the LifeCycle model
24.13.3 Experimentation: the *Agent Life Cycle with birth control* simulator
24.13.4 Refining the birth control model
24.13.5 Toward learning agents

24.14 Logging elementary operations and debugging

24.15 SQL coding in SQLfast

24.1 Introduction

Let us start with a concise but instructive definition of Agent-based models:

> *Agent-based models (ABMs) are computational models that simulate behavior of individual agents in order to study emergent phenomena at the level of the community. Depending on the application, agents may represent humans, institutions, microorganisms, and so forth. The agents' actions are based on autonomous decision-making and other behavioral traits, implemented through formal rules. By simulating decentralized local interactions among agents, as well as interactions between agents and their environment, ABMs enable us to observe complex population-level phenomena in a controlled and gradual manner.*
>
> Quoted from Šešelja, Dunja, *Agent-Based Modeling in the Philosophy of Science* in *Stanford Encyclopedia of Philosophy*, Sept 2023 [https://plato.stanford.edu/entries/agent-modeling-philscience/]

A first illustration

We consider a road network comprising highways, ordinary roads and connections between them. Many vehicles circulate on this network. As the traffic becomes denser, we observe phenomena that are difficult to explain.

For example, why, in certain sections of this network, at certain times of the day, the flow of traffic is often slowed down, sometimes to a halt, although no particular upstream disturbance has been observed to explain this phenomenon?

The attitude generally adopted by the engineers responsible for the network is to build a computer model of the network and the vehicles, and run it according to different parameters. If the model is accurate, it will behave in a similar way to the phenomena observed in reality. By varying these parameters, the users of the model can establish the relationship between certain sets of the parameter values and traffic behavior. They can then decide to widen certain roads, divert part of the traffic or limit vehicle speed.

Then comes the critical question: what modelling technique should we adopt?*

One of the most popular techniques is to consider that the flow of vehicles on the network is analogous to the flow of a compressible fluid in a pipe network. Thus, by applying the equations of *fluid mechanics* to our problem, the engineers can calculate, at each point of the network, at each instant, quantities such as the speed and density of the flows of vehicles. By varying the values of the parameters of the equation system, they try to identify the critical conditions that produce the observed phenomena.*

The technique we are going to develop in this study takes the opposite approach. Instead of building a global model of the road traffic system, we will model each section of road and each individual vehicle in the form of software components called *agents*. A section of road is characterized by the condition of the surface and the number of lanes. Each vehicle is described by its dimensions, engine power and

driver's physical condition. We also describe the driver's reaction in certain circumstances, such as a reduction in the distance between vehicles, the perception of an obstacle or the dangerous behavior of other drivers.[1]

Once the agents have been coded, they are run and their behavior is examined in order to deduce the evolution laws of the traffic system.

In the first (top-down) approach, we infer the behavior of each network point from the general model. In the second (bottom-up) approach, we deduce the general rules from the observed behavior of each network point. This latter approach is known as *agent-based modeling and sim*ulation.

Agent-based modeling

An agent is an autonomous piece of software designed to simulate some of the properties and the behavior of a real world entity. So, an agent is a *model* of this entity. We consider real world entities that communicate between each other as well as with their environment. The entities we are interested in in this essay have a dynamic behavior: they move, they explore and possibly modify their environment, they exchange information, they acquire new knowledge (they learn), they look for resources, they meet, they are born, they grow up then they die. In short, they are, or behave like, living beings.

A collection of real world entities situated in a close environment constitutes a system that evolves according to a set of laws. In the computing world, the agents that simulate these entities constitute also a system driven by a set of rules that translate these laws. Since such real systems comprise a large number of agents, possibly of different species, one often speak of *multi-agent systems*.

The agent system is intended to be a model of the real system. What one expects from a good model is that the way the agents evolve be a faithful image of the evolution of the real system. In this way, the user of the agent system hopes to learn how and why the real system evolves as it does, and get enough knowledge to modify the evolution of the real system.

What is an agent made up of?

Studies available in the scientific literature and more generally on the web show that there is a huge variety of agents, from individual grains of sand in the study of beach stability during high tides to human beings in the study of pandemics.

An intuitive way of understanding the nature of the agents we'll be studying in this essay might be to adopt a *zoomorphic* analogy. For example, to study the evolution of a primate population, we model each animal as an agent to which we assign

1. Nguyen, J. et al., An overview of agent-based traffic simulators, in Transportation Research Interdisciplinary Perspectives, Vol 12, Dec. 2021, 100486 [https://www.sciencedirect.com/science/article/pii/S2590198221001913]

certain properties (age, gender, health status, social status, etc.) and specify how it reacts to certain conditions or events.

Compared with other modeling paradigms, such as object-oriented models, agent-based models exhibit some distinctive properties; among them:

- the agents of a model are *distinct* entities; to this aim they are given a unique identifier (called AgentID in the following),
- agents are defined by their *properties* and their *behavior*,
- the behavior of an agent is defined by a set of *rules* that specify how to act according to each situation,
- agents evolve in an *environment* which they can examine and modify,
- agents are *active*; they can perform tasks, such as moving, interacting with their environment and exchanging information with other agents,
- agents have a *state* which records the current (and possibly passed) level of their properties,
- agents can *learn* from experience and accumulate knowledge,
- agents are *autonomous*; they are able to decide which actions to carry out,
- agents contribute to an individual or common *objective*.

Not all agent models include all these properties. For example, cellular automata[2] are sometimes considered as agent models, despite their extreme simplicity and deterministic evolution rules.

Among the properties mentioned above, the first and most fundamental that we will study is that of the specific strategies according to which agents *move*.

Agent-based simulation

The aim of an agent system is to generate information that enables its users to understand how and why a real-world system behaves, and to predict its behavior under different conditions.

Each run of this system simulates a possible history of the real system. Starting from realistic initial conditions, the agents are prompted to evolve according to the rules of their behavior. At regular intervals during this execution, users (the observers) collect statistics from which they expect to gain a better understanding of the behavior of the real system.

The functions of a simple ABMS simulator, as conceived in the experiments below, are as follows:

- allow the user to specify the agent model,
- run a series of simulations of a model,

2. See for example Case Study 27: *Conway's Game of Life*, available on the SQLfast web site.

– visualize the evolution of the model (through still pictures or animation),
– generate statistics,
– validate the model (verify the code for programming errors, check the adequacy of the model as a faithful representation of the real-world system).

ABMS applications

TBW

ABMS technologies and tools

ABMS simulators can be coded in general languages such as Java, Python[3] and C#, sometimes with the help of specialized libraries. However, to implement large and complex agent systems, using specific ABMS platforms is strongly recommended, thanks to the richness of their functions, notably in result analysis and visualization.

According to a recent comparison of ABMS platforms and tools[4], Swarm[5], RePast[6] and NetLogo[7] are among the most frequently referenced simulation environments.

Why use database concepts to build agent-based models?

This is the central question of this study. How can a technology whose primary purpose is simply to store data be considered suitable for implementing complex ABMS systems?

Maybe an illustration of the popular adage: "*If your only tool is a hammer, every problem looks like a nail*"? To a monomaniacal database expert, couldn't every problem be solved by a database?

The question isn't entirely bizarre. Those familiar with database technology know that most DBMS are also, in a way, *application development environments*. First

3. See for example Bassel Karami, *Intro to Agent Based Modeling* [https://towardsdata-science.com/intro-to-agent-based-modeling-3eea6a070b72]
4. https://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software
5. https://en.wikipedia.org/wiki/Swarm_(simulation)
6. https://en.wikipedia.org/wiki/Repast_(modeling_toolkit); https://repast.github.io/index.html
7. Wilensky, U., Rand, W., *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*, The MIT Press, 2015

and foremost, SQL is a universal and powerful language, particularly suited to the creation and management of large, complex and rapidly evolving data sets. Given the volume, variety and complexity of the data used and generated by ABMS simulators, storing this data in a database is the first thought that comes to mind.

Secondly, and most decisively, SQL includes the *trigger* mechanism, a powerful ECA (*Event Condition Action*) function that converts a passive data repository into an *active database*[8].

Consider a table TA whose rows represent the state of objects of type A, for instance *agents*. An SQL trigger associated with this table is the immediate translation of the type of rules that make up the agents' behavior:

- **event**: whenever an event affects an object of type A
- **condition**: if some condition is met,
- **action**: then this object is asked to execute this action.

These components are directly translated into the trigger syntax:

```
create trigger Tr_TA
after/before <data modification event> on TA
when <condition to be satisfied>
begin
   <action to execute>
end;
```

24.2 Architecture of a simulation

The simulators we will develop comprise two main components: the controller and the world. The controller is common to all the agent systems while the world is the implementation of a specific agent system.

24.2.1 The controller

The controller's responsibility is to create the space in which the agents will live and evolve, to populate it with a collection of agents and with the resources they may need, to start the simulation, to control the evolution of the world and to extract the information we would like to know about this evolution, such as statistics and static or dynamic graphical representation.

A critical question is the way the simulation makes the world evolve, that is, *time management*. The theory of agent-based simulation suggests two main strategies: discrete event simulation and discrete time simulation.

In the first strategy, *discrete event simulation*, each agent writes the events that will affect it (for instance, the date it will get sick after being infected) on the time

8. See for example Case Study 8: *Active databases*, available on the SQLfast web site.

line. Then the controller advances from date to date to activate the concerned agents.

In the *discrete time (or clock-based) simulation*, the evolution of the world is decomposed into a sequence of steps, corresponding to (generally small) equal size time slices. During each time slice, all the agents have the opportunity (or are required) to execute a small step of their *life*, that is, the scenario which they are programmed for. At the completion of a step, the world is in a new state. For small time slices, it can approximate a continous behavior of the world.[9] The controller triggers each step and analyzes each generation, not only to extract useful information and to show graphically the state of the world, but also to check that the conditions of the simulation are still met (for instance, there should be enough rabbits to feed the foxes, otherwise, they will all die, which will terminate the simulation).

The strategy we will implement in our simulators is a variant of the latter technique in which the duration of a slice is the time needed for all agents activated in the step to complete the actions required by the rules that define their behavior.

To start each step, the controller sends a *message* to some agents (all of them or a selected subset) to make them act according to their behavior rules. In the simulation, each active agent analyzes its environment, moves to a neighbor cell according to the result of this analysis then applies its evolution rules. The step stops when all the agents that were activated have finished their actions.

At the completion of each step, the controller examines the state of the world and records the desired statistics. It also determines when the simulation must be terminated.

24.2.2 The world

One generally distinguishes the agents from their environment, that is, the virtual space in which the agents act.

The world is made up of a finite space[10] structured as a grid of cells identified by their (X,Y) coordinates. This space is populated with a collection of agents. The agents can be of a single kind (for example, cars in a road network) or, in more complex worlds, agents of different types can coexists and interact (such as rabbits and foxes). Each cell may contain one or more agents as well as, in some cases, other objects the agent can interact with. The agent is represented by its properties, among them the coordinates of the cell it is located in. These properties form the

9. In the last two experiments of this part (*Life cycle* and *Life cycle with birth control*), the agents are given two new properties, Ticks and Latency, implemented as autoincrementing counters that specify future time points on which the agent acquires new capabilities. They provide a limited but inexpensive substitute to the scheduling facility on which the *discrete event simulation* is based.

10. Some agent models do not impose the finiteness constraint of their space. Such is the case of the Conway's cellular automata, already cited, which include no boundary conditions. They are not considered here.*

state of the agent. An agent can examine the content of any cell, besides that in which it resides. It can receive a message, in which case it reacts in a way that depends on the nature of the message and on its environment. It can also send messages to other agents and examine their properties.

In this essay, we consider that the agents are free to move across the cells of the space. Therefore, the way they move is an important aspect of the evolution of the world.

24.2.3 SQL implementation

The world is represented by two main tables, CELL and AGENT. Each row of CELL describes the properties of a cell: its absolute *coordinates* (columns X and Y) and its *content* (column Content), the precise meaning being left undefined for now. Each row of AGENT contains the properties of an agent, among them, its *Id* (column AgentID) unique among all the agents, the *coordinates* of the cell it is currently located in (columns X and Y), and the last *message* it received and to which it is currently reacting (column Message). To send a message to an agent the sender just updates the value of the Message column of the row of this agent, an event that will be caught by the appropriate trigger.

This is translated into the declarations of Script 24.1. Except for the primary key of AGENT, considering the strict way the data will be managed, we have defined no constraints. Other properties will be added according to the needs of the different agent systems we build.

```
create table AGENT(AgentID integer not null primary key,
                   X       integer,
                   Y       integer,
                   Message varchar(16) default '');
create table CELL(X       integer,
                  Y       integer,
                  Content integer default 0);
```

Script 24.1 - The main tables describing the world (first draft)

The space is defined by its width and height dimensions (in terms of cells), specified by variables **width** and **height**. A space of 60 rows of 80 cells is defined as follows:

```
set width = 80
set height = 60
```

If the rectangular shape of the space is considered inappropriate, more complex configurations can be built by placing special objects in certain cells to prohibit their access. They will be studied in Section 24.11 in the form of *repellers*.

24.3 The controller

24.3.1 Principles

The controller comprises four main operations:

– acquisition of simulation parameters, in particular the number of steps to be executed,

– creation of the initial state of the world, including the environment and the starting set of agents,

– execution of the specified number of simulation steps,

– processing the collected data, in particular graphical representation of the successive states of the world.

The last operation is important for the ABM simulations we plan to develop. We decide to display each state of the world graphically so that its evolution appears like an animated movie. After completion of each step, the description of the current state is stored in a graphic file using the SQLdraw graphics language.

The controller structure can then be refined in the following pseudocode:*

```
Create the tables
Get the simulation parameters
Create the world
Populate the world
Generate the initial state in the graphic file
for step = [1,maxStep]
    Send an activation message to the agents
    Generate the current state in the graphic file
endfor
Process the simulation history
```

We decide to include an intermediate concept, the clock, materialized by the CLOCK table. To start each step, the controller sends a message to the clock by inserting a row containing the sequence number of that step. Then, the clock sends an activation message to the selected agents[11]. Distributing the execution responsibility enables us to build a simpler, more abstract controller.

Now, we study the implementation of this architecture in SQL.

24.3.2 SQL code

The code of the controller is independent of the agent systems it activates and controls. Each agent system is materialized by a simulator defined by six procedures called by the controller:

11. Actually, all of them in the first part of this study

- AGENT-Create-Tables: creates the base tables of the simulator.
- AGENT-Get-parameters: asks the user for the parameters of the next simulation.
- AGENT-Create-World: specifies the behavior of the agents
- AGENT-Populate-World: if needed, places special objects in the space (such as obstacles or resources); creates the initial set of agents and places them in the space.
- AGENT-Generate-Current-State: analyzes the current state of the world and stores its description in historical data structures for further processing; this description is translated into graphical commands and, for more complex agent systems, into statistical measurements.
- AGENT-Process-History: the graphical and statistical descriptions of successive states of the world created during simulation are translated into usable information for the user.

The code of the SQLfast controller is shown in Script 24.2.

```
execSQL _AGENT-Create-Tables.sql;
execSQL _AGENT_Get-parameters.sql;
execSQL _AGENT-Create-World.sql;
execSQL _AGENT-Populate-World.sql;
execSQL _AGENT-Generate-Current-State.sql;

for step = [1,$maxSteps$];
  insert into CLOCK values($step$);
  execSQL _AGENT-Generate-Current-State.sql;
endfor;

execSQL _AGENT-Process-history;
```

Script 24.2 - General structure of the monitor

The structure of the CLOCK table is shown in Script 24.3. It associates a trigger to the table to catch the event created by the controller.

```
create table CLOCK(Tick integer);

create trigger T_CLOCK after insert on CLOCK
begin
  <step initialization>
  update AGENT set State = 'move';
  <step closure>
end;
```

Script 24.3 - The CLOCK table and its trigger

Its body itself creates an `update` event to send a message ("move", in this example) to the rows of the AGENT table. The value of this message as well as the (optional) initialization and closure sections of the body depend on the specific agent system.

Each simulator developed in this case study is available in two versions:

– the *Basic version* comprises the structures and algorithms discussed and developed in this chapter,

– the *Extended version* offers more parameters to provide detailed information on the simulation execution, such as traces of elementary actions, either to give the user a better understand of how the simulators work or for debugging purposes. The names of the procedures in these versions are suffixed with `'-extended'`. This code is of little interest for the purposes of this presentation and will not be described in detail.

24.3.3 Firering order

Because of the way the SQL engine works, the clock manager (Script 24.3) sends the same message to all agents *in a sequential order*. This order implementation dependent and is often the order of insertion or is defined by the values of the primary key: the first agent in this order will receive this message first, and the last agent will also receive it last. At each step of the simulation, it is highly likely that the same order will be followed, introducing an undesirable bias into the process.

There is a special form of the `update` query that allows a specific order to be imposed on its execution via an `order by` clause. In our case, this order is *random*, so that, for a large number of steps, all the agents tend to get an equal chance of being processed first (or last):

```
update AGENT set State = 'move' order by random();
```

However, this feature is not provided by all the SQL engines. It can be emulated by an SQLfast `for` loop:

```
for AiD = [select AgentID from AGENT order by random()]
   update AGENT set Message = 'move' where AgentID = :AiD;
```

or, in a more declarative way, by encapsulating the random order in a view[12]:

```
create view RANDOM_AGENT as
   select * from AGENT order by random();
create trigger R_AGENT
instead of update of Message on RANDOM_AGENT
```

12. A view being a (virtual) table, its rows are supposed to be unordered. While the concept of *ordered view* is a convenient extension, it violates the principles of the relational theory of databases.

```
begin
   update AGENT set Message = new.Message
   where  AgentID = old.AgentID;
end;
```

24.4 Agent behavior

An agent's behavior is defined by the actions it must perform in response to a certain type of message. Sending a message involves updating the Message column of the AGENT rows of the recipients concerned. In the first agent systems we are going to build, we will consider a single message, "`move`". In more complex agent systems, agents will have to react to several different messages.

24.4.1 SQL modeling

As explained in the *Introduction* section, the behavior of an agent is expressed in the form of ECA rules, the privileged implementation of which is via SQL triggers. Attached to the AGENT table, these triggers capture `after` events created by the updating the Message column of AGENT rows (`after update of Message on AGENT`). It fires under two conditions:

- the new value of Message is `'move'` (**when** `new.Message = 'move'`),
- (optionally) the current row satisfies a certain condition (`and <Condition>`),

The procedure of the trigger body is consists of a sequence of actions. Each action translates into an SQL query that can modify the state of the agent[13] or of the space of the world.

```
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
and  <Condition>
begin
    <Action 1>
    <Action 2>
    <...>
    <Action n>
end;
```

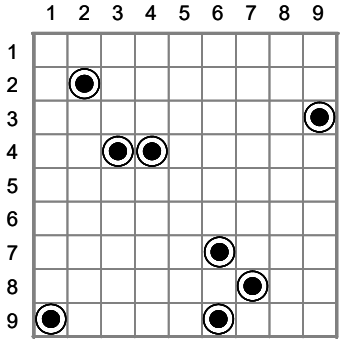
Script 24.4 - Implementing the behavior of agents via SQL triggers

13. or, in some agent systems, of the neighbors of the current agent

24.5 Graphical representation of the world

Graphical representation of the state of the world is an important function of agent-based simulators. In the family of models we are studying in this essay, we wish to identify each cell and each agent in this representation[14]. In the implementation of the different models, the space and its content will be displayed as a vector graphic image (Figure 24.1, left side) or as a simple text (right side). This figure shows a square space of 81 cells, 8 of which contain an agent.

Most simulators introduce additional properties (the gender of an agent, for example) and other objects (such as attractors and repellers). Their specific graphical representation will be described when needed.




```
  1 2 3 4 5 6 7 8 9
1 . . . . . . . . .
2 . 1 . . . . . . .
3 . . . . . . . . 1
4 . . 1 1 . . . . .
5 . . . . . . . . .
6 . . . . . . . . .
7 . . . . . 1 . . .
8 . . . . . . 1 . .
9 1 . . . . 1 . . .
```

Figure 24.1 - Two graphical representations of a 9x9 space

24.6 Five (not so) easy pieces[15]

We now have a generic *framework* from which to build a series of five agent models of increasing complexity.

- **The lawnmower** (Section 24.7): This model experiments with different trajectory styles. The world comprises a single agent whose sole task is to move according to one of these styles.
- **Aggregation** (Section 24.8): Now, the world includes several agents of the same type. These agents move according to one of the trajectory styles. When

14. This would make no sense in models based on a very large number of very small objects such as the one that represents each grain of sand by an agent to simulate the evolution of the sea shore as a function of tidal height. In such cases, the graphical representation comprises aggregates of the system, such as dm^3 or m^3 of sand.
15. Tribute to Bob Rafelson's eponymous movie (1970)

an agent visits a cell already hosting an agent, it forms an aggregate, which is a static object. If this cell contains an aggregate, the agent joins it.

– **Attractors and repellers** (Section 24.9): In addition to a collection of agents, the world includes static objects, *attractors* and *repellers*. Agents attempt to join one of the attractors, but try to avoid the repellers in their vicinity.

– **Agent lifecycle** (Section 24.12): In this model, agents behave like living entities: they have two genders, are born, grow old, mate, generate new agents and die.

– **Agent lifecycle with birth control** (Section 24.13): It appears that the life cycle model is intrinsically unstable: the population of agents tends to grow without limit and fill space, or, on the contrary, to decrease until extinction. In this model, we introduce a rule that links the fertility rate to the population size.

The Agent application (Agent-MAIN.sql) opens a selection box for the five simulators developed in this case study (Figure 24.2). Both *Basic* and *Extended* versions are available

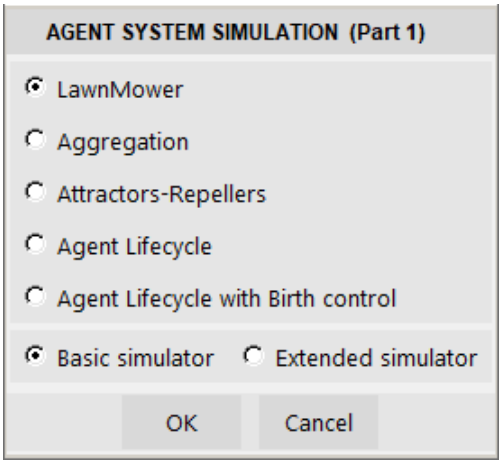


Figure 24.2 - The general selection box

24.7 Agents are moving: single-agent worlds

Although this first agent model may seem particularly simplistic, or even uninteresting, it will enable us to tackle and solve fundamental problems that will form the building blocks of all the models we develop in the rest of this study.

24.7.1 Trajectory styles

The common feature of the model we will build, agents *move around* in the space of the world. More precisely, at each step of the simulation, an agent can decide to occupy a cell in its direct environment (Figure 24.3).

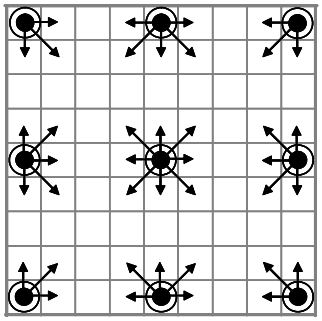


Figure 24.3 - The unconstrained valid moves of an agent

The choice the agent makes depends on its state, its goals and the characteristics of its current cell and of those of surrounding cells. It also depends on the style of trajectory the agent is trying to adopt, so that the question arises as to the shape of the trajectory.

We first consider two extreme styles: *random* and *linear*, called *Linear_0* and *Linear_5* respectively.

A. The random trajectory (*Linear 0*)

At each step of a *random trajectory*, the next position is selected at random from the surrounding cells. It is also possible that the agent decides to stay at its current position. The choice is therefore made from among nine cells. The screenshot on the left-hand of Figure 24.4 shows the random trajectory of an agent in an 8x24 space, where each visited cell is colored green.

B. The linear trajectory (*Linear 5*)

In a *linear trajectory*, on the other hand, each step extends the previous move (Figure 24.4, right-hand screenshot). On contact with a horizontal or vertical border of the space, the agent bounces off, theoretically at a symmetrical angle, as illustrated by its behavior on encountering the upper border. However, this rule risks leading to cyclical trajectories in which the agent occupies the same subset

of cells indefinitely. To counter this phenomenon, we introduce a random perturbation in the choice of the next cell during a bounce. This is illustrated by the agent's behavior when it encounters the left and bottom borders of the aforementioned figure.

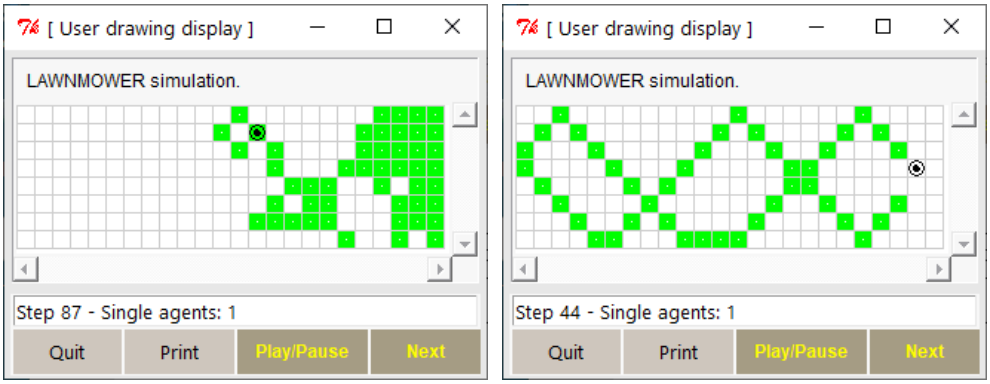


Figure 24.4 - Random (left) and linear (right) trajectories of an agent in a 8x24 world

C. Intermediate trajectories

Between these two trajectory styles, we define four variants of increasing linearity (and therefore decreasing *randomness*), denoted *Linear 1* to *Linear 4*. In each case, the next direction is chosen from a list in which the linearity (pure extension) is increasingly likely. In *Linear 1*, trajectories appear random, but with a slight tendency to extend the previous move. In *Linear 4*, trajectories are largely linear, but occasionally show a random change of direction.

These proposals raise the question of whether an agent (or, more generally, several agents) can cover the space according to each of the trajectory styles. If an agent is given enough time, will it eventually visit all the cells in the space, and if so, which trajectory style will be the fastest to achieve this goal? This is what a short series of experiments seem to show:

- whatever the dimensions of the space, all trajectory styles lead to total coverage of the space,
- not surprisingly, the last cells to be visited consume more and more steps; in particular, visiting the last cells may require as many steps as accessing all the previous ones,
- on the average (measured over some twenty trials for each), despite a very high variance, styles 3 and 4 cover the cells of the space the fastest.

This model could be used to simulate the behavior of a *lawnmower*[16]. However, the latter are generally supposed to operate more intelligently, which could translate into more elaborate trajectory styles!

24.7.2 SQL modeling

In this first study, we examine the behavior of a single agent whose sole aim is to move from one cell to one of its neighbors according to an imposed trajectory style. In brief:

- the space is finite and rectangular (bounded by four linear borders),
- only one agent is active, so no interaction with other agents, or with special objects in the space,
- the agent is located in a cell specified by its coordinates in the space,
- at each step of the simulation, the agent executes an elementary move to an adjacent cell,
- this move is determined by the selected trajectory style.

The travel of the agent is based on the following principle:

To execute the next move, the agent computes its new direction as a unitary vector[17] *($\Delta x, \Delta y$) to add to its current position (X,Y). Then, the agent moves to the cell ($X+\Delta x, Y+\Delta y$).*

A. Data structures

In all the trajectory styles (but one, *Linear 0*), the next move depends on the current direction, that is, on the previous move. We store the previous ($\Delta x, \Delta y$) vector in new columns OldX,OldY (Script 24.5).

```
create table AGENT(AgentID integer not null primary key,
                   X       integer,
                   Y       integer,
                   OldX    integer default 0,
                   OldY    integer default 0,
                   Message varchar(6) default '');
```

Script 24.5 - Thanks to columns OldX and OldY the agent remembers the direction of its previous move

Besides its (X,Y) coordinates, each cell is described by two new properties: the number of agents it contains (Content column), and the trace of an agent's passage

16. Hence the name we give to this agent model
17. Here, by *unitary*, we mean that each of its coordinates is in the set (-1,0,1).

(Trace column) (Script 24.6). The latter column is set to 1 whenever an agent visits this cell.

```
create table CELL(X integer,
                  Y integer,
                  Content integer default 0,
                  Trace   integer default 0);
```

Script 24.6 - The cells keep the trace of the passage of the agents

B. Analysis of the agent behavior

The only task of the agent is to move across the space according to a definite style such as those illustrated in Figure 24.4.

The rules that define the trajectory styles rely on the concept of *vicinity.* We call *vicinity* - or *neighborhood* - of a cell (as well as of the agent it hosts) the set of its neighboring cells. In all the trajectory styles, the *next move* of an agent sends it in one of the cells of its vicinity. In some case, the next move may also be a *no-move*, that leaves the agent in its current cell.

At each step, each agent computes its next move as the vector (Δx, Δy) to add to its current position (X,Y). Then, the agent moves to cell (X+Δx,Y+Δy).

The computing rules of (Δx, Δy) depend on the trajectory style. However, whatever this style, the next move of the agent *when it has reached a border* sends it to one of the cells of its vicinity, randomly selected.

Now, we examine the rules that dictate the selection of the next move when the agent is *in the middle of the space*. We focus first on the two extreme styles, namely pure random (*Linear 0*) and pure linear (*Linear 5*)

- **The random trajectory (*Linear 0*)**

  The random trajectory is defined by a simple rule: the next move is randomly chosen from among the eight cells of its vicinity plus the current cell. It is therefore strictly independent of the previous move (the agent ignores its OldX and OldY properties). This rule is defined as follows for agents in a *non-border* cell[18]:

```
(Δx,Δy) = random('-1,-1;-1,0;-1,1;0,-1;0,0;0,1;1,-1;1,0;1,1')
```

18. Used without argument, the `random()` function returns a random number (named `rand()` in some SQL versions). Used with an *string-list* argument, `random(S)` is a function that randomly selects an element of `S`. A *string-list* is a character string formed by a sequence of values separated by a specific symbol (*csv* is a popular *string-list* format). In this example, `S` serializes a list of couples '`x,y`' separated by semi-colons. This interpretation of function `random` is specific to SQLfast.

- **The linear trajectory (*Linear 5*)**

  In a linear trajectory (*Linear 5*), each move is a mere copy of the preceding one, defined by the properties OldX and OldY of the agent. So, for agents in a *non-border* cell:

  ```
  (Δx,Δy) = (OldX,OldY)
  ```

Before developing the intermediate styles, we solve the next step problem for the agent that has reached a border of the space.

- **Agent at the border: the bouncing rules**

  We distinguish eight current positions (see Figure 24.3), each requiring a specific reaction[19]:

  – top left corner: `X=1 and Y=1`
    `random('1,0;1,1;0,1')`
  – bottom left corner: `X=1 and Y=:height`
    `random('0,-1;1,-1;1,0')`
  – top right corner: `X=:width and Y=1`
    `random('-1,0;-1,1;0,1')`
  – bottom right corner: `X=:width and Y=:height`
    `random('-1,0;-1,-1;0,-1')`
  – left border, middle: `X=1 and Y not in (1,:height)`
    `random('0,-1;1,-1;1,0;1,1;0,1')`
  – right border, middle: `X=:width and Y not in (1,:height)`
    `random('0,1;-1,1;-1,0;-1,-1;0,-1')`
  – top border, middle: `X not in (1,:width) and Y =1`
    `random('1,0;1,1;0,1;-1,1;-1,0')`
  – bottom border, middle: `X not in (1,:width) and Y=:height`
    `random('-1,0;-1,-1;0,-1;1,-1;1,0')`

Let us now study the four intermediate trajectory styles, denoted *Linear 1* to *Linear 4*. We start with the *Linear 1* style, from which we will derive *Linear 2* to *Linear 4*.

- **The pseudo-random trajectory (*Linear 1*)**

  The agent selects its next step from three candidate directions:

  – extending the preceding step,
  – the next cell clockwise
  – the next cell counter-clockwise.

19. The suggested moves do not include the `(0,0)` *no-move*.

This rule is illustrated in Figure 24.5 for four representative cases (the other four are symmetrical). The large solid dot represents the current position of the agent, the incoming arrow its preceding step, and the outgoing arrows the three candidate next steps.

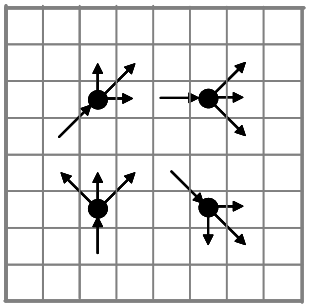


Figure 24.5 - Selection of the next step in the Linear 1 style

These patterns are translated into the rules of Figure 24.6. The remaining four patterns are specified similarly. We observe that the linear direction (in blue) is in the middle of the list.

(OldX,OldY)	(Δx,Δy)
(1,-1)	random('0,-1;**1,-1**;1,0')
(1,0)	random('1,-1;**1,0**;1,1')
(1,1)	random('1,0;**1,1**;0,1')
(0,-1)	random('-1,-1;**0,-1**;1,-1')

Figure 24.6 - Moving rules for the *Linear 1* trajectories

We still must address a special case, the *starting step*, for which there is no preceding step. By convention, the (OldX,OldY) properties are set to (0,0) and the next step is selected among the eight adjacent cells, as in the *Linear 0* style.

- **From pseudo-random to linear (*Linear 2* to *Linear-4*)**

  To augment the linearity trend of the pseudo-random style, we duplicate its linear component (which is a copy of the preceding step) in the candidate list (Figure 24.7). Now, the probability that the next step extends the preceding step is 0.5 (2 out of 4) instead of 0.33 (1 out of 3).

(OldX,OldY)	(Δx,Δy)
(1,-1)	random('0,-1;**1,-1;1,-1**;1,0')
(1,0)	random('1,-1;**1,0;1,0**;1,1')
(1,1)	random('1,0;**1,1;1,1**;0,1')
(0,-1)	random('-1,-1;**0,-1;0,-1**;1,-1')

Figure 24.7 - Moving rules for the *Linear 2* trajectories

By tripling, quadrupling, etc., this transformation, we build trajectories that look increasingly linear, such as *Linear 3* and *Linear 4*.

24.7.3 SQL code

The behavior of the agent is triggered by sending it a `'move'` message. This is performed by updating the Message column of the agent row:

```
update AGENT set Message = 'move';
```

Before examining the translation of this behavior into a SQL trigger, it's worth noting that, in the body of all the triggers we will develop in this study, the two aliases, *old* and *new*, denoting the states of the current row before (*old*) and after (*new*) the `update` operation are identical, with the exception of the value of the Message column. Consequently, for example, designating the current row identifier by the expressions "old.AgentID" or "new.AgentID" is strictly equivalent.

Architecture of the main trigger

The T_AGENT trigger is activated whenever the Message column of an AGENT row has been updated and when the new value is `'move'`. The reaction is decomposed into four successive tasks, each expressed as a single SQL query (Script 24.7).

Let us analyze each of these tasks.

```
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
begin
    <1. The agent quits its current position>
    <2. The agent computes its next direction>
    <3. The agent computes its next position>
    <4. The agent moves to its new position>
end;
```

Script 24.7 - Architecture of the main trigger

1. The agent quits its current position

The content of the current cell is decremented by one and this cell is marked to indicate that an agent once visited it.

```
update CELL
set    Content = Content - 1,
       Trace = 1
where (X,Y) = (old.X,old.Y);
```

Script 24.8 - The agent quits its current cell

2. The agent computes its next direction

This is the central activity of the agent. We suggest to design a generic query frame that provides a simple way to implement the six trajectory styles identified above (and any extensions of them).

Each style is defined by a series of lists of (Δx, Δy) directions. Each list specifies the recommended directions for one of the eight directions that has lead the agent in its current cell and that are stored in columns (OldX,OldY). For example, when the last directions was (1,1), the *Linear 2* strategy suggests to select the next direction in this list: `'1,0;1,1;1,1;0,1'`. If the agent selects the direction `'1,1'`, it will extend its preceding direction.

The specification of four of the six trajectory styles is shown in Script 24.9. A style is described by eight lists, one for each (OldX,OldY) couple.

These lists are stored in variables A to H. The special list in variable V (for ***V**icinity*) is used to start the simulation, when columns (OldX,OldY) have not been set yet, and in the random trajectory style.

The current style is denoted by the `linear` variable. We note that the rules of *Linear 0* do not depend on the preceding direction and the rules of *Linear 5* strictly depend on the preceding direction only.

The generic query that computes the next position is structured as shown in Scripts 24.10 and 24.11. It returns in SQL variables `DeltaX` and `DeltaY` the direction the agent will apply according to the trajectory style and its current position in the space. It is instantiated by the values of these static variables:

- `width`, `height`: dimensions of the space
- V: vicinity of the agent (list of the eight directions of the vicinity of the agent, to which we add the *no-move* (0,0))
- `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`: list of candidate (Δx, Δy) directions according to the last step of the agent.

```
set V = '-1,-1;-1,0;-1,1;0,-1;0,0;0,1;1,-1;1,0;1,1';
if ($linear$ = 0);
   set A = $V$;
   set B = $V$;
   set C = $V$;
   set D = $V$;
   set E = $V$;
   set F = $V$;
   set G = $V$;
   set H = $V$;
endif;
if ($linear$ = 1);
   set A = '1,1;0,1;-1,1';
   set B = '-1,-1;0,-1;1,-1';
   set C = '1,-1;1,0;1,1';
   set D = '1,0;1,1;0,1';
   set E = '0,-1;1,-1;1,0';
   set F = '-1,1;-1,0;-1,-1';
   set G = '0,1;-1,1;-1,0';
   set H = '-1,0;-1,-1;0,-1';
endif;
if ($linear$ = 2);
   set A = '1,1;0,1;0,1;-1,1';
   set B = '-1,-1;0,-1;0,-1;1,-1';
   set C = '1,-1;1,0;1,0;1,1';
   set D = '1,0;1,1;1,1;0,1';
   set E = '0,-1;1,-1;1,-1;1,0';
   set F = '-1,1;-1,0;-1,0;-1,-1';
   set G = '0,1;-1,1;-1,1;-1,0';
   set H = '-1,0;-1,-1;-1,-1;0,-1';
endif;

...

if ($linear$ = 5);
   set A = '0,1';
   set B = '0,-1';
   set C = '1,0';
   set D = '1,1';
   set E = '1,-1';
   set F = '-1,0';
   set G = '-1,1';
   set H = '-1,-1';
endif;
```

Script 24.9 - Definition of trajectory styles *Linear 0*, *Linear 1*, *Linear 2* and *Linear 5*

```
select Next[1],Next[2] into DeltaX,DeltaY
from (select
        case
        -- bouncing moves
           when old.X in (1,$width$) or old.Y in (1,$width$)
           then <compute the bouncing step [Script 24.11]>
        -- starting move
           when (old.OldX,old.OldY) = (0,0)   then random($V$)
        -- internal moves
           when (old.OldX,old.OldY) = (0,1)   then random($A$)
           when (old.OldX,old.OldY) = (0,-1)  then random($B$)
           when (old.OldX,old.OldY) = (1,0)   then random($C$)
           when (old.OldX,old.OldY) = (1,1)   then random($D$)
           when (old.OldX,old.OldY) = (1,-1)  then random($E$)
           when (old.OldX,old.OldY) = (-1,0)  then random($F$)
           when (old.OldX,old.OldY) = (-1,1)  then random($G$)
           when (old.OldX,old.OldY) = (-1,-1) then random($H$)
        end as Next
     );
```

Script 24.10 - Computing the next direction

```
case
   when old.X=1 and old.Y=1 then random('1,0;1,1;0,1')
   when old.X=1 and old.Y=$height$ then random('0,-1;1,-1;1,0')
   when old.X=$width$ and old.Y=1 then random('-1,0;-1,1;0,1')
   when old.X=$width$ and old.Y=$height$
   then random('-1,0;-1,-1;0,-1')
   when old.X=1 and old.Y not in (1,$height$)
   then random('0,-1;1,-1;1,0;1,1;0,1')
   when old.X=$width$ and old.Y not in (1,$height$)
   then random('0,1;-1,1;-1,0;-1,-1;0,-1')
   when old.X not in (1,$width$) and old.Y =1
   then random('1,0;1,1;0,1;-1,1;-1,0')
   when old.X not in (1,$width$) and old.Y=$height$
   then random('-1,0;-1,-1;0,-1;1,-1;1,0')
end
```

Script 24.11 - Computing the bouncing steps

The query frame comprises three sections:

– The first section defines the behavior of the agent when it has reached a border of the space. It is developed in Script 24.11, which translates the bouncing rules

suggested above (title *Agent at the border: the bouncing rules*). This behavior is common to all the trajectory styles.

– The second section specifies the starting behavior of the agent. The first step is selected in the agent vicinity. It, too, is common to all the trajectory styles.

– The third section states the eight rules of which the Figures 24.6 and 24.7 are excerpts. These rules select at random one of the valid directions for each preceding move (old.OldX,old.OldY).

Remarks

The variables appearing in SQL queries and, more generally, in the algorithms we develop in this study are of two kinds: *static* and *dynamic*.

A *static variable* is a parameter of the world that is set before this world is built and remains constant during the simulation. The dimensions of space, width and height, are such variables. When used in the definition of the world, they appears as `$<variable>$` anywhere in a query or in an algorithm. For example, if variable `width` has been set to `24`, the expression "`where X = $width$`" must be read "`where X = 24`" throughout the simulation run.

A *dynamic variable* describes the instantaneous state of an object during the simulation. Its value can change during the execution of the current step and is usually computed by an SQL query. When used in a query, it appears in the standard SQL syntax `:<variable>`. Assuming that the future direction of an agent has been stored in variables `DeltaX` and `DeltaY`, the clause "`set X = X + :DeltaX`" of an `update` query computes the new `X` coordinate of this agent.

The queries of Scripts 24.10 and 24.11 are comprised in the body of the agent trigger and are executed during each simulation run, while the algorithm of Script 24.9 is executed at the initialization stage, when the world is built.

3. The agent computes its next position

The new direction is added to the agent current position and saved in the OldX and OldY columns. The Message column, no longer needed, is cleared (Script 24.12).

```
update AGENT
set    X = X + :DeltaX,
       Y = Y + :DeltaY,
       (OldX,OldY) = (:DeltaX,:DeltaY),
       Message = ''
were   AgentID = old.AgentID;
```

Script 24.12 - Computing the next position of the agent

4. The agent moves its new position

The content of the cell defined by the new position of the agent is incremented by one (Script 24.13). Note that (new.X,new.Y) do not denote the new position of the agent. Indeed, these values refer to the old and new states of the agent *when the Message column was updated*. To get the *current values* of X and Y, we must extract them from the database[20].

```
update CELL
set    Content = Content + 1
where (X,Y) = (select X,Y from AGENT
               where  AgentID = old.AgentID);
```

Script 24.13 - The new cell of the current agent is updated

Toward a smarter lawnmower

In an earlier section, we mentioned that the last cells to be visited consume more and more steps. As an example, the state of the world illustrated in Figure 24.8 has been reached in 620 steps. However, visiting the remaining six empty cells will require no less that 600 additional steps! Moreover, observing the movements of the agent is particularly frustrating. Quite often, its vicinity includes an empty cell that the agent blindly ignores in its next moves, missing the opportunity to reach its goal more quickly. Not to mention, in multi-mower configurations, the agents that roll over each other when they occupy the same cell!

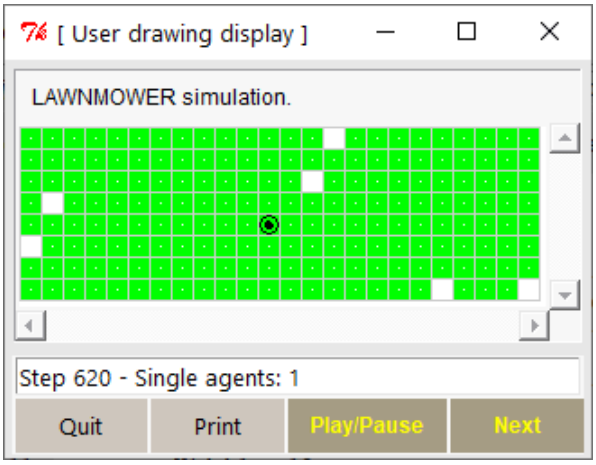


Figure 24.8 - State of the world after 620 steps.

20. or recalculate them as we did in Script 24.12, which would be less *SQL-minded*!

Could we improve the agent's behavior (its *intelligence*) by enabling it to consult its environment and, if necessary, to redirect its trajectory towards an empty cell? Certainly a good idea, as long as this process is only activated when the space contains only a small number of empty cells. Otherwise, the notion of trajectory style would become meaningless.

24.7.4 Experimentation: *The Lawnmower*

The control panel of the *Basic* module devoted to single moving agent (aka *Lawnmowers*) is shown in Figure 24.9. It comprises two elementary boxes.

The first box (1. STARTING PARAMETERS OF THE WORLD) defines the space dimensions, the maximum number of steps of the simulation, the delay between steps (in ms.) in the animated graphical representation and the trajectory style (from 0 to 5).

The second box (2. THE LAWNMOWERS) is a *bonus feature*; it allows the experimenter to create more than one agent. Note that, in this case, agent can run over another without damaging it!

A. The *Basic* simulator

The basic *LawnMower simulator* is implemented through the script **MAIN-LawnMower**. It calls 7 procedures that, together, define the agent system:

- _AGENT-Init-Parameters: assign initial values to agent system parameters.
- _AGENT-Create-Tables: creates of the tables.
- _AGENT-Get-Parameters: asks the user the parameters of the agent system.
- _AGENT-Create-World: creates the triggers that specify the behavior of the agents.
- _AGENT-Populate-World: initializes the world by placing an agent (or several agents) in random cell(s).
- _AGENT-Generate-Current-State: when a step is finished, writes in file AGENT.draw the commands that draw the content of the cells.
- _AGENT-Process-History: displays the evolution of the agent system as an animated movie.

Three ancillary procedures are independent of the agent system:

- _AGENT-Trajectories, called by _AGENT-Create-World, initializes the set of variables V, A, B, C, D, E, F, G, H containing the lists of candidate directions as defined in Script 24.9.

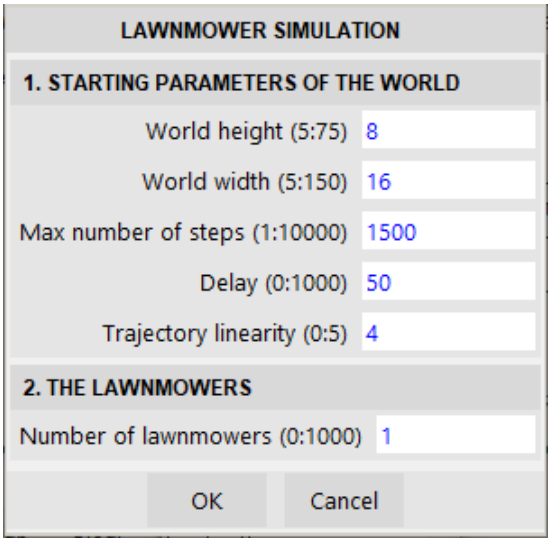


Figure 24.9 - The control panel of the basic *Lawnmower* simulator

– _AGENT-Generate-Grid, called by _AGENT-Populate-World, writes in file AGENT.draw the commands drawing the lines that delimit the cells.

– _AGENT-Clear-Tables: empties the base tables.

B. The *Extended* simulator

The *Extended* version of the simulator includes additional functions by which the experimenter can be provided with more detail on the simulations.

In particular, it records a synthesis of the state of the world after each step in the JOURNAL table, structured as follows:

```
create table JOURNAL(StartTime timestamp,
                     Step      integer,
                     Traces    integer);
```

Each record comprises the timestamp of the start of the simulation, the index of the step and the cumulative number of cells visited. Once completed, this table can be queried or represented graphically.

The control panel of the *Extended* version is shown in Figure 24.10. It includes 5 elementary boxes, two of which are the same as those of the *Basic* version: (STARTING PARAMETERS OF THE WORLD) and (THE LAWNMOWERS). The three new boxes offers the following features.

– 2. WORLD CREATION MODE: this box lets the user choose whether to create a new starting configuration (defining the number and initial position of the agents) or to reuse the previous one.
– 3. INFORMATION: this box specifies the tracing information to print in the SQLfast output window. It can be used to observe the detailed behavior of each agent at each step or to debug a model.
  • Show start state: print the state of the AGENT table before the simulation.
  • Show final state: print the state of the AGENT table when the simulation is finished.
  • Show grid: at the end of each step of the simulation, print the state of the cells; the format is that shown in Figure 24.1/right-hand side; this feature is useful for small spaces, up to 10x10.
  • Trace moves: print the detail of each move of each agent in each step.
  • Show AGENT table: at the end of each step of the simulation, print the state of the AGENT table.
  • Graphic animation: display the animated graphical representation of the simulation.
– 5. DISPLAY JOURNAL: the last box is used to write the contents of the JOURNAL table in various formats to the output window. At the end of each step, the controller inserts into this table either the cumulative number of cells visited, or the density of these cells (percentage of cells visited).
  • No, thank you: no output.
  • Full tabular: content of the JOURNAL table in a standard tabular layout.
  • Step + density CSV: at the end of each step, density (percentage) of cells visited; CSV format with *tab* delimiters.
  • Full CSV: content of the JOURNAL table in CSV format with *tab* delimiters.

The application code of the *Extended* simulator

The *Extended LawnMower simulator* is implemented by the script **MAIN-Lawn-Mower-extended**. It calls 7 procedures similar to those of the *Basic* version:

– _AGENT-Init-Parameters-extended
– _AGENT-Create-Tables-extended
– _AGENT-Get-Parameters-extended
– _AGENT-Create-World-extended
– _AGENT-Populate-World-extended
– _AGENT-Generate-Current-State-extended
– _AGENT-Process-History-extended

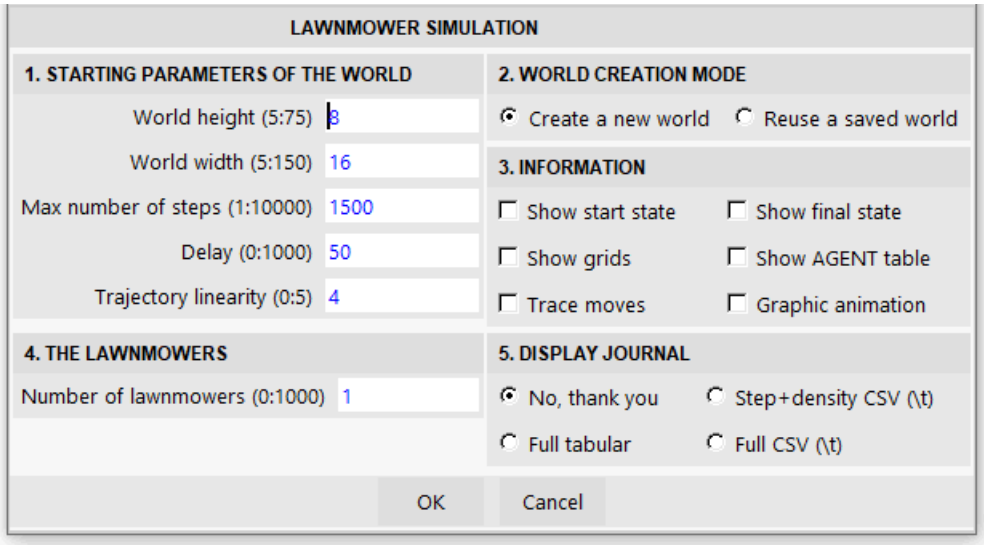


Figure 24.10 - The control panel of the extended *Lawnmower* simulator

Four ancillary procedures: _AGENT-Trajectories, _AGENT-Generate-Grid, _AGENT-Clear-Tables and Display-CELL-extended (called by INFORMATION function Show grid).

A first conclusion

We observe that the agent's behavior is entirely defined in the trigger body via just 4 SQL queries. This first experiment seems to validate the choice of database concepts and technology for implementing agent-based models and systems.

We have developed a first primitive but useful skill for agents: *moving in space*. We will build on this skill in the following experiments.

24.8 A first multi-agent world: *Aggregation*

In this second essay, the world hosts several agents of the same type and allows them to interact, albeit still in a very basic mode. Agents are randomly distributed in the space, then start to move according a predefined trajectory style. When two agents meet in the same cell, they assemble to form a new object, an *aggregate*. Similarly, when an agent moves to a cell containing an aggregate, it joins the latter, augmenting it by one unit. Aggregates are static and stay in their cell until the completion of the simulation. When an agent joins (or forms) an aggregate, it becomes inactive and stop moving.

The simulation ends when each agent has joined an aggregate, which can be considered the *objective* of the agent system.

24.8.1 SQL modeling

A. Space structure

The Content property of a cell indicates the number of agents it contains. Now, a cell is in one of three states depending on its content:

- Content = 0: empty
- Content = 1: contains one moving agent
- Content > 1: contains a static aggregate of size Content.

B. Analysis of the agent behavior

The way an agent reacts when activated is similar to that of the single-agent case studied in Section 24.7. Two differences however in the activation process: the definition of the set of agents to activate and the order in which the agents are activated.

The agents that form an aggregate must be discarded from the activation process. So, this process concerns only the agents that are alone in their cell.

During the execution of a step, all the agents are activated. To this aim, each of them receives a message ('move') instructing it to move. Though the sending of the message to all the agent translates into a single query and therefore appears as a global, set-oriented operation, we know that, internally, the agents will be processed sequentially, one at a time, according to an order which depends on the implementation technique of the table. The consequence is that, in the successive steps, the agents are likely to be activated in *the same order*, which creates a significant bias that can invalidate the evolution simulation. To overcome this problem, we suggest sending the activation messages to the agents in a random order.

24.8.2 SQL code

The extensions to multi-agent worlds of the code developed in Script 24.7 are shown in Script 24.14:

- the agents are activated in a random order (`update ... order by random()`),[21]
- the "`when`" clause of the trigger delimits the category of agents that will respond to the event: those located in a cell containing exactly one agent,
- it is no longer necessary to leave a trace of every cell visited.

```
-- The triggering event
update AGENT set Message = 'move' order by random();
...
-- The reaction
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
     and (select Content from CELL where (X,Y)=(new.X,new.Y)) = 1
begin
    <moving the agent [same as Script 24.7]>
end;
```

Script 24.14 - The trigger that controls the behavior of the agents: only moving agents are concerned

Installing the initial population

The creation of the initial set of agents is done in two steps (Script 24.15). First, the *content* property of a certain number of cells is set to 1, indicating that an agent is present, while the others are set to 0. This number is defined by the *density* of the agent population within the space. This density is specified through the density variable, set from 1 to 100 by the user. The decision to create an agent in a cell is implemented by the following expression, where `random_i(i1,i2)`[22] returns a random integer between `i1` and `i2`:

```
random_i(1,100) <= $density$
```

For instance, if density = 10, this expression is expected to be True for about 10% of the cells.

21. When this study was published, SQLite did not yet implement this form. A workaround would be to create a view cloning the AGENT table with an `order by random()` clause, then to send the message to the rows in that view.
22. The syntax depends on the DBMS. In SQLfast, `random_i` is defined as a UDF wrapping the `random()` SQLite core function.

Then, an AGENT row is inserted for each cell the Content of which is 1.

```
with
   CX(X,Y) as (select 1,1
                 union all
               select X+1,1 from CX where X < $width$),
   CY(X,Y) as (select X,Y from CX
                 union
               select X,Y+1 from CY where Y < $height$)
   insert into CELL(X,Y,Content)
        select X,Y,case when random_i(1,100) <= $density$
                        then 1 else 0
                   end
        from CY;
insert into AGENT(X,Y) select X,Y from CELL where Content = 1;
```

Script 24.15 - Creating the initial state of the agent population

Graphical rendering

The graphical representation of the result of each step must show both agents and aggregates with specific forms and colors, such as those suggested in Figure 24.11 that inform on the content of a cell, and, therefore, on the size of the aggregates (from 2 to 6+).

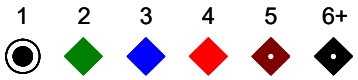


Figure 24.11 - Color codes for the different cell contents

The Figure 24.12 shows the evolution of a world in which 28 agents move according to a *Linear 1* trajectory and eventually generate 9 aggregates.

24.8.3 Experimentation: the *Aggregation* simulator

The simulator is available in two versions, *Basic* and *Extended*.

The control panel of Figure 24.13 can be used to experiment with the *Basic* version of the *Aggregation module* of the Agent application.

The parameters are similar to those of the *Lawnmower module* (Figure 24.9) except for the new parameter Agent density, through which one specifies the percentage of cells that contain an agent at start time.

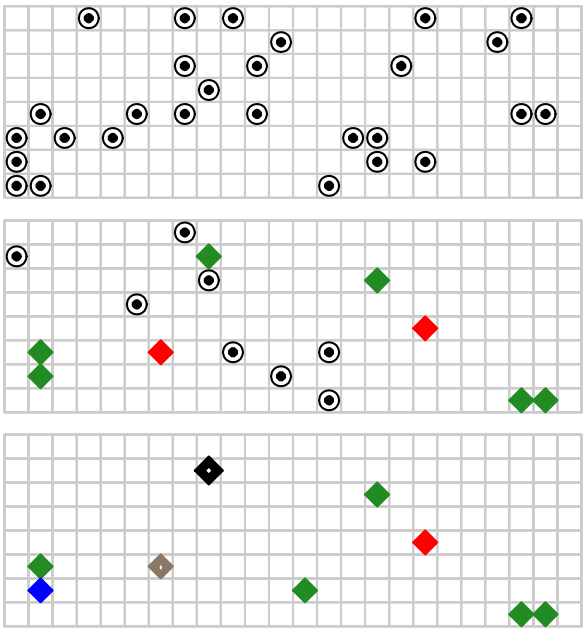


Figure 24.12 - Three representative states of the world: start (top), at step 10 (center) and final (bottom)

The application code of the simulator

The basic *Aggregation* simulator is implemented through the script **MAIN-Aggregation**. It calls the same 7 procedures as the *Lawnmower* simulator that define the agent system:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World
- _AGENT-Populate-World
- _AGENT-Generate-Current-State
- _AGENT-Process-History

The *Extended* simulator

In the same way as we did for the *Lawnmower* simulator, the *Extended* version of the *Aggregation* simulator includes specific functions by which the experimenter can be provided with more control and more detail on the simulations.

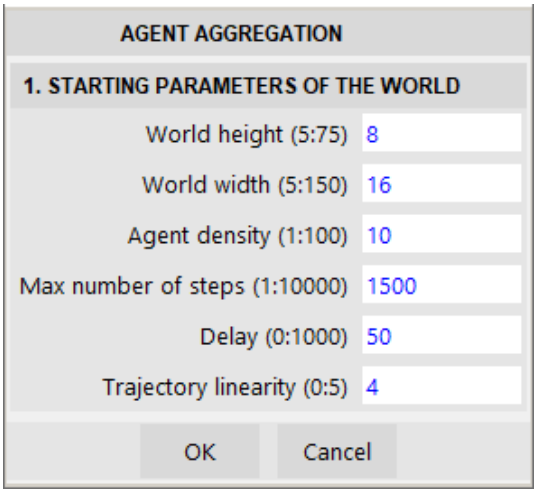


Figure 24.13 - The control panel of the basic *Aggregation* simulator

The synthesis of the state of the world after each step is recorded in the JOURNAL table, structured as follows:

```
create table JOURNAL(StartTime timestamp,
                     Step      integer,
                     Nagent    integer default 0,
                     Agreg2    integer default 0,
                     Agreg3    integer default 0,
                     Agreg4    integer default 0,
                     Agreg5    integer default 0,
                     Agreg6    integer default 0,
                     AgregMax  integer default 0);
```

Each record comprises the timestamp of the start of the simulation, the index of the current step, the number of remaining independent agents and the number of aggregates of 2, 3, 4, 5, 6 and more agents.

The control panel of the *Extended* version comprises three additional elementary boxes similar to those of the *LawnMower* simulator.

Observation

We observe that, once the problem of controlling agent moves has been mastered, extending the *Lawmower* model to agent aggregation just requires adapting the **`when`** clause of the **T_AGENT** trigger.

24.9 Attractors and repellers

In the first two experiments, the agents have not proved to be particularly intelligent. They just blindly follow the rules of their trajectory style, but are not even aware that they are forming aggregates. They never consult their environment to decide on their subsequent actions (with the sole exception of the bouncing rules).

In this section, we consider objects that agents can encounter and interact with, either positively or negatively. Interactions consist, for these new objects, in attracting or repelling agents moving in their vicinity. Agents will therefore try to join nearby *attractors* and to avoid *repellers*. For example, an attractor can be any resource, such as food or medicine, and a repeller a pit or a trap.

For now, we decide that attractors and repellers are not agents but fixed, static objects. In more complex agent models, we could think to assign the roles of attractor and repeller to active agents, such as potential partners or predators.

24.10 Agents and attractors

For the sake of simplicity, we choose to represent attractors by the aggregates we described in section 24.8. As soon as two agents meet in the same cell, they create an aggregate which, from that moment on, will attract and absorb any other agent moving nearby. We will consider two forces of attraction, *close* and *distant*. Near attraction applies between the attractor and an agent in its vicinity, i.e. when their distance is one cell. Distant attraction applies when the agent is close to the attractor's neighborhood, i.e. when their distance is two cells.

24.10.1 SQL modeling of attractors

A. Data structure

No new features needed.

B. Analysis of the agent behavior

The attraction is active in the eight neighboring cells of the vicinity of an attractor. If an agent enters the vicinity of such an object, it is forced, or invited, to join the latter. Considering the way attractors are created, any cell the content of which is greater than 1 comprises an attractor.

If an agent is in the vicinity of one or several attractors, then, conversely, these attractors are comprised in the vicinity of this agent. Therefore, an agent is under the influence of attractor(s) whenever its vicinity includes at least one cell with a content greater than 1. The Figure 24.14 shows a fragment of the world in which an agent just moved from cell (6,6) to cell (7,5), therefore entering in the vicinity of

two attractors, located in (6,4) and (8,5). Once these attractors have been identified, the agent joins one of them, randomly selected.

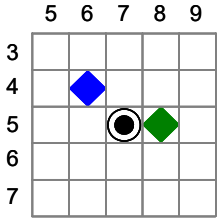


Figure 24.14 - The vicinity of current agent (7,5) includes two close attractors

Distant attractors are illustrated in Figure 24.15. Though the direct vicinity of the current agent (its 8 surrounding cells) does not include any attractor, there are cells in its vicinity that would place the agent next to an attractor and that are worth to move to. Let us imagine that the attractor provides some food, that this food smells particularly good (perceptible at a 2-cell distance) and that the agent is desperately hungry. It's not unreasonable to imagine that this agent tries to get closer to this *distant* attractor in order to be able to profit from its resource in the next step. So, when this agent is asked to move (`update AGENT set Message = 'move'`), its next move will be among the cells marked with a question mark in Figure 24.15 and not to those with an X mark.

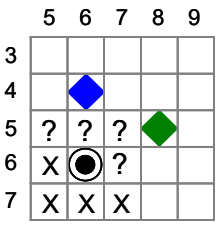


Figure 24.15 - Recommended (**?**) and invalid (**X**) moves of the current agent trying to *get closer* to a distant attractor

24.10.2 SQL code for attractors

A. SQL code for close attractors

When the current agent is activated, it is located in cell (`old.X`,`old.Y`). Its vicinity is computed as follows:

```
select X,Y
from   CELL
where  X between old.X - 1 and old.X + 1
```

```
and    Y between old.Y - 1 and old.Y + 1
and    (X,Y) <> (old.X,old.Y)
```

From the 8 cells of this area, we derive the coordinates of the attractors of the agent:

```
select X,Y
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1
and    Content > 1;
```

Two remarks

We have not kept the specific, but redundant, condition that discards the current cell (`(X,Y) <> (old.X,old.Y)`) since its content is not greater than 1.

We don't need to worry about neighboring cells beyond the space boundary, as they are not represented in the AGENT table.

This query produces a *set* of cell coordinates, for instance {(6,4),(8,5)} in the example of Figure 24.14, from which we must choose the attractor that will absorb the current agent. We could choose the biggest attractor, that with the largest content, but, for now, let us leave this selection to chance!

We convert, through the group_concat function, the set of attractor coordinates into a *string-list* (i.e., `'6,4;8,5'`), on which we can further apply the random function. The query of Script 24.16 translates this conversion, in which the restriction on the content of the cells has been converted into a more elegant *filter* condition[23].

```
select group_concat(X||','||Y ,';')
       filter (where Content > 1) as Att
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1;
```

Script 24.16 - Looking for the close attractors of the agent in cell (old.X,old.Y)

Actually, we are interested in the relative coordinates of the valid cells, that is, their directions. In addition, considering that the group_concat function returns a *null* value when applied to an empty set, we convert this value into an empty list with the coalesce function (Script 24.17).

23. The `filter` clause is associated with an aggregate or window function. It limits the set of rows on which the aggregate function is computed. While the same condition in the `where` clause of the query applies to all the aggregate functions, the filter clause applies to one function only.

```
select coalesce(group_concat((X-old.X)||','||(Y-old.Y) ,';')
               filter (where Content > 1), '') as Att
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1);
```

Script 24.17 - Computing the valid moves to the close attractors of the agent in cell (old.X,old.Y) - Coping with empty sets

Once an attractor has been selected, the agent moves to its cell to be absorbed.

B. SQL code for distant attractors

Let's go back to Figure 24.15. We postulate that the (close) vicinity of the current agent does not comprise an attractor, otherwise, the search for distant attractors would be useless.

We observe that the area of potential distant attractors is a square ring at a 2-cell distance from (`old.X,old.Y`). This square has four sides, N, S, E, W, with the following coordinates:

- **N side**: `old.X` - 2 ≤ X ≤ `old.X` + 2 and Y = `old.Y` - 2
- **S side**: `old.X` - 2 ≤ X ≤ `old.X` + 2 and Y = `old.Y` + 2
- **W side**: X = `old.X` - 2 and `old.Y` - 2 ≤ Y ≤ `old.Y` + 2
- **E side**: X = `old.X` + 2 and `old.Y` - 2 ≤ Y ≤ `old.Y` + 2

More synthetically,

- **horizontal sides**: abs(X - `old.X`) ≤ 2 and abs(Y - `old.Y`) = 2
- **vertical sides**: abs(Y - `old.Y`) ≤ 2 and abs(X - `old.X`) = 2

This translates easily in the following query which, in addition finds the attractors within this area (`Content > 1`),

```
select X,Y
from   CELL
where  (   (abs(X - old.X) <= 2 and abs(Y - old.Y) = 2)
        or (abs(Y - old.Y) <= 2 and abs(X - old.X) = 2))
and    Content > 1;
```

Now, we describe the *free cells* of the vicinity of these attractors. The query of Script 24.18 selects the empty cells in the vicinity of all the distant attractors of the current agent. The **Att** alias denotes the set of distinct attractors. The **Vatt** alias, joined to **Att**, describes the vicinity of each of these attractors.

```
select distinct Vatt.X,Vatt.Y
from  (select X,Y
       from   CELL
       where  (   (abs(X - old.X) <= 2 and abs(Y - old.Y) = 2)
               or (abs(Y - old.Y) <= 2 and abs(X - old.X) = 2)
              )
       and    Content > 1) Att,
       CELL Vatt
where  Vatt.X between Att.X - 1 and Att.X + 1
and    Vatt.Y between Att.Y - 1 and Att.Y + 1
and    Vatt.Content = 0;
```

Script 24.18 - Selecting the moves to the free cells in the vicinity of the distant attractors of the agent in cell (old.X,old.Y)

Finally, we compute the valid moves of the current agent to the vicinity of its distant attractors. The destination cells are defined as the set of common empty cells in the vicinity of the attractors and the vicinity of the agent. Expressing the coordinates of these cells (actually their relative coordinates ($\Delta x, \Delta y$)) in a single SQL query deserves a bit of explanation (Script 24.19). The main query computes the intersection of two sets of free cells expressed as two string-lists, **AttVic**, the merged vicinity of all the distant attractors, and **CurVic**, the empty vicinity of the current agent:

```
intersect(AttVic,CurVic)
```

The set **AttVic** has been computed in Script 24.18 while the set **CurVic** derives from the query of Script 24.17.[24]

C. Extension of the T_AGENT trigger

The overall structure of the trigger that specifies the behavior of the agents, and, in particular the `when` filter, are similar to those of the aggregation scenario (Script 24.20).

The content of the body is extended to apply the new rules governing the attraction phenomenon. The reasoning leads to a hierarchy of rules: first, the agent looks for close attractors; if none, it tries to find distant attractors. Finally, if no attractor has been found, it applies the rules of its trajectory style.

The updated version of the query that computes the next direction ($\Delta x,\Delta y$) is shown in Script 24.22.

24. One of the two conditions "`Vatt.Content = 0`" and "`Content = 0`" can be dropped. However we have kept both to make the query more readable.*

```
select intersect(AttVic,CurVic) as ValMov
from (select
            coalesce(group_concat((Vatt.X-old.X)||','
                                  ||(Vatt.Y-old.Y),';'),
                     '') as AttVic
      from  (select X,Y
             from   CELL
             where  (   (abs(X-old.X) <= 2 and abs(Y-old.Y) = 2)
                     or (abs(Y-old.Y) <= 2 and abs(X-old.X) = 2)
                    )
             and    Content > 1) Att,
             CELL                Vatt
      where Vatt.X between Att.X - 1 and Att.X + 1
      and   Vatt.Y between Att.Y - 1 and Att.Y + 1
      and   Vatt.Content = 0
     ),
     (select
            coalesce(group_concat((X-old.X)||','
                                  ||(Y-old.Y) ,';'),
                     '') as CurVic
      from  CELL
      where X between old.X - 1 and old.X + 1
      and   Y between old.Y - 1 and old.Y + 1
      and   Content = 0
     );
```

Script 24.19 - Computing the valid moves to the vicinity of distant attractors

```
create trigger T_AGENT
after update of Message on AGENT
for  each row
when new.Message = 'move'
     and (select Content from CELL where (X,Y)=(old.X,old.Y)) = 1
begin
    <moving the agent in presence of attractors>
end;
```

Script 24.20 - The trigger that controls the behavior of the agents when the space comprises attractors

```
-- 1. The agent looks for close attractors            --
    set V2 = <Script 24.17>
-- 2. The agent looks for for distant attractors      --
    set V3 = <Script 24.19>
-- 3. The agent chooses its closest attractor         --
    set V1 = case when :V3 = '' then :V2 else :V3 end;
-- 4. The agent quits its current position            --
update CELL set Content = Content - 1
where  (X,Y) = (old.X,old.Y);
-- 5. The agent computes its next direction           --
select item(Next,1,','),item(Next,2,',') into DeltaX,DeltaY
from   (<compute the (Δx,Δy) direction [Script 24.22]>);
-- 7. The agent computes its next position            --
update AGENT
set    X = X + :DeltaX, Y = Y + :DeltaY,
       OldX = :DeltaX, OldY = :DeltaY,
where  AgentID = old.AgentID;
-- 8. The agent moves to its next position            --
update CELL set Content = Content + 1
where  (X,Y) = (select X,Y from AGENT
                where  AgentID = old.AgentID);
```

Script 24.21 - Algorithm driving the behavior of the agents in presence of close and distant attractors

Just by curiosity

We have compared the speed (number of steps) of the three aggregation techniques to aggregate all the 128 agents of a representative world comprising 20x60 cells:

– pure aggregation, no attractors: 125 steps

– close attractors only: 50 steps

– close and distant attractors: 30 steps.

Each result is the average of the 17 central values provided by 21 simulation runs (the two lower and higher values being discarded).

24.10.3 Experimentation: the *Attractors-Repellers* simulator

The control panel of Figure 24.18 can be used to experiment with the *Attractors module* of the Agent application.

The parameters are similar to those of the *Aggregation module* (Figure 24.13). The simulator offers three modes: pure *aggregation*, *close attractors* and *distant attractors*.

```
select
   case
   -- an attractor has been found
      when :V1 <> '' then random(:V1)
   -- bouncing moves
      when X in (1,$width$) or Y in (1,$width$)
      then <compute the bouncing step [Script 24.11]>
   -- starting move
      when (OldX,OldY) = (0,0)   then random($V$)
   -- internal moves
      when (OldX,OldY) = (0,1)   then random($A$)
      when (OldX,OldY) = (0,-1)  then random($B$)
      when (OldX,OldY) = (1,0)   then random($C$)
      when (OldX,OldY) = (1,1)   then random($D$)
      when (OldX,OldY) = (1,-1)  then random($E$)
      when (OldX,OldY) = (-1,0)  then random($F$)
      when (OldX,OldY) = (-1,1)  then random($G$)
      when (OldX,OldY) = (-1,-1) then random($H$)
   end as Next
from  AGENT
where AgentID = old.AgentID;
```

Script 24.22 - Extended version of Script 24.10 to cope with attractors

24.11 Agents and repellers

Similarly to attractors, we consider two forces of repulsion, *close* and *distant*. Near repulsion applies between a repeller and an agent in its vicinity, i.e. when their distance is one cell. Distant repulsion applies when the agent is close to the attractor's neighborhood, i.e. when they are at a 2-cell distance.

24.11.1 SQL modeling of repellers

A. Data structure

Unlike attractors, repellers cannot be derived from the meeting of agents. They must be installed in the initial state of the space as specific objects. In the graphical representation, they will appear as black diamond.

B. Analysis of the agent behavior

Avoiding *close repellers* consists, for the current agent, in ignoring any move to the cells of its repellers. This means that the valid move of the agent are to any cell of its vicinity, except to the cells of its repellers (Figure 24.16).

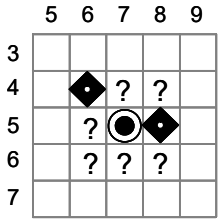


Figure 24.16 - The recommended moves (noted **?**) of current agent (7,5) trying to avoid its close repellers

The goal of trying to escape a *distant repeller* can also mean avoiding any future move that will place the agent in the vicinity of this repeller.

The world extract of Figure 24.17 shows the cells to avoid (marked with an X) and those that the agent must move to (marked with a ?).

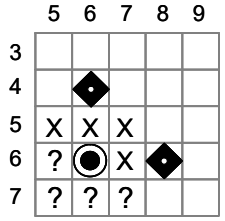


Figure 24.17 - Valid moves of the agent in cell (6,6) to get away from the *proximity* of its distant repellers

Interestingly, this figure is the exact opposite of Figure 24.15, in which the X and ? cells have been swapped. The valid destination cells are those of the vicinity of the current agent *except* those of the vicinity of the distant repellers.

24.11.2 SQL code for repellers

In this implementation, two agents are allowed to share the same cell while keeping their autonomy, without merging into an aggregate.

A. SQL code for close repellers

We decide to designate the cells of the repellers with the value Content = 9. Identifying the moves to safe cells in the vicinity of an agent is straightforward (Script

24.23). From the safe cells of Figure 24.16 (`'7,4;8,4;6,5;7,5;6,6;7,6;8,6'`), this query returns the six safe moves:

```
'0,-1;1,-1;-1,0;0,0;-1,1;0,1;1,1'
```

```
select coalesce(group_concat((X-old.X)||','||(Y-old.Y),';')
                filter (where Content <> 9), ''))
from   CELL
where  X between old.X - 1 and old.X + 1
and    Y between old.Y - 1 and old.Y + 1);
```

Script 24.23 - The valid moves of the agent in cell (`old.X`,`old.Y`) that *avoid* its close repellers

B. SQL code for distant repellers

The query derives from the algorithm for the distant attractors shown in Script 24.19, in which the set operator `intersect` is replaced by `except` (Script 24.24).

```
select except(CurVic,RepVic)
from (select coalesce(group_concat((Vrep.X-old.X)||','
                                   ||(Vrep.Y-old.),';'),
                      '') as RepVic
      from (select X,Y
            from   CELL
            where  (   (abs(X-old.X) <= 2 and abs(Y-old.Y) = 2)
                    or (abs(Y-old.Y) <= 2 and abs(X-old.X) = 2)
                    )
            and    Content = 9) Rep,
            CELL                Vrep
      where Vrep.X between Rep.X - 1 and Rep.X + 1
      and   Vrep.Y between Rep.Y - 1 and Rep.Y + 1
     ),
     (select coalesce(group_concat((X-old.X)||','
                                   ||(Y-old.Y) ,';'),
                      '') as CurVic
      from  CELL
      where X between old.X - 1 and old.X + 1
      and   Y between old.Y - 1 and old.Y + 1
     );
```

Script 24.24 - Computing the valid moves to avoid the vicinity of distant repellers

Applied to the world fragment of Figure 24.17, the query computes the set of valid cells:

```
'5,6;5,7;6,6;6,7;7,7'
```

converted into the relative coordinates of the recommended moves:

```
'-1,0;-1,1;0,0;0,1;1,1'
```

24.11.3 Experimentation: the *Attractors-Repellers* simulator

The *Attractors* simulator has been extended to include close and distant repellers. The new control panel is shown in Figure 24.18.

The Figure 24.19 shows the state of a 8x24 world comprising 6 distant repellers and 12 agents. To make the agent/repeller interactions more visible, the agents leave a trace in the cells they have visited, as in the lawmower simulation. The result clearly shows that no agent attempted to cross the distant frontier of any repeller.[25]*

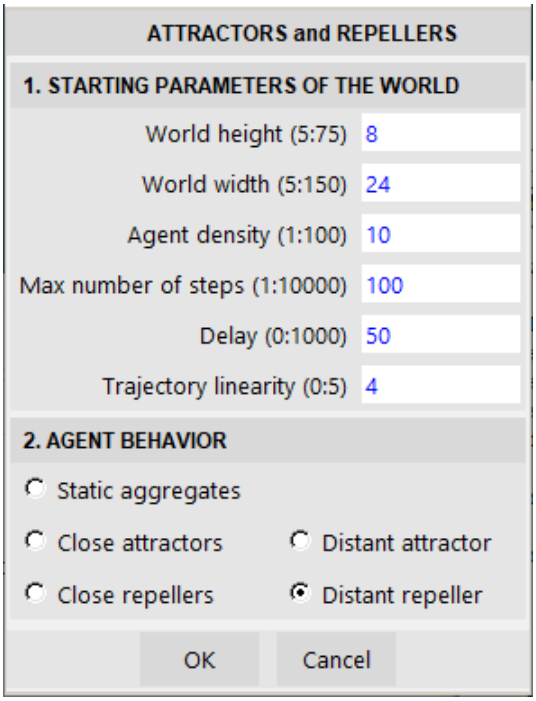


Figure 24.18 - Control panel of the Basic Attractor-Repellers simulator

25. The so-called *final state* shows only 11 agents. This means that two agents are currently in the same cell.

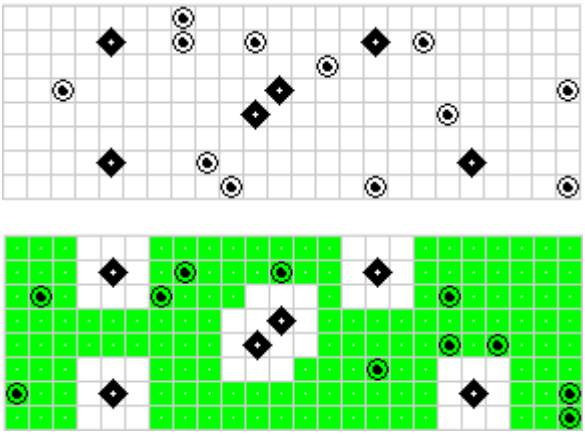


Figure 24.19 - Starting and final states of a 8x24 world comprising 6 distant repellers

The application code

The *Basic Attractors-Repellers* simulator is implemented through the script MAIN-Attractors-Repellers.sql. To make the code more readable, the *world creation* and *world population* procedures are split in two parts, one devoted to attractors and the other to repellers. The main script calls 9 procedures:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World-attractors
- _AGENT-Create-World-repellers
- _AGENT-Populate-World-attractors
- _AGENT-Populate-World-repellers
- _AGENT-Generate-Current-State
- _AGENT-Process-History

24.12 Agent life cycle

The attraction and repulsion reactions developed in the previous experiment, despite their primitive nature, nonetheless bring our agents closer to the behavior of living beings, be they animals or even humans. Let's now try to enrich our agent model by giving them a simple but reasonably realistic life cycle.

24.12.1 Towards autonomous agents

To mimic living beings, we allow agents to be born, to grow old, to generate new agents and to die, generally in this order[26]. Let us briefly analyze these events.

– *An agent is born*. Unlike the previous models, the population is not fully defined at the start of the simulation. Indeed, while the first members are created to start the evolution process[27], new agents will appear thanks to the generation procedure described below.

– *An agent ages*. Time will bring about a change in the agent's capabilities: depending on the time spent in its society, an agent acquires certain capabilities (such as food autonomy or the generation of new agents) and loses others, including life. Age measurement is based on the concept of an *age period* (days, months or years, depending on the species to be modeled), a unit defined by a number of clock ticks[28].

– *Agents generate new agents*. Given the infinite variety of reproduction modes in the animal world, we chose one of the most classical: two agents, of different gender, meet in the same place, and, if certain conditions are met, generate a new agent. This procedure is open to many possible refinements. Can one gender be attracted by the other (close or distant attraction), or do they meet by chance? Can one gender sometimes repel the other's attempts? Does attraction depend on the age of the agents? Can *parents* (so to speak) generate more than one new agent at a time? Does each parent have to respect a latency period between two successive generations? Is there a delay between the meeting and the birth of the new agent (a sort of pregnancy period)? Does each meeting automatically lead to the birth of a new agent? Can certain environmental conditions influence the generation process, such as overpopulation or food abundance?

– Finally *agents die*. After a certain number of age periods, agents are expected to die. Let us submit the description of this event to two questions: (1) does it depend on environmental conditions (food scarcity, disease, attacks from predators), (2) does this age limit represent an exact value, or an average and, in this interpretation, what is the distribution function?

26. In short, *Hatching, matching and dispatching*
27. Starting with the first amoeba would really take up too much time
28. We remember that each clock tick triggers a simulation *step*.

From this general analysis, we draw the subset of characteristics we intend to implement.

- At its birth, an agent has an age of 0. Each additional age period is acquired when the agent has received a definite number of clock ticks.
- To be able to reproduce, an agent must have acquired a definite level of maturity, defined as a minimum age. There is no maximum age limit. In addition, between two successive generations, an agent must wait a latency period defined as a definite number of clock ticks. An agent that satisfies the conditions of maturity and latency is said *fertile*. Otherwise it is qualified *sterile*.
- When a fertile agent finds a fertile partner of a different gender, it asks this partner to generate a new agent. This generation occurs with a certain probability.
- An agent dies when it has just completed its last age period. This limit is common to all the agents.

24.12.2 SQL modeling

A. World structure

To control the evolution of living agents, we need some reference parameters valid for all the simulations.

- ticksPerAge: the number of clock ticks of an age period
- maxAge: the common age of death of the agents
- maturity: the minimum age for an agent to be allowed to generate a new agent
- matchingProba: probability that two partners that meet actually generate a new agent
- latency: minimum period, measured in number of clock ticks, between two generation events from the same agent[29].

These global parameters are implemented as static variables.

B. Agent structure

So far, the state of the agents were limited to their position in the world space (X,Y) and their trajectory (OldX,OldY). Now we must give them new properties that represent their evolution. Hence the four new column of table AGENT:

- Age: its age, the number of completed age periods since its birth

29. This period is designed to model various real or technical phenomena, such as the feeding period of a newborn, pregnancy, preventing partners to continue generating immediately after a generation, therefore avoiding uncontrolled demographic expansion.

- Ticks: the current number of clock ticks since the last age increase
- Gender: its gender (a constant); conventionally denoted by meaningless characters 'F' or 'M'.
- Latency: the current number of clock ticks since the last generation.

Their values must satisfy these constraints:

- $0 \leq \text{Age} < \text{maxAge}$
- $0 \leq \text{Ticks} < \text{ticksPerAge}$

An agent is fertile iif:

- $\text{Age} \geq \text{maturity}$ and $\text{Latency} \geq \text{latency}$

A couple of agents (a1,a2) are allowed to generate a new agent iif:

- a2.X between a1.X - 1 and a1.X + 1
- a2.Y between a1.Y - 1 and a1.Y + 1
- $\text{a1.Age} \geq \text{maturity}$ and $\text{a2.Age} \geq \text{maturity}$
- $\text{a1.Latency} \geq \text{latency}$ and $\text{a2.Latency} \geq \text{latency}$
- $\text{a1.Gender} \neq \text{a2.Gender}$

C. Analysis of the agent behavior

The agents are free to move according to the chosen trajectory style. A cell can host several agents; they do not aggregate and keep their independence.

At each step, before moving, a fertile agent searches its close vicinity for potential fertile partners of different gender. It asks one of these partners, randomly selected, to generate a new agent by sending it the message 'generate'. The partner will generate, in its own cell, one new agent with probability matchingProba.

The new agent is assigned the following properties:

- (X,Y) = position of the generating partner
- (OldX,OldY) = (0,0)
- Ticks = 0
- Age = 0
- Gender = random('M,F')
- Latency = 0

Column Latency of the current agent and of the partner is reset to 0, so that they both become sterile. After the generation event, the current agent moves one position, as usual.

24.12.3 SQL code

The code of Script 24.25 defines the new structure of the AGENT table and sets some realistic values of the global parameters.

```
create table AGENT(AgentID integer primary key,
                   X       integer,
                   Y       integer,
                   OldX    integer default 0,
                   OldY    integer default 0,
                   Message varchar(6) default '',
                   Ticks   integer default 0,
                   Age     integer default 0,
                   Gender  char(1),
                   Latency integer default 0);

set maxAge = 30;
set ticksPerAge = 3
set matchingProba = 100;
set maturity = 6;
set latency = 3;
```

Script 24.25 - Structure of *living* agents

The initial population is created by Script 24.26 (the content of the CELL rows has been initialized according to the desired density). These initial values are chosen to ensure a certain diversity among the agent population.

```
insert into AGENT(X,Y,Ticks,Age,Gender,Latency)
     select X,Y,
            random_i(0,$ticksPerAge$ - 1),
            random_i(int(0.30*$maxAge$),$maxAge$-2),
            iif(random_i(1,100) <= 50,'M','F'),
            random_i(1,$latency$ - 2)
     from CELL where Content = 1;
```

Script 24.26 - Initial state of the agents

Architecture of the main trigger

The T_AGENT_MOVE[30] trigger must be activated for each AGENT row, so, the `when` clause is made up of the sole condition on the nature of the message. The body comprises six tasks, the order of which is important. The first task, *Managing agent age*, consists in updating the time-dependent properties of the agent stored in columns Age, Ticks and Latency. This task can change the status of a agent, which,

30. The new name of the T_AGENT trigger.

from sterile, becomes fertile. In this case, this agent will be allowed to look for a partner (*The agent looks for a partner*) and, in case of success, to ask this partner to generate a new agent (*The agent generates a new agent*). The next tasks move the agent as in the preceding experiments.

The development of each of the first three tasks is examined below.

```
create trigger T_AGENT_MOVE
after update of Message on AGENT
for  each row
when new.Message = 'move'
begin
    <1. Managing agent age>
    <2. The agent looks for a partner>
    <3. The agent generates a new agent>
    <4. The agent computes its next direction>
    <5. The agent computes its next position>
    <6. The agent moves to its new position>
end;
```

Script 24.27 - Architecture of the main trigger

1. Managing agent age

Managing the passing of time of an agent is straightforward (Script 24.28).

```
update AGENT set   Ticks = Ticks + 1,                          [1]
                   Latency = Latency + 1
           where AgentID = old.AgentID;
update AGENT set   Age = Age + 1,                              [2]
                   Ticks = 0
           where AgentID = old.AgentID
           and   Ticks = $ticksPerAge$;
delete from AGENT                                              [3]
           where AgentID = old.AgentID
           and   Age = $maxAge$;
```

Script 24.28 - Managing the age of agents

The values old.AgentID, old.X, old.Y contain the properties of the current agent at the time of the initial event (`update Message`), even when this agent just was deleted.

– The number of ticks received from the clock is incremented (query [1]).

– Once the maximum of ticks is received, the age is incremented and the number of ticks is reset to 0 (query [2]).

– At the death of an agent, the current row is deleted from the table AGENT (query [3])
– The CELL row of a dead agent will be updated later, in task 6.

2. The agent looks for a partner

The query of Script 24.29 stores in variable partner the value of AgentID of a valid partner (randomly selected in PartList) for the current agent [1].

```
set partner = '';                                                    [0]
select random(PartList) into :partner                                [1]
from  (select coalesce(group_concat(AgentID,';')                     [2]
                       filter (where Gender  <> new.Gender           [3]
                                and   Age >= $maturity$              [4]
                                and   Latency >= $latency$)          [5]
                       '') as PartList                               [6]
       from   AGENT
       where  X between old.X - 1 and old.X + 1                      [7]
       and    Y between old.Y - 1 and old.Y + 1                      [8]
       )
where  new.Age >= $maturity$                                         [9]
and    new.Latency >= $latency$                                     [10]
and    exists (select * from AGENT
               where  AgentID = old.AgentID);                        [11]
```

Script 24.29 - Searching the vicinity of the current agent for a partner

The quest for a partner is allowed if the current agent has not been deleted [11] and is fertile [9,10].

The subquery extracts a list of valid candidate partners that it returns in PartList [2-8]. It first explores the vicinity of the current agent [7,8]. It concatenates the Id's of those agents [2] that are of the opposite gender [3] and that are fertile [4,5]. If no valid partner has been found, the *null* value is converted into an empty string [2,6].

The existence of a valid partner is indicated by a non-empty value of variable partner.

If the current agent is not allowed to search for a partner, the query fails and the value of variable partner is an empty string [0].

3. The agent generates a new agent

The basic laws of agent modeling implicitly include the principle of *autonomy* according to which an agent is not allowed to change the state of another agent without its consent. Therefore, either the current agent decides to create a new agent,

or it asks its partner to create one of its own. We have decided to adopt the second behavior[31] and to code it as a new trigger.

To ask a definite service to an agent, we send it a message explaining the nature of this service. This is what the current agent does through the query of Script 24.30. The structure of the generation trigger will be described later in this section.

We note that this query fails (that is, it performs no action) if the current agent has been deleted.

```
update AGENT
set    Message = 'generate'
where  AgentID = :partner;
```

Script 24.30 - Asking the partner to generate a new agent

Then, the current agent changes its status from fertile to sterile by resetting its latency period (Script 24.31)[32]. Here too, this query is a *no-op* if the current agent has been deleted.

```
update AGENT
set    Latency = 0
where  AgentID = old.AgentID
and    :partner <> '';
```

Script 24.31 - The current agent become sterile.

4. The agent computes its next direction

This query fails if the current agent has been deleted.

5. The agent computes its next position

This query fails if the current agent has been deleted.

6. The agent moves to its new position

1. The agent quits its current position (also valid for the deleted agent)
2. Then moves to its next position (if the agent still exists)

31. We could chose a more realistic idea, to let the agent of a certain gender generate the new agent. This is left as an exercise.
32. This behavior is not completely correct is the value of the matchingProba variable is not 100. If the partner doesn't satisfy the demand of the current agent, there is no need to *sterilize* it. This also is left as an exercise.

Architecture of the generation trigger

Now, let us concentrate on the new trigger that captures the generation event (Script 24.32). Its `when` clause controls the message received. It also expresses that the demand of generation can be accepted or rejected according to the probability specified in variable matchingProba. Each of the three tasks of the body can be expressed as a simple query (Script 24.33).

```
create trigger T_AGENT_GENERATE
after update of Message on AGENT
for  each row
when new.Message = 'generate'
     and random_i(1,100) <= $matchingProba$
begin
    <1. The agent creates a new agent>
    <2. The agent adds a unit in its own cell>
    <3. The agent becomes sterile>
end;
```

Script 24.32 - Architecture of the generation trigger

1

```
insert into AGENT(X,Y,Gender) values(old.X,old.Y,random('M;F'));
update CELL set Content = Content + 1
where (X,Y) = (old.X,old.Y);
update AGENT set Latency = 0 where AgentID = old.AgentID;
```

Script 24.33 - Creating a new agent and updating the parent agent

24.12.4 Experimentation: the *LifeCycle* simulator

The first section of the control panel of the *Basic Agent Life cycle* simulator is similar to that of the previous experiments. It also includes a second section specifying the life cycle parameters of the agents (Figure 24.20).

The *Extended* simulator comprises the same additional sections as the preceding simulators. It also includes a JOURNAL table that records at each step, the number of agents, the (cumulative) number of births and the (cumulative) number of deaths.

The graphical representation of the agents in their space needs some additional properties to better show the status of each agent. The new code is presented in Figure 24.21.

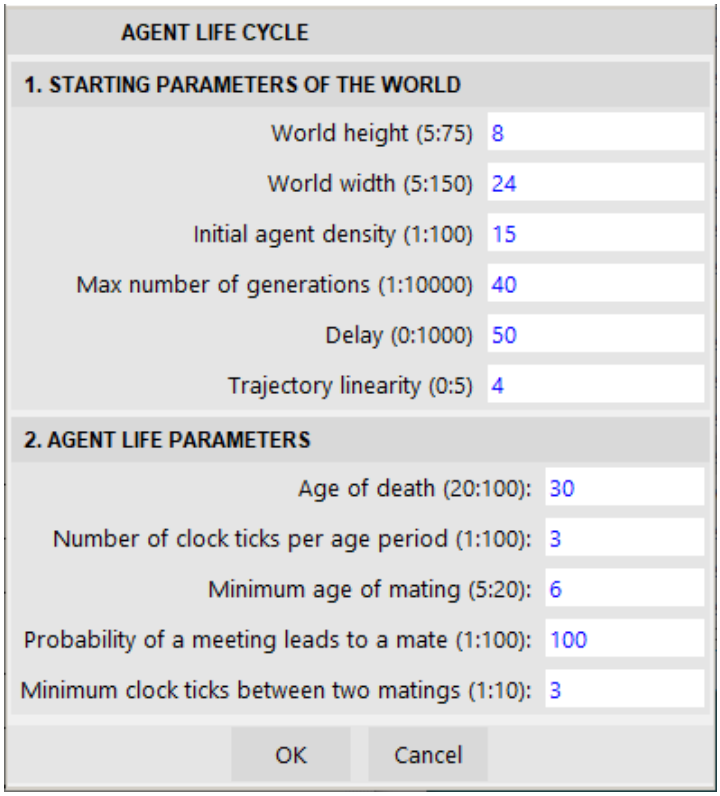


Figure 24.20 - Control panel of the Basic Agent Life Cycle simulator

A simulation has been run with the sample parameters shown in the control panel screen shot. The initial population comprised 25 (sterile) agents including 14 of gender 'F'. The content of the AGENT table (Figure 24.22) describes the state of the 52 agents at the completion of the 24th step. This state results from 35 births and 8 deaths among the initial population.

The graphical representation makes more explicit the spatial configuration of the agent population (Figure 24.23). It shows that,

- 29 cells contain one sterile agent (black circle)
- 6 cells contain one fertile agent of gender F (red circle)
- 6 cells contain one fertile agent of gender M (blue circle)
- 2 cells contain several sterile agents (green diamond)
- 1 cell contains fertile agent(s) of gender F but none of gender M (red diamond)
- 1 cell contains fertile agent(s) of gender M but none of gender F (blue diamond)
- 1 cell contains both fertile agent(s) of gender F and fertile agent(s) of gender M (magenta diamond)



Figure 24.21 - Color code of the content of cells

AgentID	Y	X	Ticks	Age	Gender	Latency
1	6	22	0	25	M	0
2	6	3	0	28	F	2
4	4	3	2	20	M	2
5	7	6	2	21	F	8
6	1	7	0	27	F	14
8	7	10	0	26	M	2
10	8	15	1	26	F	0
12	4	3	0	25	F	10
13	5	17	0	25	F	6
14	2	16	2	29	M	4
17	5	8	2	24	F	18
18	4	7	1	25	F	2
20	1	1	2	18	M	9
21	3	8	0	29	M	1
23	7	19	0	20	F	2
24	3	11	2	29	F	25
25	8	20	0	20	F	0
26	7	15	0	7	M	2
27	2	10	0	7	F	1
28	6	15	0	7	M	1
29	4	9	0	7	F	2
30	1	24	0	7	M	21
31	2	7	0	7	F	21
32	5	11	2	6	M	20
33	7	2	1	6	M	19
34	2	16	0	6	F	18
35	3	7	0	6	M	18
36	6	22	0	6	M	18
37	7	23	0	6	M	18
38	4	22	2	5	F	17
39	5	13	0	5	F	15
40	2	5	0	5	M	15
41	1	20	2	4	F	14
42	4	21	2	4	M	14
43	7	24	1	4	F	13

44	1	13	0	4	M	12
45	7	13	1	3	M	10
46	1	18	0	3	F	9
47	4	4	0	3	F	9
48	8	10	2	2	M	8
49	7	8	2	2	F	8
50	2	19	0	2	F	6
51	6	17	0	2	F	6
52	2	8	2	1	M	5
53	3	17	1	1	M	4
54	7	4	2	0	M	2
55	4	7	2	0	M	2
56	6	19	2	0	F	2
57	8	8	2	0	M	2
58	4	9	1	0	M	1
59	7	16	0	0	F	0
60	6	22	0	0	M	0

Figure 24.22 - State of the AGENT table after step 24

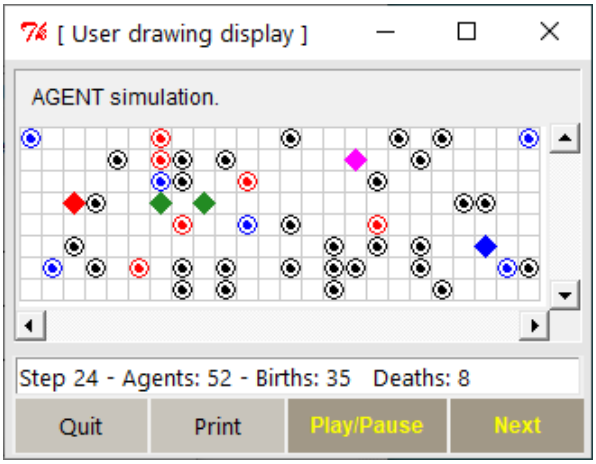


Figure 24.23 - State of the space after step 24

The application code

The *Basic Life cycle* simulator is implemented through the MAIN-Lifecycle.sql script The main script calls the same 7 procedures as the preceding simulators:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World
- _AGENT-Populate-World
- _AGENT-Generate-Current-State
- _AGENT-Process-History

24.13 Agent life cycle with birth control

Running the *Life cycle* model built in the previous section shows that maintaining a stable population can be difficult, as the evolution is highly sensitive to values of the parameters. Let us analyze its behavior and try to better control it.

24.13.1 The LifeCycle model is unstable

Most of the time, the population grows continuously until it completely fills the world, or, on the contrary, decreases until extinction, leaving an empty world. This what the curves of Figure 24.24 show.

Let Ncell be the number of cells and Nag the number of agents. Ncell is a constant of the simulation while Nag is time-dependent (i.e., step-dependent). Let also density denote the ratio 100*Nag/Ncell. Since a cell can host several agents, the density has no upper limit and can get far greater than 100.

Each curve plots the evolution of density during a 200-step simulation for a definite value of matchingProba, from 100 (each meeting generates a new agent) to 20 (only 20% of the meetings generate an agent). The five simulations have been run with these parameter settings:

```
height,width = 8,16        ticksPerAge    = 3
density      = 10          maxAge         = 35
maxStep      = 200         maturity       = 10
trajectory   = 4           latency        = 6
                           matchingProba = 100,80,60,40,20
```

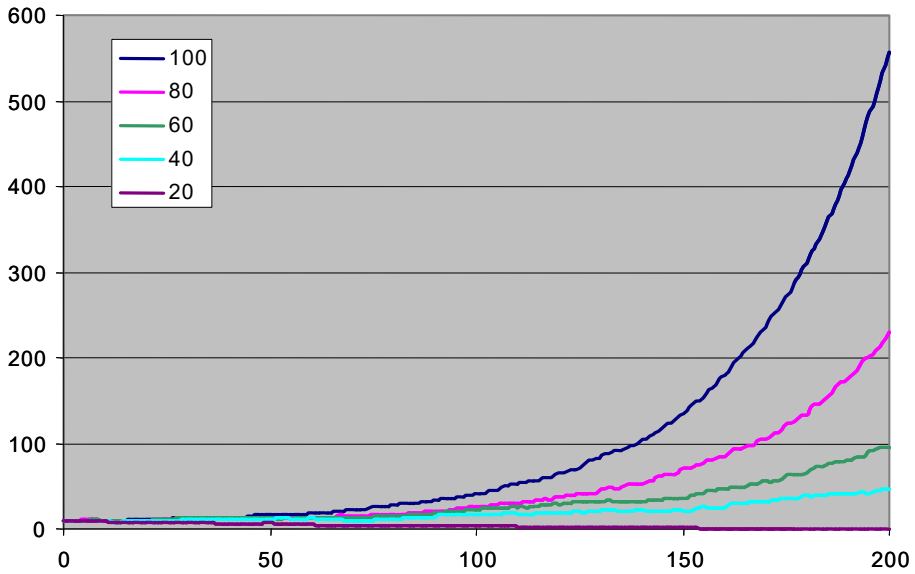


Figure 24.24 - Unstable growth of the agent population

For the largest values of matchingProba (from 100 down to 40) the curves exhibit an exponential growth, while the smallest values (e.g., 20) lead to the extinction of the agent population. Choosing a value that gives a stable evolution on the long run is quite difficult, if not impossible. All the more so since experimenting with the other behavior parameters, namely ticksPerAge, maxAge, maturity and latency lead to the same conclusion: ***this model is intrinsically unstable***.

Given this property, the simulator stops when the number of agents drops to 0 or exceeds twice the number of cells.

24.13.2 Adding birth control to the LifeCycle model

Modeling real systems of this kind requires additional regulation rules that limit the increase and the decrease of the population. Introducing in the model a resource that is naturally limited, such as food, can be a good example. If the world can supply a quantity of Q_f food units per step and if each agent needs Q_a units per step, the world cannot feed more than Q_f/Q_a agents in each step[33].

Such a rule can result in the dynamic adaptation of some of the global parameters of the model. For example, the matchingProba probability, which was a constant in the previous experiments, can be computed in each step: reduced when the population size becomes too large, and increased when the population shrinks to a dangerous level. Similarly, the global parameters latency, maturity and maxAge can also be dynamically adjusted to regulate the population. By fine tuning these parameters, we can develop a better model, closer to the way the target system behaves. For instance, in a close world:

too many agents ⟶ less food ⟶ higher death toll ⟶ fewer agents
too few agents ⟶ more food ⟶ enthusiasm to reproduce ⟶ more new agents

These arguments are valid for models with only one type of agent. The introduction of another competing type will change the game, particularly in the case of a *predator-prey* scheme:

too many predators ⟶ preys decrease ⟶ predators decrease ⟶ fewer predator
too few predators ⟶ preys increase ⟶ predators reproduce ⟶ predators increase

Working on the expression of the matchingProba parameter, as suggested above, seems a good idea.

Let us adopt a new constant parameter, maxDensity, that specifies the maximum density that the agent population can never exceed. At any step of the simulation, the generation process, implemented by the T_AGENT_GENERATE trigger, must ensure that this condition always be satisfied:

$$0 < 100*Nag/Ncell \leq maxDensity$$

33. Simplified hypothesis of the reality: an agent can make do with just under Q_a food units for a limited time before dying.

Let us try to convert the matchingProba probability into a dynamic parameter. Instead of assigning it a constant when the world is built (see Script 24.25),

```
set matchingProba = 100;
```

we *compute* its value at the beginning of each step:

```
select <f(Nag)> into matchingProba from AGENT;
```

Let us consider the following formula

$$matchingProba = max(100 \times (1 - 100/maxDensity \times Nag/Ncell), 0)$$

Let us consider that the space is made up of Ncell = 8*16 = 128 cells and the maximum density is maxDensity = 60%. The population size cannot exceed Nag = 0.6 x 128 = 76.8 ≈ 77 agents. As the number of agents tends to that limit, the probability of two fertile agents generating a new agent tends to zero. On the contrary, when this number decreases, this probability augments.

The table of Figure 24.25 gives the values of matchingProba for increasing values of Nag. The SQL expression of matchingProba is shown in Script 24.34.

Nag	matchingProba
2	97
10	87
20	74
30	61
40	48
50	35
60	22
70	9
76	1
77	0

Figure 24.25 - Values of matchingProba as a function of Nag

```
select max(100*(1 - (100/$maxDensity$) * (count(*)/$Ncell$)),0)
       into matchingProba
from   AGENT;
```

Script 24.34 - SQL expression of matchingProba

The T_AGENT_GENERATE trigger described in Script 24.32 is slightly modified, matchingProba being now a dynamic variable. Its new version is shown in Script 24.34.

```
create trigger T_AGENT_GENERATE
after update of Message on AGENT
for  each row
when new.Message = 'generate'
     and random_i(1,100) <= :matchingProba
begin
    <1. The agent creates a new agent>
    <2. The agent adds a unit in its own cell>
    <3. The agent becomes sterile>
end;
```

Script 24.35 - New version of the T_AGENT_GENERATE trigger

24.13.3 Experimentation: the Agent *LifeCycle with birth control* simulator

The control panel of the *LifeCycle* simulator of the previous experiment has been modified to add the absolute maximum density (Figure 24.26).

The figure 24.27 shows the result of five simulations for different values of maxDensity. They have been run with these parameter settings:

```
height,width = 8,16       ticksPerAge = 3
density      = 10         maxAge      = 35
maxStep      = 400        maturity    = 10
trajectory   = 4          latency     = 6
                          maxDensity  = 100,80,60,40,20
```

Each curve plots the evolution of density during a 400-step simulation for a definite value of maxDensity, from 100 to 20.

This model protects the agents from overpopulation. However, it does not prevent them from extinction for certain combinations of evolution parameter values. As in the real world, extinction threatens very small populations, which are particularly vulnerable to uneven age and gender distribution. For example, the disappearance of one gender automatically leads to the extinction of the entire population.

The application code

The *Basic LifeCycle with birth control* simulator is implemented through the script MAIN-Lifecycle-Control.sql. It calls the same 7 procedures as the *LifeCycle* simulators:

- _AGENT-Init-Parameters
- _AGENT-Create-Tables
- _AGENT-Get-Parameters
- _AGENT-Create-World

– _AGENT-Populate-World
– _AGENT-Generate-Current-State

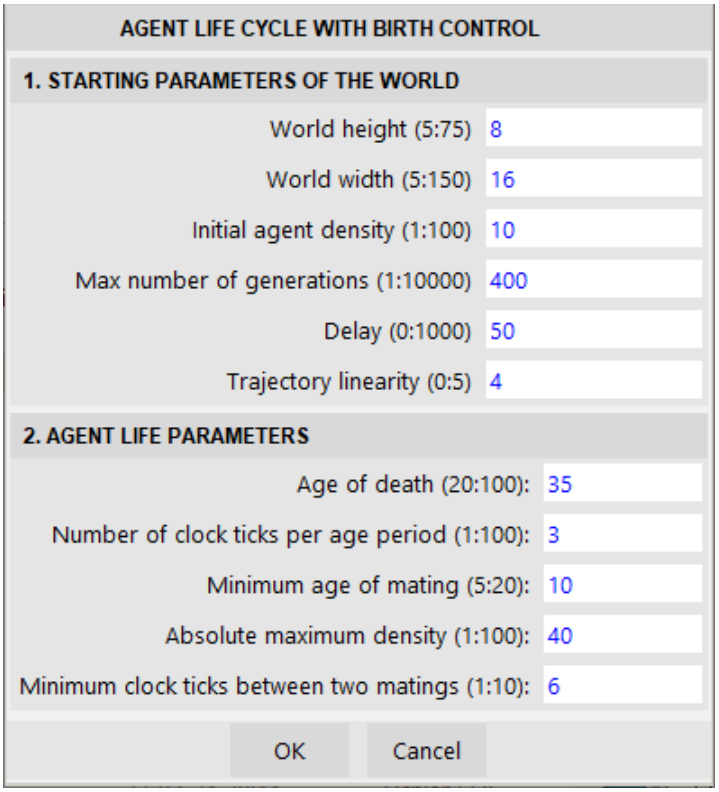


Figure 24.26 - Control panel of the Basic *Agent Life Cycle with birth control* simulator

24.13.4 Refining the birth control model

The formula for calculating the matchingProba parameter is an abstract rule that stands out in a model whose rules, so far, describes as closely as possible the concrete behavior of a population of living entities in the real world. The result is a hybrid model in which certain components, because of their abstract nature, are no longer representative of the real system.

Wouldn't it be possible to replace this equation with an explicit representation of the concrete objects and behaviors that govern the evolution of populations?

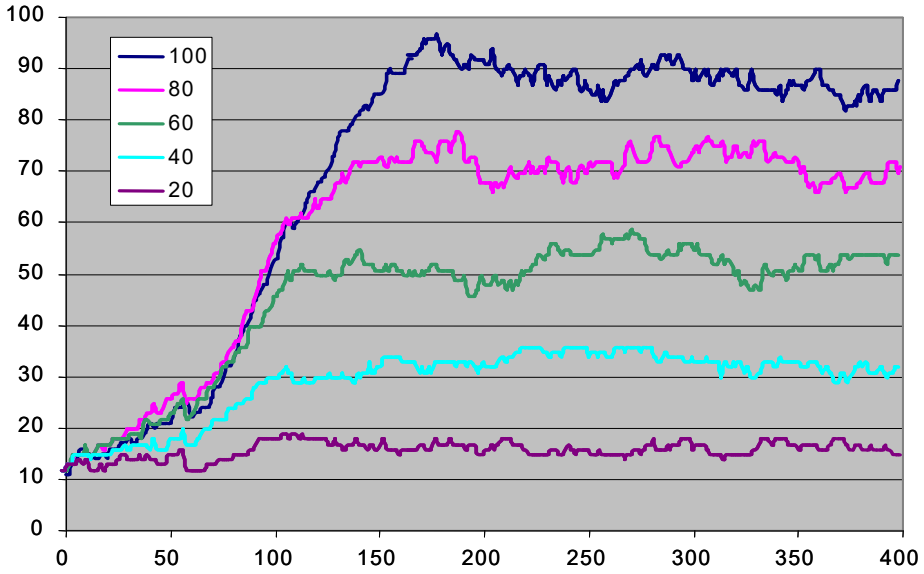


Figure 24.27 - LifeCycle simulations with birth control

Let's consider one of the most obvious sources of this phenomenon: the *limited availability of food*, as suggested in the introduction. An agent must feed to remain active. It finds food in every cell it visits. Once the food in a cell has been consumed, it is replenished by a certain amount with each step. An agent's ability to reproduce depends on its state of satiety: a better-fed agent tends to reproduce more *dynamically*. A malnourished agent, on the other hand, will not reproduce and may even die of starvation.

Here are a few ideas for refining the agent model (Figure 24.28). Their implementation is left to the reader's initiative.

Facts about the system	**Modeling suggestions**
each cell contains a variable quantity of food,	add column **Food** to table **CELL.**
this quantity increases with each unit of time (i.e., each step); it decreases when an agent consumes it,	static parameters **foodInc** (increase per clock tick) **foodDec** (consumption by agent per clock tick).
the maximum amount of food in a cell,	static parameter **maxFood**.
every agent has a health capital,	add column **Health** to table **AGENT.**

this health level is limited to a maximum. It increases with the amount of food consumed. It decreases with each unit of time,	static parameters **maxHealth**, **foodHealthRatio** (level increase per unit of food consumed), **healthPerStep** (level loss per step).
when an agent's health level decreases, its ability to reproduce decreases,	dynamic parameter **matchingProba** is no longer global but depends on the health level of each agent; it is also possible to act on the **maturity** and **latency** parameters; the value of these parameters is calculated at each step by each agent.
as well as its life expectancy,	*ditto* for the **maxAge** parameter (premature death) or perhaps **ticksPerAge** (accelerated aging).
an agent dies as soon as its health level drops to zero.	

Figure 24.28 - A better description of the evolution of the population

24.13.5 Toward learning agents

The behavior of agents, no matter how sophisticated, obeys rules set in advance and applied in the same way by each agent. In other words, agents' knowledge of their environment is constant from birth to death. This knowledge is therefore non-evolving, which raises the question of *learning*: can an agent, through experience or exchanges with other agents, enrich the knowledge it has of its environment and thus behave more efficiently or more safely?

To illustrate this point, let's imagine that the food found in the cells are plants of two varieties, which the agent initially ignored. Some are edible, others toxic. When an agent eats a toxic plant, it gets sick, resulting in an abnormal decrease in its health level. While the standard decrease is one unit per step, ingesting a toxic plant results in a loss of, say, three units, which weakens the agent and can even lead to death. The combination of these two events - the ingestion of a toxic plant and the drop in health - can provide the agent with new knowledge that will help it survive.

We can imagine several ways to implement the reaction of the agent to this danger. One, let’s say more *agent-oriented*, would be to interpret a health level loss of three units as an event that activates a specific trigger. The latter would analyze the environment of the agent, infers the rule *toxic food in the current cell* $\longrightarrow$ *danger* and records it in the state of the agent.

What's more, when two agents meet (they visit the same cell, for whatever reason), they can share their knowledge of which plants to avoid, thus contributing, through a peer-to-peer process, to the development of a collective knowledge.

24.14 Logging elementary operations and debugging

Being more technical, this section is intended for experienced experimenter.

So far, we've designed and developed the code for the *Basic* simulators. The existence of another variant, called *Extended*, has only been mentioned. We will now give more details on this version.

The code of the controller and of the triggers of the Basic version has been enriched with instructions that inform the user/developer about the intermediate states of the world (agent population and space) and of the elementary operations of the agents at each step of the simulation. This additional code is activated or disabled in the INFORMATION part of the control panel. It is intended,

- to better understand the way the agents behave,
- to track and debug the logic errors, in particular in the body of the triggers.

This practice is of general use in the development of complex programs, where it is called *code instrumentation*.[34]

Let us examine how it works by running the *Attractors-Repellers* simulator to create a small world populated with 5 agents and 4 distant repellers. Its starting state is shown in Figure 24.29

Through the settings of the INFORMATION part of the control panel (Figure 24.30), we ask the simulator to print in the output window,

- the initial and final states of the cells,
- at each step, the resulting state of the cells (Show grids),
- at each step, for each agent, the sequence of the elementary actions carried out (Show moves).

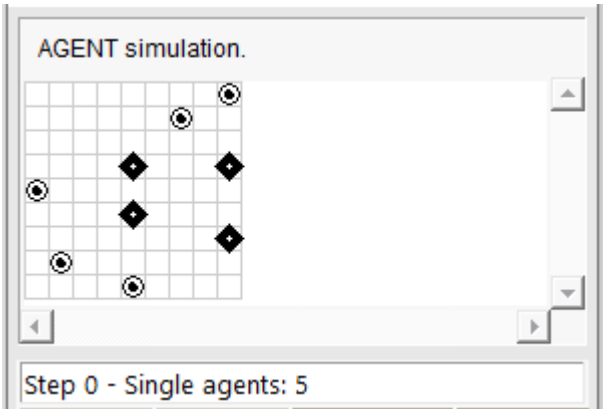


Figure 24.29 - Starting state of the simulation

34. https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)

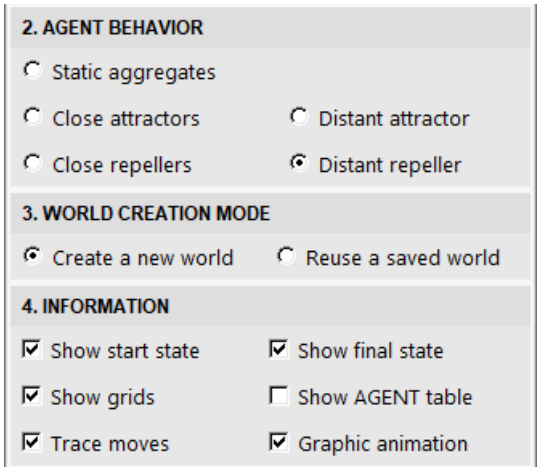


Figure 24.30 - Information settings

From the (fairly long) trace of the simulation, we examine what happened during the first step. The trace, an excerpt of which is shown in Figure 24.31, comprises the initial state of the space[35], the detail of the actions performed by each of the five agents during the execution of step 1 then the new state of the cells at the end of this first step. Actually, we focus on the behavior of agent #2 (fragment in blue color), initially located in cell (Y,X) = (2,7). Its trace, entitled Processing Agent #2 in (2,7) [0,0][36], comprises four paragraphs:

1. *Identifying distant repellers*

   In this action, the agent identifies its two distant repellers and collects their position in relative distances: `[2,2;-2,2]`[37].

   Then, it computes the union of their vicinity: [2,2;-2,2;-3,1;-2,1;-1,1;1,1;2,1;-3,2;-1,2;1,2;-3,3;-2,3;-1,3;1,3;2,3],

2. *Selecting the next move*

   The agent computes its own vicinity: [-1,1;1,1;-1,-1;0,-1;1,-1;-1,0;0,0;1,0;0,1] ,

   By computing the set difference of these cell coordinates, it derives the *clean cells* of its vicinity, that is, those that do not belong to the vicinity of the repellers [-1,-1;1,-1;0,1;0,0;1,0;-1,0;0,-1].

   Finally, it selects, at random, one of these cells: **[0,1]**.

35. Remember that repellers are coded by Content = 9.
36. This title indicates the current (Y,X) position and the values of (OldX,OldY) of the agent.
37. (X offset,Y offset)

3. *Selecting deltas*

   Since avoiding the repellers is of highest priority, the trajectory style is ignored. The next move is the direction selected above.

4. *Final decision*

   The destination cell is computed: (Y,X) = (3,7).

The resulting space of the first step closes this trace.

```
   1 2 3 4 5 6 7 8 9
 1 . . . . . . . . 1
 2 . . . . . . 1 . .
 3 . . . . . . . . .
 4 . . . . 9 . . . 9
 5 1 . . . . . . . .
 6 . . . . 9 . . . .
 7 . . . . . . . . 9
 8 . 1 . . . . . . .
 9 . . . . 1 . . . .

-- Execute generation 1 --

Processing Agent #1 in (1,9) [0,0]
...
- Final decision:
  Agent #1 moves from (1,9) to (1,9)

Processing Agent #2 in (2,7) [0,0]
- Identifying distant repellers:
  Distant repellers:           [2,2;-2,2]
  Vicinity of repellers:       [2,2;-2,2;-3,1;-2,1;-1,1;1,1;2,1;
                                -3,2;-1,2;1,2;-3,3;-2,3;
                                -1,3;1,3;2,3]
- Selecting the next move:
  Vicinity of the agent:       [-1,1;1,1;-1,-1;0,-1;1,-1;
                                -1,0;0,0;1,0;0,1]
  Clean vicinity of the agent: [-1,-1;1,-1;0,1;0,0;1,0;-1,0;0,-1]
  Selected direction:          [0,1]
- Selecting deltas:
  Selected deltas of (2,7): 0,1
- Final decision:
  Agent #2 moves from (2,7) to (3,7)

Processing Agent #3 in (5,1) [0,0]
...
- Final decision:
  Agent #3 moves from (5,1) to (4,2)

Processing Agent #4 in (8,2) [0,0]
...
- Final decision:
  Agent #4 moves from (8,2) to (8,3)

Processing Agent #5 in (9,5) [0,0]
- Final decision:
  Agent #5 moves from (9,5) to (8,4)

...
```

```
  1 2 3 4 5 6 7 8 9
1 . . . . . . . . 1
2 . . . . . . . . .
3 . . . . . . 1 . .
4 . 1 . . 9 . . . 9
5 . . . . . . . . .
6 . . . . 9 . . . .
7 . . . . . . . . 9
8 . . 1 1 . . . . .
9 . . . . . . . . .
```

Figure 24.31 - Excerpts of the trace of a simulation (distant repellers)

24.15 SQL coding in SQLfast

This material has been borrowed from the section ***Coping with the limitations of the trigger language*** *of Case study* ***Active Databases****.*

The body of the triggers developed in this study is written in some kind of *pseudocode* that makes the algorithm structure readable but is not executable as such. The reader of this study may be surprised when comparing this code with that of the SQLfast applications. There are two reasons. First, the syntax of the triggers in SQLite, which is also that in SQLfast, strictly complies with the standard that defines the body of a trigger as a *sequence of SQL statements*. So, usual features of programming languages such as *variables*, *if-then-else* control statements and *procedure calls* are not allowed[38]. This point is discussed below. The second reason is functional. The actual code of the triggers comprises ancillary functions without interest at the problem modeling phase. Such is the case of the *tracing functions* Show grids and Trace moves (enabled in the INFORMATION part of the control panels) executed by the triggers.

We show now that we can easily compensate for limited syntax of the trigger body.

Simulating variables

Usually, a value used in several statements of the trigger is stored in a local variable. This is particularly recommended if the acquisition cost of this value is high, for instance, if it is computed by an aggregation query.

If the concept of local variable is missing, a technical table containing a single row, where each column implements a variable can be used instead, as illustrated by table **Var1** in Script 24.36.

38. Most major DBMSs provide a richer programming language to write stored procedures and triggers.

```
create table Var1(DeltaX integer, DeltaY integer);
create trigger ...
begin
   ...
   insert into Var1(DeltaX,DeltaY)
      select ...
      from   AGENT
      where  AgentID = new.AgentID;

   update AGENT
   set X = X + (select DeltaX from Var1),
       Y = Y + (select DeltaY from Var1)
   where AgentID = new.AgentID;;
   ...
   delete from Var;
end;
```

Script 24.36 - Single-row table **Var1** acts as a series of local variables

This example also shows how the query form `select ... into DeltaX,DeltaY` is simulated with the form `insert into Var1(DeltaX,DeltaY) select ...`.

Simulating *if-then-else* statements

If the trigger language of the DBMS does not include *if-then-else* alternative control statement, then Script 24.37, in which <C>, <C1> and <C2> denote arbitrary SQL conditions, is invalid.

```
create trigger TRG_DEL_T
before delete on T
begin
   if (<C>)
      then delete from S where <C1>;
      else update S set B = null where <C2>;
   endif;
end;
```

Script 24.37 - Standard if-then-else alternative

We suggest two methods to transform this trigger into pure sequential code. The first one consists in adding respectively <C> and not <C> to each branch of the alternative (Script 24.38). If the evaluation of <C> is expensive, its result can be stored in a local variable, that will be used in next statements as a substitute for <C>.

```
create trigger TRG_DEL_T
before delete on T
begin
   delete from S where <C1> and <C>;
   update S set B = null where <C2> and not <C>;
end;
```

Script 24.38 - Transformation of if-then-else alternative - Method 1

According to the second method, each branch is translated into a *distinct trigger* the `when` clause of which decides whether it will fire (Script 24.39).

```
create trigger TRG_DEL_T1
before delete on T
when <C>
begin
   delete from S where <C1>;
end;

create trigger TRG_DEL_T2
before delete on T
when not <C>
begin
   update S set B = null where <C2>;
end;
```

Script 24.39 - Transformation of if-then-else alternative - Method 2

It is important to note that the validity of these transformations depends on whether the result of the execution of the first branch affects the evaluation of the second instance of condition <C>. For example, in Script 24.38, if <C> is true, the first statement is executed. However, this execution may make condition <C> false, in which case both branches of the alternative will be executed.

Calling an external procedure

Most DBMS allow the body of a trigger to ask the execution of a procedure written in some proprietary or general programming language. Such a statement will generally look like this one (Postgresql):

```
execute procedure printMessage('Agent #'||old.AgentID);
```

or (Oracle):

```
printMessage('Agent #'||old.AgentID);
```

The procedure `'printMessage'` is intended to write a message on the user console. 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 native or UDF function that (optionally) returns a value (which is ignored) after executing some desired action (the interesting part). So, instead of

```
printMessage('Agent #'||old.AgentID);
```

we will write:

```
select printMessage('Agent #'||old.AgentID);
```

In the same way, instead of:

```
if (<condition>) printMessage('Agent #'||old.AgentID);
```

we will write:

```
select printMessage('Agent #'||old.AgentID) where <condition>;
```

If we find this syntax a bit awkward, we can improve it as follows:

```
set message = select printMessage;
```

Then, wherever in the body of a trigger we want to write a message in the output window, we simply write:

```
$message$('Agent #'||old.AgentID') where $showMove$;
```

Case study **27**

Conway's Game of Life

Objective. This study is about games, worlds, life and death, borderline SQL applications and dramatic database optimization. The goal of the project is to implement the graphical animation of Conway's cellular automata, aka *Game of Life*. A game of life is made up of an infinite array of cells in which live a population of small *animals*, each of them occupying one cell. The transition of one state of the population to the next one is specified by a set of simple computing rules. The goal of the game is to observe and study the evolution of the population. A game of life is implemented as a table in a database in which each row contains the coordinates and the content of a cell. The algorithms developed in this study load the initial state of a population then compute the next states thanks to the evolution rules. Finally, they visualize this evolution as an animated cartoon. The contribution of this study is twofold. It stresses the importance of database and algorithm optimization (the last version is 1,400 times faster than the first one) and it shows that relational databases and SQL may be quite efficient to develop matrix manipulation procedures (the SQL version is nearly 7 times faster than the equivalent Python program).

This study is also a tribute to E. F. Codd, the inventor of the relational model of databases, who first studied self-replicating cellular automata.

Keywords. cellular automata, replicating system, Conway, glider, Codd, matrix manipulation, algorithm optimization, database optimization, declarative algorithm, table indexing, in-memory database, CTE, recursive query, vector graphics, animated simulation, Python.

27.1 Introduction

Cellular automata are mathematical objects invented by J. H. Conway in 1970 in search for self replicating systems. A cellular automaton lives in a 2-dimensional grid made up of square cells. It has an initial state that evolves as a sequence of states derived according to a small set of elementary transformation rules. Each state is defined by certain cells containing a small critter of which we only know that it is present or not. The presence of such an animal in a cell (which is then called a *live cell*) is represented by a black square while an empty cell (a *dead cell*) is white.

In Figure 27.1, the grid shows the initial state of the most simple *glider* (top), a popular object that moves like a wriggling, hyperkinetic worm crawling down the grid diagonal. It comprises 5 live cells. After 4 steps, it recovers its form but one step down along the diagonal. And so on. Forever ...

The evolution rules are very simple. To compute the next state of a cell of an automaton, we count those of its eight neighbors that are live. Let N be this number. Then we apply the following transformation rules:

– if N = 2, the cell contents is preserved, be it live or dead,

– if N = 3, the cell becomes (or remains) live,

– for all the other values of N (0, 1, > 3), the cell becomes dead.

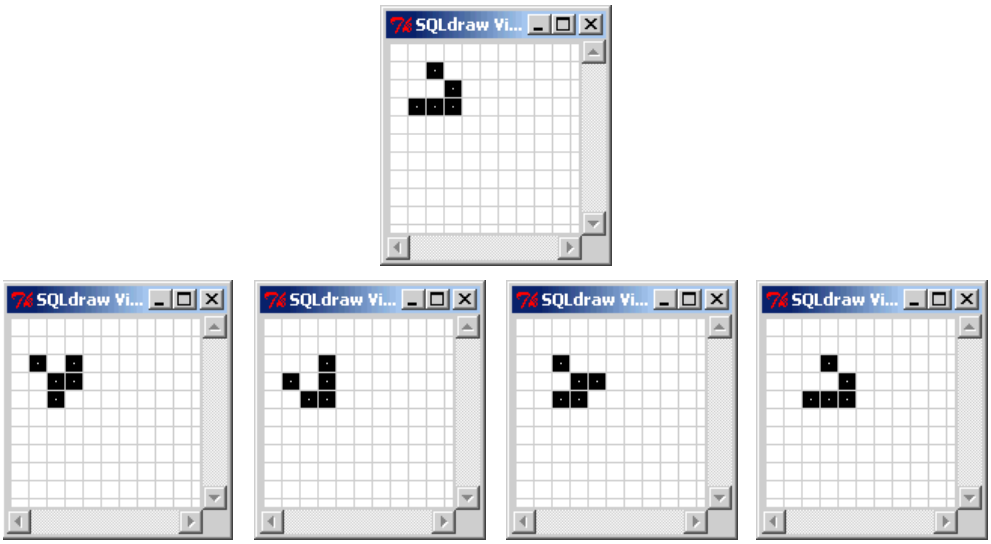


Figure 27.1 - The starting first five states of a simple Glider

For reasons that are not hard to guess, this system has been given the popular name of *Game of Life*. It has triggered much research among computer science theoretists[1], epidemiologists studying how diseases spread and vanish[2] and in some gamer

communities[3]. In particular, several families of patterns have been identified: *still* (each state is the same as the initial one), *gliders* (moving patterns), *guns* (that oscillate while producing *shells*, that may be gliders), *oscillators*, *mirrors*, *bumpers*, *spaceships*, etc. Some patterns move, grow infinitely[4], vanish, oscillate, replicate themselves or produce other patterns. Just like societies in real life! See http://en.wikipedia.org/wiki/Conway's_Game_of_Life as a good starting point.

This topic may come as a surprise in a text devoted to databases and database programming[5]. Indeed, animating cellular automata mainly requires much computation and fast graphical rendering. Not precisely an application domain in which database technology is renowned for! Or is it?

However, by trying to develop cellular automata with the SQLfast environment, we will encounter several interesting challenges the solution of which can be generalized to other, more mainstream, database problems.

In addition, we will once more appreciate the expressiveness and power of SQL. As an appetizer, just consider that cellular automata can be elegantly and efficiently implemented **in 2 (two) SQL queries**.

27.2 World and automata representation

Cellular automata live in a infinite world. Due to reason that are not worth being made explicit, we will reduce them to finite worlds, represented by arrays. A world is represented by table CELL in which each row describes a cell of an array. CELL has three columns:

- **I**, that indicates the line number of the cell,
- **J**, that represents the column number of the cell,
- **C**, that indicates whether the cell is live (`1`) or dead (`0`).

For instance, row `(4,4,1)` tells that cell `4,4` is live while row `(3,3,0)` denotes a dead cell. Table CELL is defined as follows:

```
create table CELL(I integer, J integer, C integer);
```

1. The rules mentioned above define a system that have been proved to be Turing-complete.
2. Thanks to more sophisticated automata, with more than two states (*healthy*, *infected*, *immune/dead*) and probabilistic evolution rules
3. OK, not quite the same as that of WoW!
4. Seemingly, because this cannot be neither observed nor proved.
5. It is interesting to note that E. F. Codd, the inventor of the relational model of databases, formerly worked on cellular automata. See for instance E. F. Codd, *Cellular Automata*, Academic Press, New York (1968)

Its primary key is (I,J) but we do not declare it for now. Similarly, we do not declare column *not null*. The scripts that will make use of this table (as well as of the second one) are guaranteed to insert correct data.

Computing the next state of an automaton requires that we can analyze all the cells of the current state unmodified. Therefore, we need an additional array in which we will store the data required to compute the next state without altering the current state. In this array, we will store, for each cell, the number of live neighbours. Then, from this information, we will be able to update the contents of table CELL to create the next state by applying the evolution rules.

This additional array is implemented as table SCORE, with the same structure as CELL, and in which column N represents the number of live neighbors of cell (I,J):

```
create table SCORE(I integer, J integer, N integer);
```

For instance, row `(2,3,1)` of SCORE tells that cell `(2,3)` in CELL has only `1` live neighbor, which is no good news for its viability! Same for the cell corresponding to row `(3,3,5)`. On the contrary, row `(4,4,2)` tell that cell `(4,4)` will keep its former state.

Now, we are able to create the database (Script 27.1). Since it will be used only to compute the successive steps, we declare it as a non persistent, *in-memory* database. A world that comprises **maxLin** lines and **maxCol** columns is initialized by the insertion of **maxLin** x **maxCol** cells with `C = 0` (Script 27.2). To make the graphical layout more elegant, and to make cell evaluation easier, we leave the first line and column empty, that is, they are not part of the world. For our experiments, a 50 X 50 square world will prove comfortable enough. Later, we will show that this algorithm is not fast enough for our ambition, so that we will call it *First* (or *naive*) *version*.

We also install the initial state of the *glider* of Figure 27.1 (Script 27.3).

```
createDB InMemory;
create table CELL (I integer,J integer,C integer);
create table SCORE(I integer,J integer,N integer);
```

Script 27.1 - Life Game: creation of the database

```
for I = [0,$maxLin$];
  for J = [0,$maxCol$];
    insert into CELL values ($I$,$J$,0);
  endfor;
endfor;
```

Script 27.2 - Life Game: creation of a **maxLin x maxCol** world - First version

```
update CELL set C = 1 where I = 1 and J = 2;
update CELL set C = 1 where I = 2 and J = 3;
update CELL set C = 1 where I = 3 and J = 1;
update CELL set C = 1 where I = 3 and J = 2;
update CELL set C = 1 where I = 3 and J = 3;
```

Script 27.3 - Life Game: creation of a Glider

27.3 Computing the next state

To compute the next state of the automaton we proceed in two steps: computing the number of live neighbor cells and updating the current state.

First, we compute the number of live neighbors of each cell and we store these numbers in table SCORE. This can be done in a single **`insert`** SQL query. The current cell under examination is denoted by alias **`C0`** while its neighborhood is explored by alias **`C1`**. The relation between **`C0`** and **`C1`** (i.e., the *join condition*) is that coordinate **`I`** of **`C1`** is between **`C0.I`**`-1` and **`C0.I`**`+1` and coordinate **`J`** of **`C1`** is between **`C0.J`**`-1` and **`C0.J`**`+1`. But we must exclude **`C0`** itself of course. Value **`N`** of cell **`C0`** is the sum of the values of **`C`** of its neighbor cells. Hence the query of Script 27.4.

The second step consists in updating column **`C`** of each **`CELL`** row according to the number of its live neighbors, previously stored in **`SCORE`**. This is expressed by a `case` construct that assigns the appropriate (`0`,`1`) value to column **`C`** (Script 27.5).

```
insert into SCORE(I,J,N)
select C0.I, C0.J, sum(C1.C)
from   CELL C0, CELL C1
where  C1.I between (C0.I - 1) and (C0.I + 1)
and    C1.J between (C0.J - 1) and (C0.J + 1)
and    not (C1.I = C0.I and C1.J = C0.J)
group by C0.I, C0.J;
```

Script 27.4 - Computing the neighborhood of the current state - First version

27.4 Graphical display of automaton evolution

Technically speaking, the job is done. However, it would be nice to show graphically, as a *cartoon*, the successive states of an automaton. We suggest to convert

these states into a sequence of vector graphics pictures expressed in the SQLdraw language. These pictures, written in a text file with extension ***.draw**, can then be rendered through statement **showDrawing** (see Chapters 13 and 21 of the Tutorial).

```
update CELL
set C = (select case
                when N<2 then 0
                when N=2 then CELL.C
                when N=3 then 1
                when N>3 then 0 end
         from SCORE S
         where S.I = CELL.I and S.J = CELL.J);
```

Script 27.5 - Deriving the next state from the current one - First version

The first part of this file, shown in Script 27.6, defines the useful part of the world (**area** `192,192`) expressed in pixels. Its origin (`0,0`) is at the top-left corner of the graphical space. In this parts the border of the cells are drawn as a grid made up of two polylines:

`polyline "Vgrid",1,"gray80"`: defines the vertical lines of the grid

`polyline "Hgrid",1,"gray80"`: defines the horizontal lines of the grid.

A polyline is a contiguous chain of segments defined by their connection points. Statement **polyline** assigns a name to the polyline and specifies the thickness and the color of its segments (the fourth parameter is ignored). It is followed by the list of the coordinates of its points.

```
area 192,192
polyline "Vgrid",1,"gray80",""
0,192
0,0
0,192
12,192
12,0
...
polyline "Hgrid",1,"gray80",""
192,0
0,0
192,0
192,12
0,12
...
```

Script 27.6 - SQLdraw script creating the grid of a game

When a state is available, we generate the SQLdraw command **points** that displays its current population. This command comprises a header, follows by a series of points, each positioned on a live cell. The header assigns a name to the set of points

and specifies its shape: thickness and color of its border polygon, filling color and the point type, here type 23 (see the point catalogue in Chapter 21 of the Tutorial). Point type 23 is a black 11x11 pixel square with a small white hole in its center. If we allow room for a separation of 1 pixel for the grid, each cell is 12 pixel large. Each point is defined by its coordinates and an optional label. The coordinates of a point are those of its center, so, the graphical coordinates of cell I,J are (6 + 12 * J, 6 + 12 * I).

Animation is created as follows. Once a state has been drawn, a wait period of, say, 150 ms., is started (`wait 150`), after which, the last state is erased (`delete last`). Then, the next state is drawn.

For instance, the first two states of the *Glider* automaton (Figure 27.1) can be displayed by Script 27.7.

```
points "life-0",1,"black","black",23
30,18,""
42,30,""
18,42,""
30,42,""
42,42,""
wait 150
delete last
points "life-1",1,"black","black",23
18,30,""
42,30,""
30,42,""
42,42,""
30,54,""
wait 150
delete last
```

Script 27.7 - SQLdraw script of the first two states of the *Glider*

27.5 Generation of SQLdraw scripts.

Script 27.8 generates the **`points`** command of the current state from the contents of table CELL. Variables Lnbr and Cnbr specify the numbers of the last line and column of the world. Note that the `delete` statement is lacking. It is generated elsewhere to avoid erasing the very last state.

27.6 Packaging the LIFE application

We have enough material to define the architecture of our LIFE application. Remember that the application available in directory **Case_Life_Game** may be a bit

different (more sophisticated) from the description given below. More on this in the last section.

The main script could look like Script 27.9.

```
write points "life-$seq$",1,"black","black",23;
for I = [0,$Lnbr$[;
   for J = [0,$Cnbr$[;
      extract R = select C from CELL where I = $I$ and J = $J$;
      if ($R$ = 1);
         compute X = 6 + $J$ * 12;
         compute Y = 6 + $I$ * 12;
         write $X$,$Y$,"";
      endif;
   endfor;
endfor;
if ($Delay$ > 0) write wait $Delay$;
```

Script 27.8 - Generation of the SQLdraw script of the current state - First version

It first comprises a simple dialogue to collect the useful world size (variables `Lnbr` and `Cnbr`), the number of successive states to compute (variable `Niter`) and the wait delay, in ms (variable `Delay`). Then, it calls the three database creation procedures: _LIFE-Create-Database.sql for the database, _LIFE-Create-World.sql that inserts the world cells in the database and _LIFE-Create-Glider.sql that installs the initial state of the *Glider* of Figure 27.1. Procedure _LIFE-Compute-History.sql is in charge of computing and generating the `Niter` successive states of the automaton in text file `LIFEGAME-Glider.draw`. Finally, it shows the animated rendering of the Niter states through statement **`showDrawing`**.

```
set Lnbr,Cnbr,Niter,Delay = 15,15,40,150;
ask-u Lnbr,Cnbr,Niter,Delay = [Simulation parameters]
      Lines:|Columns:|Iterations:|Delay (ms):;
   if ('$DIALOGbutton$' = 'Cancel') exit;

execSQL _LIFE-create-Database.sql;
execSQL _LIFE-create-World.sql;
execSQL _LIFE-create-Glider.sql;
execSQL _LIFE-compute-History.sql;
showDrawing LIFEGAME-Glider.draw;
```

Script 27.9 - The main script of the LIFE application

Procedure _LIFE-Create-Database.sql includes the statements of Script 27.1. Procedure _LIFE-Create-World.sql is defined from Script 27.2 while procedure _LIFE-Create-Glider.sql is defined from Script 27.3.

The most interesting procedure obviously is _LIFE-Compute-History.sql, that does all the job. Its code is shown in Script 27.10. It starts by opening external file LIFE-GAME-Glider.draw, that will receive the SQLdraw statements [1] and finishes by redirecting the output channel to `window`, which closes the external file [7].

Statements [2] and [3] insert the points (the *live cells*) of the starting state of the automaton. For each of the following states,

- statement [4] inserts an SQLdraw `delete` statement to erase the last state from the drawing area,
- statement [5] computes the next state as suggested in Scripts 27.4 and 27.5,
- statement [6] inserts into the SQLdraw script the points (the *live cells*) of the current state.

```
outputOpen LIFE-draw.draw;                                    [1]
set seq = 0;                                                  [2]
execSQL _LIFE-Generate-Current-State.sql;                     [3]
for seq = [1,Niter];
   write delete last;                                         [4]
   execSQL _LIFE-Compute-Next-Step.sql;                       [5]
   execSQL _LIFE-Generate-Current-State.sql;                  [6]
endfor;
outputAppend window;                                          [7]
```

Script 27.10 - Procedure *_LIFE-Compute-History.sql*

27.7 First performance analysis

We would like the LIFE application to show the result as soon as the parameters have been entered. Unfortunately, this is not the case, even if the speed of computing and generating 30 states of the *Glider* automaton in a 15x15 world seems to be acceptable with this respect: 2.6 s. We guess than wider worlds, larger automata examined through hundreds of states will require much longer computing and generation time.

Trying to improving the performance of the LIFE application is an interesting challenge. It will allow us to better understand how a database and a database application work and to experiment with various ways to boost the performance of a database.

To measure the impact of performance improvement techniques, we need a reference point. Computing 100 successive states of the *Glider* automaton in a 60x60 world seems a good starting point: it is sufficiently small to imply a *bearable* computing time with the current version and sufficiently large to let us compare the

successive optimization. The run times of the main four procedures are shown in Figure 27.2. A total of about **14** minutes is quite impressive! Not surprisingly, the major part of the cost (> 99.6%) is that of procedure LIFE-compute-History.sql.

procedure	raw version
Create-Database	0.000
Create-World	2.7
Create-Glider	0.002
Compute-History	835.715
Total	**838.417**

Figure 27.2 - Computing times of the raw version (Glider, 60X60 world, 100 iterations)

27.8 Optimization 1: indexing tables

Let us first examine the way the database is used. Procedure LIFE-Create-World.sql is a sequential process while LIFE-Create-Glider.sql makes a small number of random updates. There is no real room for improvement in the first three procedure, at least for now.

Procedure LIFE-Compute-History.sql is our main target. It includes heavy database access and update queries:

- In procedure LIFE-Compute-Next-Step, query 27.4 executes a *self-join* followed by a *group by* operator on table CELL. Both operators are based on columns **I** and **J**.
- In procedure LIFE-Compute-Next-Step, query 27.5 is a massive update of table CELL from table SCORE. The 60x60 cell updates require each a random access to SCORE based on (**I**, **J**) values.
- In procedure LIFE-Generate-Current-State, the double loop executes, for each state, 60x60 random accesses to table CELL, that is, 3,600 `select` queries on column **I** and **J**, that is, 360,000 queries for the 100 states.

From this simplified analysis, we conclude that an index on column **I** and **J** on both tables CELL and SCORE should decrease access time in all these queries.

Let us try that. We change the contents of procedure LIFE-Create-Database.sql by Script 27.11, that declares these indexes.

```
createOrReplaceDB LIFE.db;
create table CELL (I integer,J integer,C integer);
create table SCORE(I integer,J integer,N integer);
create index XCELL  on CELL (I,J);
create index XSCORE on SCORE (I,J);
```

Script 27.11 - Life Game - Creation of the *LIFE.db* database with indexes

The resulting execution times (Figure 27.3) are quite encouraging.

procedure	raw version	indexed
Create-Database	0.000	0.001
Create-World	2.7	2.7
Create-Glider	0.002	0.001
Compute-History	**835.715**	**442.752**
Total	**838.417**	**445.454**

Figure 27.3 - Indexing the database more than halves the run time

Encouraging but still excessive. Waiting more than 7 minutes for the show will not attract a large audience! Actually, procedure LIFE-Compute-History.sql mixes database queries and non-database statements, so that its global execution time figure is too gross to allow us to identify other ways of improvement.

Splitting this time into computation time (LIFE-Compute-Next-Step) and generation time (LIFE-Generate-Current-State) could be quite informative. We change the code of script LIFE-Compute-History.sql a little bit, by deactivating the two statements **`execSQL`** `LIFE-Generate-Current-State.sql` (we merely prefix them with the `'--'` comment indicator). The resulting times when executing the raw and indexed versions are stunning (Figure 27.4): the time drops from 566.45 s. to 6.36 s.! Subtracting these times from the global times provide the execution time of LIFE-Generate-Current-State.[6]

6. To be quite precise, this time also includes the execution time of the proper statements of procedure LIFE-compute-History.sql and of procedure LIFE-generate-grid.sql. However, these times are very small and can be ignored. For instance, generating the 60x60 grid takes 0.4 sec.

procedure	raw version	indexed
Create-Database	0.000	0.001
Create-World	2.7	2.7
Create-Glider	0.002	0.001
Compute-History	835.715	442.752
- Compute-Next-Step		**16.547**
- Generate-Current-State		**426.205**
Total	**838.417**	**445.454**

Figure 27.4 - Splitting times between computing and generation

Clearly, our optimization effort should now concentrate on the **generation process**.

27.9 Optimization 2: *windowing* table access

The SQL queries of procedures LIFE-compute-History.sql and LIFE-Generate-Current-State visit all the cells of the world (and, in parallel, those of the SCORE table, which has the same size). In fact, when we consider the unitary transition from one state to the next one, we observe that the set of cells that could change often is quite small.

Considering the fourth state of the *Glider* automaton represented in Figure 27.1, the cells that are likely to change in the next state are (1) the five live cells and (2) their seventeen neighbor cells. They are shown in the left array of Figure 27.5: live cells in black and their neighbors in gray. So, we only need to compute the next state for these cells, ignoring all the white cells. Similarly, when generating the SQLdraw statements of the current state, examining the white cells would be pure wasted time.

However, the area defined in this way can be irregular and quite difficult to define. We will therefore consider a simpler area, a bit larger than necessary, but easier to compute: the *bounding rectangle*. It comprises the smallest rectangle that completely contains the automaton plus a border one cell large (Figure 27.5, right array).

The coordinates (I1,I2,J1,J2) of this bounding rectangle can be computed by the following SQL query:

```
select min(I)-1 as I1, max(I)+1 as I2,
       min(J)-1 as J1, max(J)+1 as J2
from CELL where C = 1;
```

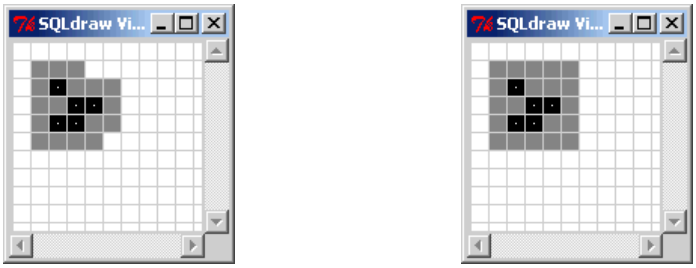


Figure 27.5 - The bounding area of an automaton (strict and rectangular)

This formula must be refined a little bit to avoid negative values for the cells at the edge of the world:

```
select max(min(I)-1,0) as I1, min(max(I)+1,$Lnbr$-1) as I2,
       max(min(J)-1,0) as J1, min(max(J)+1,$Cnbr$-1) as J2
from CELL where C = 1;
```

Its translation in the SQLfast syntax is straightforward:

```
extract I1,I2,J1,J2 =
        select max(min(I)-1,0),min(max(I)+1,$Lnbr$-1),
               max(min(J)-1,0),min(max(J)+1,$Cnbr$-1)
        from CELL where C = 1;
```

We can now adapt (what we will call *windowing*) the SQL queries so that they only examine and process the cells that lie in the (I1,I2,J1,J2) rectangle (Scripts 27.12 and 27.13). Let us now measure the effect of this optimization (Figure 27.6).

procedure	raw	indexed	window
Create-Database	0.000	0.001	0.001
Create-World	2.7	2.7	2.7
Create-Glider	0.002	0.001	0.001
Compute-History	835.715	**442.752**	**4.17**
- Compute-Next-Step		**16.547**	**0.33**
- Generate-Current-State		**426.205**	**3.84**
Total	**838.417**	**445.454**	**6.87**

Figure 27.6 - The effect of windowing SQL queries

The improvement is really dramatic: the execution time of LIFE-Compute-History has been divided by more than **100**!

```
insert into SCORE(I,J,N)
   select C0.I, C0.J, sum(C1.C)
   from   CELL C0, CELL C1
   where  C0.I between $I1$ and $I2$
   and    C0.J between $J1$ and $J2$
   and    C1.I between (C0.I - 1) and (C0.I + 1)
   and    C1.J between (C0.J - 1) and (C0.J + 1)
   and    not (C1.I = C0.I and C1.J = C0.J)
   group by C0.I, C0.J;

update CELL
set C = (select case
                when N<2 then 0
                when N=2 then CELL.C
                when N=3 then 1
                when N>3 then 0
                end
         from SCORE S
         where S.I = CELL.I and S.J = CELL.J)
where  I between $I1$ and $I2$
and    J between $J1$ and $J2$;
```

Script 27.12 - *Windowing* the SQL queries of procedure LIFE-Compute-History.sql

```
for I = [$I1$,$I2$[;
   for J = [$J1$,$J2$[;
      extract R = select C from CELL where I = $I$ and J = $J$;
      if ($R$ = 1);
         compute X = 6 + $J$ * 12;
         compute Y = 6 + $I$ * 12;
         write $X$,$Y$,"";
      endif;
   endfor;
endfor;
```

Script 27.13 - *Windowing* the loops of procedure LIFE-generate-Current-State.sql

27.10 Optimization 3: building a cheap world

At this stage, one of the highest cost becomes that of creating the world (2.7 sec.) by procedure LIFE-Create-World.sql. The reason is the same as for the generation process: the SQLfast interpretation of the two embedded loops. There is several ways to decrease the building cost. We will examine three of them.

1. Executing a predefined script that creates the world

   Instead of generating on the fly and executing the `insert` queries as shown in Script 27.2, we store these queries in a standard SQLfast script file for further reuse. The cost of creating the world is that of executing this script,. This cost is much lower (about 0.05 s.) but this technique requires that we create one or several such predefined scripts.

2. Using a preloaded database

   The idea is similar: we create a collection of predefined databases in which worlds of various sizes have been preloaded. Same improvement and drawback as above.

3. Creating the world with a recursive query

   We keep Script 27.2, but we translate the double loop into a recursive CTE. The script is shown in Script 27.14. Outstanding result: the 60 X 60 world is created in **0.008** s. instead of **2.7**. And no cumbersome additional scripts or databases!

   This technique is quite elegant, but also is a bit less intuitive for users less familiar with recursive queries, and therefore deserves some explanations.

   The first recursive CTE, **INIT1**, creates rows (I,0,0) for I in [0,maxLin]. In other terms, it creates the cells of the *first column* of the maxLin X maxCol world.

   The second CTE, **INIT2**, starts from these cells to create the remaining cells of each line of the world.

```
with recursive
  INIT1(I,J,C)
  as (select 0,0,0
        union
      select I+1,J,0 from INIT1 where I < $maxLin$),
  INIT2(I,J,C)
  as (select * from INIT1
        union
      select I,J+1,0 from INIT2 where J < $maxCol$)
insert into CELL select * from INIT2;
```

Script 27.14 - Building a world through a recursive query

Considering the last technique, we get the result shown in Figure 27.7. The total execution time now drops to **4.18** sec., less than **1** sec. of which being spent by the execution of SQL queries. We can admit that this figure is quite satisfying.

procedure	raw	indexed	window	world
Create-Database	0.000	0.001	0.001	0.001
Create-World	2.7	2.7	**2.7**	**0.008**
Create-Glider	0.002	0.001	0.001	0.001
Compute-History	835.715	442.752	4.17	4.17
Total	**838.417**	**445.454**	**6.87**	**4.18**

Figure 27.7 - Creating a world through a recursive query

27.11 Optimization 4: let SQL generate vector graphics

Now, the highest cost is again that of procedure Compute-History, and more specifically procedure Generate-Current-State (run time 3.84 s.). Its code is that of Script 27.13. It comprises two embedded loops parsing the live cells of the current world window. For each of these cells, two elementary computations and the writing of their result as a short string.

This can easily be done by an SQL query. Let us convert the current script into pure SQL. We consider temporary table LINE in which we will insert the SQLdraw commands that draw the points of the successive generations of the automaton:

```
LINE(TXT)
```

The rewriting of Script 27.13 as an SQL query is shown in Script 27.15. The contents of table LINE is then copied in the SQLdraw file.[7]

The run time of this generation dramatically drops, from **3.84 s.** to 0.593 - 0.33 = **0.26 s.** (Figure 27.8).

```
insert into LINE
  select cast(6+I*12 as char)||','||cast(6+J*12 as char)||',""'
  from   CELL
  where  C = 1
  and    I between $I1$ and $I2$ and J between $J1$ and $J2$;
```

Script 27.15 - Rewriting Script 27.13 as a pure SQL query

7. This way of working could be disputed. Indeed, we postulate that the order of the rows of the result set will be that of the insertion queries. Since table LINE has no key nor index, most DBMS will implement it as a pure sequential structure, *first in, first out*. If we are uncomfortable with this hypothesis, we could add columns STEP, that represents the current step, and ORD, that specifies the position of the command in the SQLdraw file. When writing in this file, the TXT values are output in `order by STEP,ORD`. This will just add a dozen milliseconds to the total cost.

procedure	raw	indexed	window	world	graphics
Create-Database	0.000	0.001	0.001	0.001	0.001
Create-World	2.7	2.7	2.7	0.008	0.008
Create-Glider	0.002	0.001	0.001	0.001	0.001
Compute-History	835.715	442.752	4.17	4.17	0.593
Total	**838.417**	**445.454**	**6.87**	**4.18**	**0.603**

Figure 27.8 - Generating vector graphics with an SQL query

27.12 Optimization 5: Rewriting the application in Python

Of course, rewriting the procedures in a standard programming language, like Java or Python, should lead to still better times. Quite surprisingly, the improvement is more subtile than it could be thought. Indeed, the generation of the SQLdraw file through a Python program that has the same structure as the SQLfast application (including windowing) and that implements CELL and SCORE by lists of lists instead of database tables leads to a run time of **3.52** s., that is, nearly **6 times slower** than SQLfast version.[8] If we focus on the core computing, that is on the work of procedure **Compute-Next-Step.sql**, we observe that computing the 100 states by the Python program costs **2.48** s., to compare with **0.33** s. for the SQLfast application: the SQL engine appears to be **more than 7.5 times faster** than Python.

procedure	raw	indexed	window	world	graphics	Python
Create-Database	0.000	0.001	0.001	0.001	0.001	
Create-World	2.7	2.7	2.7	0.008	0.008	
Create-Glider	0.002	0.001	0.001	0.001	0.001	
Compute-History	835.715	442.752	4.17	4.17	0.593	
Total	**838.417**	**445.454**	**6.87**	**4.18**	**0.603**	**3.52**

Figure 27.9 - Generating drawing with an SQL query

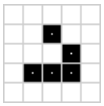
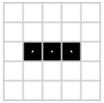
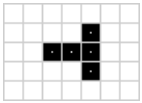
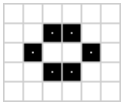
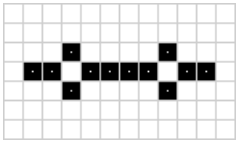
To be fair, this is just one experiment, on one small automaton, implemented in a dynamic language that is not known for its lightening speed (C or even Java could

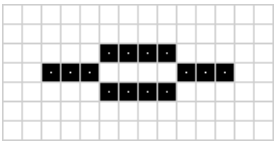
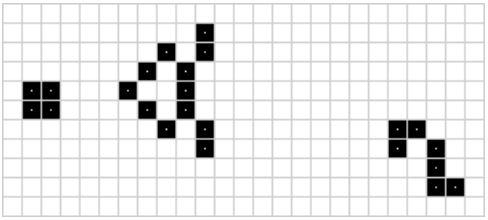
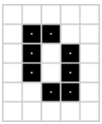
8. Program **LIFE-Game-in-Python.py** is included in the SQLfast distribution.

do better). However, it provides an interesting information on the ability of an SQL engine to efficiently perform matrix manipulation.

27.13 Some representative Conway's automata

The literature and the many dedicated web sites provide countless automata, the starting states of which range from three to several thousands live cells. For instance, try video clip http://www.youtube.com/watch?v=C2vgICfQawE for a dramatic presentation of outstanding patterns, some of them comprising hundreds of thousands of cells. We present here below some of the most popular cellular automata.

Starting state	Name and family
	**Elementary glider**. This automaton moves but does not grow. It has a period of 4 states.
	**Blinker**. The simplest oscillator with two distinct states.
	**Blinker2**. After a dozen evolution states, explodes into four simple Blinkers.
	**Beehive**. Still automaton. Each life cell has exactly two live neighbors.
	**Pentadecathlon**. Oscillator with a period of 15 states.

	**FlyingSaucer**. Transforms into six simple Blinker.
	**Buckaroo**. Oscillator with a period of 30 states.
	**Upside-Down**. Expands then shrinks into four still 2x2 squares.

27.14 Family life

The initial state of the *Gosper Gun* automaton comprises four entities: two bumpers (the left and right side 2x2 squares) and two generating entities (we will call them the *parents*) moving horizontally in opposite directions (Figure 27.10). Together, the parents form the so-called *Gun*. When a parent touches a bumper, it moves in the reverse direction, toward its partner. When the parents meet, they generate a little glider (as is natural for parents), after what they reverse their direction and target the bumpers.

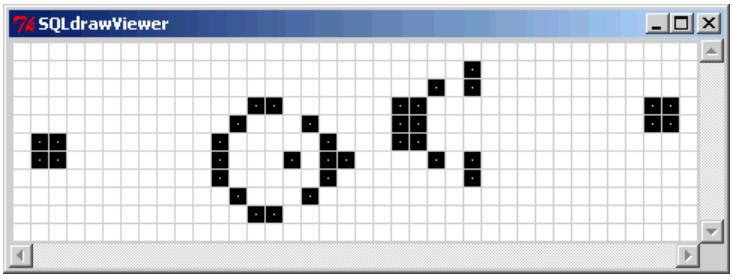


Figure 27.10 - Initial state of the Gosper Glider Gun

The *Gosper Gun* has a period of 30 states. This means that every 30 iterations, the couple of parents (the *Gun*) recovers its initial state (Figure 27.11) and has generated a new glider. The total time to compute and generate 90 states, that is three periods, is **2.6** s.

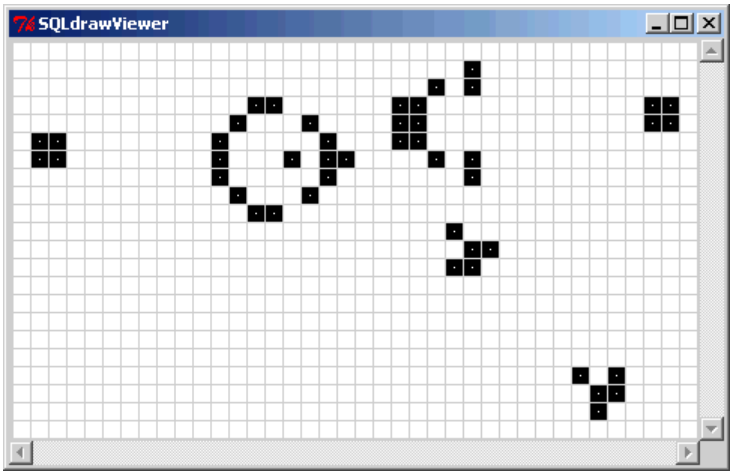


Figure 27.11 - The Gosper Gun at its 60th states. Two gliders have been generated.

27.15 Lessons learned

As it has been usual in most case studies developed so far, the very goal of this one is not to solve a specific problem, i.e., animating Conway's cellular automata,[9] but rather to illustrate general strategies of problem solving. One of the most interesting aspect of this study is the various ways a slow naive solution can be improved. Let us recall and comment the design decisions leading to the creation of worlds and automata and to the computing of their successive states.

1. **Using an *in-memory* database**. The main cost of data intensive applications is that of external memory access (such as reading from and writing to magnetic disk). Maintaining the data needed by the application in main memory allows keeping this cost as low as possible. Making the whole database reside in main memory is one of the simplest way to avoid this cost.[10] This is partic-

9. Let us be honest, solving this problem is fun but not terribly useful in the real life!
10. The other technique consists in assigning a large buffer to the database so that the data are read only once from the disk, then kept in the buffer, so that further access will find them in main memory, saving costly disk access. This technique requires less main memory space but may be less efficient than an *in-memory* database.

ularly effective when the database is transient (which is the case of our database) and fairly large.

2. **Dropping integrity constraints**. As a general rule, the critical properties of the data must be expressed as integrity constraints. *Uniqueness* constraints must be ensured by *primary* and *unique keys*, *referential* constraints must be translated into *foreign keys* and *mandatory* values into `not null` constraints. More complex properties can be expressed by `check` predicates or by `triggers`. This is quite true for general purpose databases, that are modified by several agents, from external sources, both likely to be unreliable. Of course, this security has a cost, the ROI being a high data quality. In some cases however, this cost can be avoided, notably when only one reliable process is in charge of updating the data. This (hopefully!) is the case of the application we have developed in this study. This is why we have declared no primary nor foreign keys and no `not null` constraints.
3. **Creating indexes**. An index, if carefully designed, reduces the cost of accessing a set of selected rows in a table (for reading or updating). And this, all the more so as the table is large and this set is a small part of this table. Though we have experimented on very small tables, we have observed a drastic improvement in execution time of the application.
4. **Reducing the problem space**. The first, naive, algorithms were built to explore, analyze and potentially update all the cells of the worlds. This process was useless for most of these cells. Identifying the smallest possible subset of cells likely to be usefully examined and modified has lead to an impressive gain of execution time. This identification is an essential part of the problem modeling process.
5. **From procedural to declarative expressions**. This last aspect may be one of the most important in the context of this series of case studies. Most programmers, be they professional or casual, tend to apply a procedural reasoning when they have to solve problems such as those of this case study. The main pattern of this reasoning could be informally translated as:

   for each element **`e`** of set **`E`**, execute action **`A`** on **`e`**.

   Hence the multiple loops that structure the algorithms of the naive version. Most such loops translate a higher level declarative reasoning that can be expressed into an SQL query, recursive[11] or not. These queries are preprocessed by the optimizer of the SQL engine and translated into an access plan quite often much more efficient than the procedural version. In our application, the gain was very dramatic.

Relational databases and the SQL language are often considered to be poorly fit to matrix manipulation. Surely, they are not the primary tools we can think about for

11. Many recursive procedures can be converted into loop-based structures. What we suggest here is to apply the converse and to let the SQL engine apply this conversion. This approach is natural to functional and logic programmers, but less familiar to mainstream developers.

such operations. However, this study shows that, in some cases, when they are carefully crafted, SQL applications can provide simple, intuitive and quite efficient solutions without requiring the use of special-purpose languages and environments.

Note: trying alternative algorithms

The technique applied in this study may be the most intuitive but it is not the only one that have been documented. The alternative approach described below is as simple but may be more efficient:

> *Instead of visiting all the neighbors of each cell, be it live or dead, as we did in this study, we examine* ***each live cell*** *and add 1 to the score of to all its neighbors. An index*[12] *on column C of CELL is likely to help a lot.*

Its implementation is fairly easy and is left as an exercise.

27.16 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Life_Game**. Actually, they can be run from main script **LifeGame-MAIN.sql**, that displays the selection box of Figure 27.12, through which the user can choose to compile and run a new game or to run a game previously compiled. We call *compiling a game* the transformation of the specification of this game (such as through Script 27.3) into executable code, that is, the corresponding SQLdraw file.

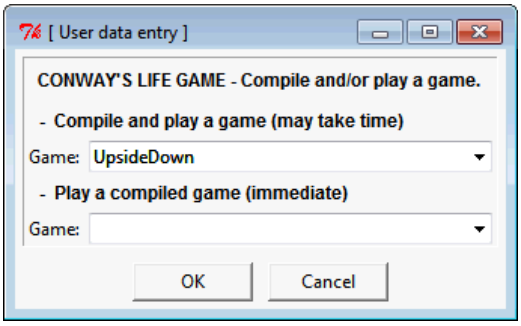


Figure 27.12 - Selecting a game to compile and run, or a script to run

12. In particularly a partial index that references live cells only (`where C = 1`).

The standard format of the scripts that create cellular automata (*games of life*) is shown in Script 27.16. It includes the `insert` queries that create the initial state in table CELL as well as the optimal settings of the following parameters:

- Lnbr: number of rows of the world shown in the graphical window
- Cnbr: number of columns of the world shown in the graphical window
- Niter: number of states to compute
- Delay: wait time between each state when the automaton is animated.

```
set Lnbr = 15;
set Cnbr = 15;
set Niter = 40;
set Delay = 150;

update CELL set C = 1 where I = 1 and J = 2;
update CELL set C = 1 where I = 2 and J = 3;
update CELL set C = 1 where I = 3 and J = 1;
update CELL set C = 1 where I = 3 and J = 2;
update CELL set C = 1 where I = 3 and J = 3;
```

Script 27.16 - Creation of the standard Glider (script Create-LIFEGAME-Glider.sql)

If the setting statements are missing or incomplete, a new data entry box opens, letting the user set the parameters at its preferred values (Figure 27.13).

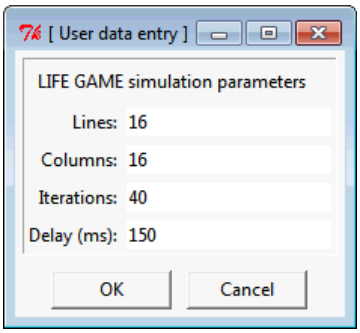


Figure 27.13 - Completing the parameters of the selected game

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

Case study **28**

From data bulk loading to database book writing

When data have to be loaded in a database from an external source, the order in which tables are filled is important as far as referential integrity is concerned. This order is determined by the directed graph formed by tables and foreign keys. From this graph one have to derive a linear ordering that represent one of the valid order in which table data are loaded. This derivation is called **topological sorting**, for which this chapter discusses and implements a simple algorithm. However, things are a bit more complex when the graph is not acyclic, as is often the case for database loading. Therefore, the chapter studies ways to transform a graph that includes circuits into a purely acyclic graph. These techniques are also applied to the ordering of topics when planning the writing of a book.

Key words: data loading, database schema, (non) acyclic graph, topological sorting, strongly connected components, graph contraction, condensation of a graph, transaction management

28.1 Introduction

We consider a set of objects among which a *partial order* relation holds. For any pair {A,B} of these objects, the relation tells that either A is before B, or B is before A, or there is no relationship between A and B.

A popular example is that of a *set of courses* and the *prerequisite* relation between them. We can draw a graph in which each node represents a course and the directed edges define the prerequisite relation: an edge from course C1 to course C2 indicates that C1 must be taken before C2. Practically, one must translate this graph into a programme, that is, a linear sequence of courses such that the *prerequisite* relation is preserved.

This translation is called **topological sorting**. It consists in deriving a strict linear order from the partial order. This technique encompasses a very general class of problems that appear frequently in computer science in general and in databases in particular. We will illustrate it through two representative examples: *database migration* and *book structuring*.

28.2 The database loading problem

We want to migrate data from a source database to a target database. The most intuitive way to proceed is to write an SQL script that translates the source data into `insert into` statements. This script is then executed against the target database. In which order must we write these `insert` statements in such a way that referential integrity is ensured *a priori*? For instance, in a script that loads external data rows in the ORDERS.db database, inserting data in the CUSTOMER table must precede the insertion of rows in table CUSTORDER. Figure 28.1 shows that inverting the *foreign key to primary key* relation produces the precedence relation in ORDERS schema.

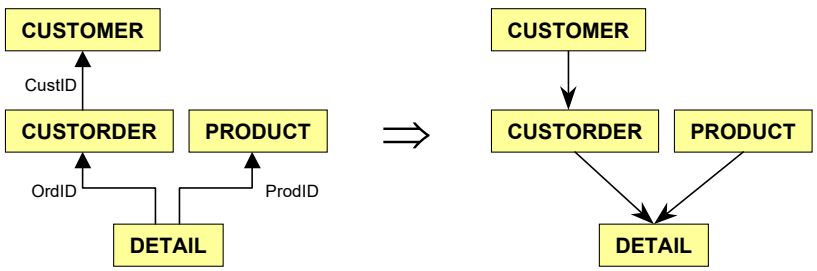


Figure 28.1 - The foreign keys graph defines the loading precedence relation among the tables [script GRAPH-Acyclic-ORDERS.sql]

With this respect, the following three data loading procedures fully satisfy referential integrity:

1. load CUSTOMER rows, then PRODUCT rows, then CUSTORDER rows and finally DETAIL rows,

2. load CUSTOMER rows, then CUSTORDER rows, then PRODUCT rows, and finally DETAIL rows,
3. load PRODUCT rows, then CUSTOMER rows, then CUSTORDER rows and finally DETAIL rows.

On the contrary, a sequence such as the following will trigger an error message for each DETAIL row:

4. load CUSTOMER rows, then CUSTORDER rows, DETAIL rows and finally PRODUCT rows.

The rules are quite intuitive: if table B references table A, then the rows of A must be loaded before those of B. Figure 28.2 depicts the derivation of the three valid loading sequences from the precedence relation of database ORDERS.db.

We observe that the precedence relation of the source schema also holds in each of these sequences. If we follow one of them, referential integrity will be ensured. Any other linear ordering would be invalid.

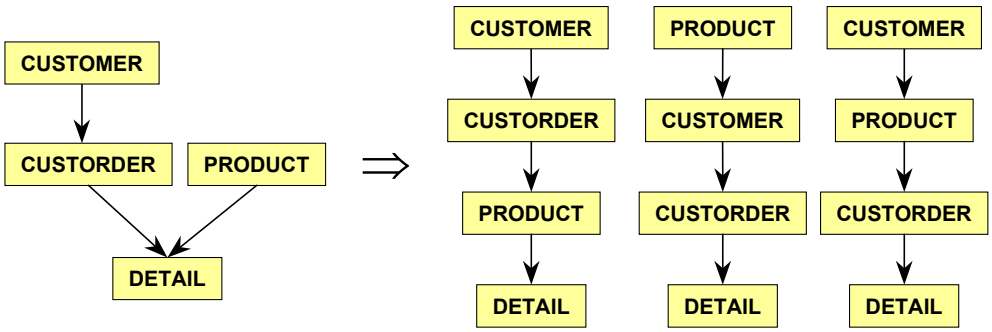


Figure 28.2 - The tables of database ORDERS.db and the three valid sequences in which their rows must be loaded

We have described a simple way to load data based on the relation between the tables in the database schema: we load **all** the CUSTOMER rows, then **all** the CUSTORDER rows, and so on. We could build other loading procedures based on relationships between **individual** data rows: rows may be loaded in any order provided that, for instance, CUSTOMER row C400 is loaded before CUSTORDER rows 30179, 30184 and 30186. However, we will ignore this technique in this chapter, since the former, called **bulk loading**, generally is both much faster and independent of the data.

Deriving a valid bulk loading sequence is an application of **topological sorting**. Considering an acyclic graph G, i.e., a graph with no circuits, a *topological sorting* of G is a *sequence* S comprising all the nodes of G, such that if node A precedes B in G, then it also precedes B in S.[1]

The schemas of Figure 28.1 are examples of acyclic graphs. Whatever path we define by following the arcs, we cannot visit a table twice. Even small schemas may

yield many valid linear sequences. For the purpose of correctly loading SQL data, only one of them will suffice.[2]

There are two main ways to generate linear sequences from an acyclic graph. The first one consists in visiting all the nodes of the precedence graph according to the *depth-first* algorithm, which has been described in Section 19.6 of the tutorial (*Tree traversal ordering*).[3] Since this procedure is frequently used in this book, we will experiment with the other one, which is particularly simple and intuitive, the *root extraction* algorithm.[4]

Considering the precedence graph of Figure 28.2 (right), we observe that there is no risk to first load either table CUSTOMER or table PRODUCT, since they both depend on no other table (they have no incoming arc). Let us call them the ***root*** tables of the schema. We arbitrarily choose to load table CUSTOMER for instance. We remove it from the schema (Figure 28.3, left). The resulting schema shows two root tables, CUSTORDER and PRODUCT. We chose (still arbitrarily) table PRODUCT, that we remove from the schema (Figure 28.3, center). The new schema has one root table, CUSTORDER. We remove it, which produces a schema with a single root table, DETAIL (Figure 28.3, right), which is removed. The schema is now empty, which terminates the procedure. We have extracted the root tables in the following order: CUSTOMER > PRODUCT > CUSTORDER > DETAIL, which is a valid loading sequence.

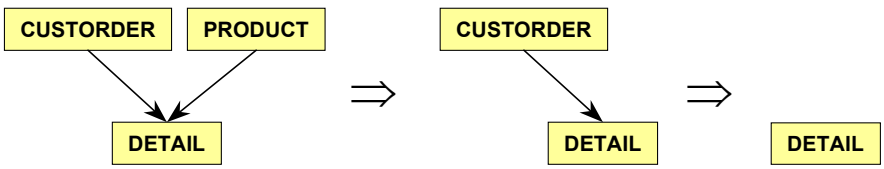


Figure 28.3 - Topological sorting through the root extraction algorithm

From this result, we can derive the statements of Script 28.1, which relies on a series of secondary scripts, each comprising the `insert` queries loading the rows of one table. The structure of this script is a faithful image of the topological ordering of the database schema.

1. See http://en.wikipedia.org/wiki/Topological_sorting (last visited August 2017) for example
2. Generating **all** valid linear sequences is a specific problem that is not addressed in this chapter. There are very few references proposing algorithms for this problem. See for instance "D. Knuth, J. L. Szwarcfiter, *A structured program to generate all topological sorting arrangements* (1974)", available at http://www.cs.ncl.ac.uk/publications/trs/papers/61.pdf (not checked) or http://ftp.cs.stanford. edu/pub/japan/topo.tex (not checked).
3. This technique is called *Tarjan's algorithm*.
4. Generally called *Kahn's algorithm*.

```
execSQL load_CUSTOMER.sql;
execSQL load_PRODUCT.sql;
execSQL load_CUSTORDER.sql;
execSQL load_DETAIL.sql;
```

Script 28.1 - Loading the data according to the referential constraints expressed in the schema

28.3 Representing the table graph of a database

To develop general procedures implementing topological sorting, we will use simple abstract data structures describing as intuitively as possible nodes and links between nodes.

They are shown in Figure 28.4, where their contents represent the table graph of database ORDERS.db. Table NODE (left) assigns an **id** to each node of the graph and records its name. Table LINK (right) assigns an **id** to each arc (or link) of the graph and specifies its source (FromNode) and destination (ToNode) nodes. In this example, each node represents a table and each link a foreign key from a source table (FromNode) to the target table (ToNode).

We have to note that the directed arcs these tables actually implement are the reference links of the foreign keys of the schema (Figure 28.1 - left) and not the precedence relation (Figure 28.1 - right). This choice represents the structure of the schema more naturally, though the term ***root*** to designate CUSTOMER and PRODUCT in this graph may be felt a bit mistaken!

NODE

NodeID	NodeName
N1	CUSTOMER
N2	PRODUCT
N3	CUSTORDER
N4	DETAIL

LINK

LinkID	FromNode	ToNode
5	N3	N1
6	N4	N2
7	N4	N3

Figure 28.4 - Tables NODE (left) and LINK (right) provide an synthetic view of the tables and foreign keys of database ORDERS.db

We will examine in section 28.18 how to insert the description of a database in these tables.

Since the concept of **root** is important in solving the sorting problem, we define view ROOT, that shows the *root nodes* of the graph, that is, nodes the id of which does not appear in column FromNode of LINK. We also create an `instead of` trigger allowing NODE rows to be deleted through this view (Script 31.3). Actually,

neither this view nor this trigger are necessary but they will make our algorithms more concise and more elegant.

```
create table NODE(NodeID   char(12) not null primary key,
                  NodeName varchar(64));
create table LINK(LinkID   integer not null primary key,
                  FromNode char(12) not null,
                  ToNode   char(12) not null,
                  foreign key (FromNode) references NODE
                          on delete cascade,
                  foreign key (ToNode) references NODE
                          on delete cascade);
create view ROOT
as select NodeID,NodeName from NODE
   where  NodeID not in (select FromNode from LINK);
create trigger Delete_ROOT
instead of delete on ROOT
for each row
begin
   delete from NODE where NodeID = old.NodeID;
end;
```

Script 28.2 - Creating tables NODE and LINK and view ROOT

28.4 Topological sorting of the table graph

The algorithm we have chosen for topological sorting requires iteratively eliminating the root nodes of the graph, until this graph eventually becomes empty.

This approach translates into Script 28.3. It comprises a simple loop, the body of which first counts the root nodes [1] and stops when table NODE becomes empty [2], then writes the loading statements for all the root nodes [3], and finally deletes these root nodes [4].

```
execSQL UTIL-SELECT-parameters-for-Lists.sql;
while (True);
   extract Nroot = select count(*) from ROOT;                [1]
      if ($Nroot$ = 0) exit;                                 [2]
   select 'execSQL load'||NodeName||'.sql;' from ROOT;       [3]
   delete from ROOT;                                         [4]
endwhile;
```

Script 28.3 - Generating a valid SQLfast script for the loading of database ORDERS.db

That's all. Just two elementary SQL queries plus the termination control statements!

To avoid the standard tabular formatting of the printed text, we choose the simple list format (`UTIL-SELECT-parameters-for-Lists`). The result for ORDERS.db database is shown in Script 28.1.

28.4.1 Topological sorting abstract graphs

By slightly modifying this generation script, we can transform it into a general purpose topological sorting procedure in which the nature of nodes and links is not specified. Script 28.4 produces table SORTED in which each node receives a level number.

```
create table SORTED(Level integer, NodeName varchar(64));
for Lev = [1,99];
   extract Nroot = select count(*) from ROOT;
      if ($Nroot$ = 0) exit;
   insert into SORTED select $Lev$,NodeName from ROOT;
   delete from ROOT;
endfor;
```

Script 28.4 - Topological sorting of an acyclic graph

Any extraction from this table in ascending order of the level number is a valid topological sequence (Figure 28.5).

```
+-------+-----------+    +-------+------------------+
| Level | NodeName  |    | Level | NodeNames        |
+-------+-----------+    +-------+------------------+
| 1     | CUSTOMER  |    | 1     | CUSTOMER,PRODUCT |
| 1     | PRODUCT   |    | 2     | CUSTORDER        |
| 2     | CUSTORDER |    | 3     | DETAIL           |
| 3     | DETAIL    |    +-------+------------------+
+-------+-----------+
```

Figure 28.5 - Two presentations of the topological sorting of the graph of database ORDERS.db

When applied to the schema of ORDERS.db, this procedure generates two of the three valid sequences only, but this is irrelevant since we are fully happy with any one of them.

A last remark: these procedures delete data from NODE and LINK tables. To keep source data unchanged in the process, we will process data from working table NODE_W and LINK_W in which the source data have been copied.

Figure 28.6 shows graphically the result of the application of Script 28.4 to a larger schema, comprising 10 tables and 11 foreign keys. The rows at a certain level can be loaded in any order.

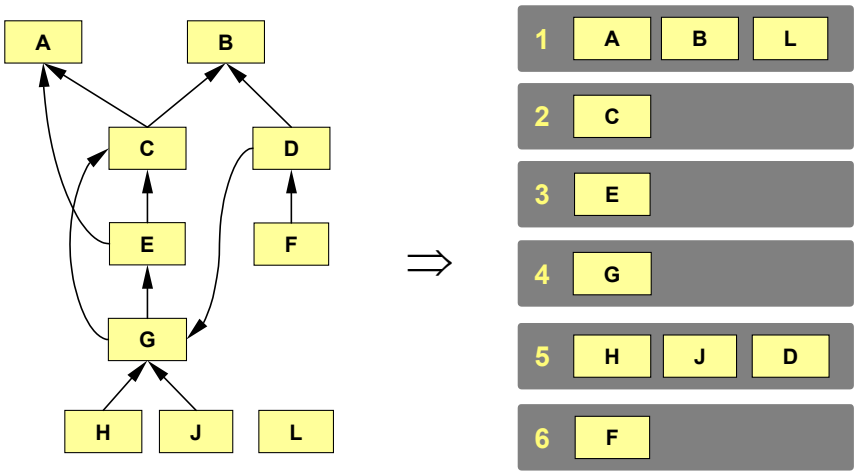


Figure 28.6 - Sorting a 10 table schema [script GRAPH-Acyclic-[10-11].sql]

28.5 Coping with non acyclic graphs

Topological sorting only applies to **acyclic** graphs, that is, graphs in which, if there is a path from node **n1** to node **n2**, then there is no path from **n2** to **n1**. In other words, an acyclic graph includes no **circuits**, that is, paths from a node to itself.

However, many database schemas do not satisfy this acyclicity condition.

28.5.1 Non acyclic graphs

This is where things are getting interesting. Figure 28.7 (left) represents a fragment of a database schema comprising tables DEPARTMENT and EMPLOYEE. Through foreign key (DepartName), each EMPLOYEE row specifies the (row of the) department by which the employee has been hired and each DEPARTMENT row specifies the (row of the) manager of this department, who happens to be an employee, through foreign key (Manager). This fragment forms a circuit.

Clearly, the graph of this schema has no root (Figure 28.7, right), so that the procedure developed so far does not apply. However, the original problem remains: how to load data in order to let them comply *a priori* with referential integrity constraints?

Such schemas raise two questions:

1. How can we check whether a schema is acyclic or not?
2. How to load data in a database the schema of which includes circuits?

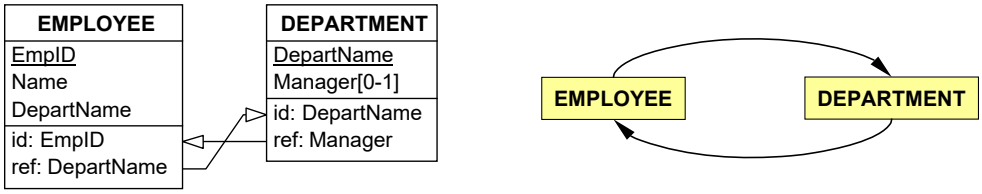


Figure 28.7 - An elementary cyclic schema

To experiment with cyclic schemas, we will work on the graph of Figure 28.8, deriving from that of Figure 28.6, in which we have added a link (a foreign key) from E to F that forms circuit E ⟶ F ⟶ D ⟶ G ⟶ E, or .E.F.D.G.E. for short[5].

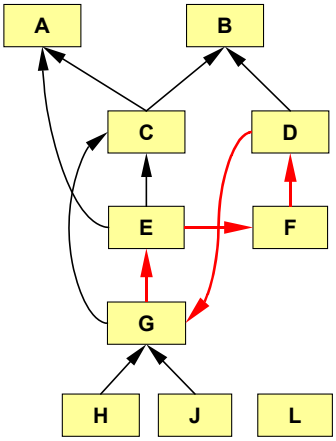


Figure 28.8 - A graph including a circuit [script GRAPH-Cyclic-[10-12].sql]

28.5.2 Checking the acyclicity of a graph

Let us apply the topological sorting procedure to the graph of Figure 28.8, even though we know it will be hopeless. After the first step, root nodes A, B and L are removed. The second step removes node C, leaving the graph *root-less* (Figure 28.9, left). At this point, we could stop the analysis. Indeed, if the resulting graph is not empty, then we know that it includes at least one circuit, and therefore is not acyclic.

5. When we build paths through SQL queries and SQLfast scripts, we use to add leading and trailing dots, so that this path will read ".E.F.D.G.E.". This form makes checking whether a node already belongs to a path easier.

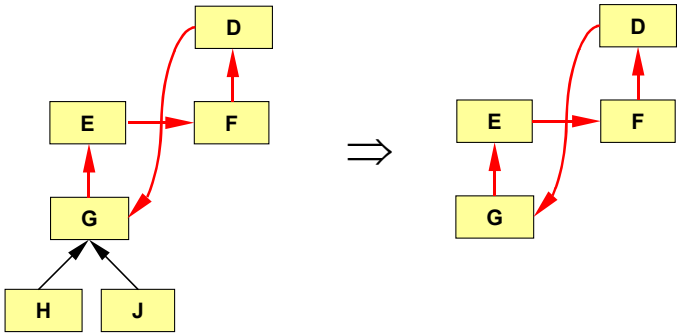


Figure 28.9 - Extracting the circuits by removing root (left) then leaf (right) nodes

28.5.3 The cyclic kernel of a graph

If we consider the left-side fragment of Figure 28.9 as a table graph, we observe that it also includes two *innocent* table nodes, H and J, that could be securely loaded, provided their target table, G, has been, by some magic trick, already loaded. We call them *leaf* nodes, by analogy with the *root* nodes.

Let us remove them by extending the sorting procedure. We define SQL view LEAF, together with its `delete` trigger. Let us extend Script 28.4 to iteratively remove not only *root* nodes but also *leaf* nodes (Script 28.6). The loop stops when there are no *root* nor *leaf* nodes any more (Figure 28.9, right). If the resulting graph is not empty, it comprises the tables that cannot be loaded through the standard procedure, therefore deserving a special treatment. Let us call ***cyclic kernel*** this set of nodes.

Though we do not know, until now at least, how to load the rows of tables of circuit D, G, E, F (noted .D.G.E.F.), we can sketch the general loading procedure as suggested in Script 28.5.

```
insert rows into tables A, B and L, in any order
insert rows into table C
insert rows into table circuit .D.G.E.F.
insert rows into tables H and J, in any order
```

Script 28.5 - Loading procedure of table graph of Figure 28.8

```
while (True);
   extract Nroot = select count(*) from ROOT;
      if ($Nroot$ = 0) exit;
   delete from ROOT;
endwhile;
while (True);
   extract Nleaf = select count(*) from LEAF;
      if ($Nleaf$ = 0) exit;
   delete from LEAF;
endwhile;
extract Nnode = select count(*) from NODE_W;
if ($Node$ = 0) write The schema is acyclic and is loadable;
if ($Node$ > 0) write The schema includes circuit(s) with
                      $Nnode$ tables. It cannot be loaded.;
```

Script 28.6 - Checking the loadability of a table graph by reducing it to its cyclic kernel

28.6 *Cyclic* vs *hard cyclic* kernel of a table graph

Strictly speaking, the graph theory about topological sorting falls short to help us to design secure sequences of table loading for non acyclic graphs. However, the graphs of database schemas are more subtile that plain *node-and-edge* mathematical graphs. Declaring a table graph *not loadable*, as we did in the previous section when it has been proved to include circuits, may be a bit hasty.

When we examine the foreign keys of Figure 28.7, we observe that they are of two kinds. We cannot insert an EMPLOYEE row if the row of the target department has not be loaded yet (because column DepartName is not nullable), but we can insert a DEPARTMENT row for which there is no manager (because column Manager is nullable). It appears that table EMPLOYEE depends on table DEPARTMENT while the latter does not depend on the former.

As a consequence, despite the existence of a circuit, we can load the rows of table DEPARTMENT without worrying about the referential constraint, then the rows of table EMPLOYEE. Afterwards, if and when the information is known, we update the DEPARTMENT rows to connect them to their manager rows (Script 28.7)

```
insert rows into table DEPARTMENT
insert rows into table EMPLOYEE
update rows of table EMPLOYEE
```

Script 28.7 - Optional links are translated into `update` statements

We will say that foreign key (DepartName) is *mandatory* and foreign key (Manager) is *optional*. Similarly, links in a table graph are either *mandatory* or *optional*.

So, we can redefine the loading problem as follows:

a database schema admits at least one secure loading sequence if its schema, reduced to its mandatory links, is acyclic.

To represent this property, we add column **Opt** to table LINK (and to its derivatives) (Script 28.8): Opt = 'Y' for optional links and 'N' otherwise. These values derive from column KeyNull of table SYS_KEY of the dictionary, as shown in Section 28.18.

Now, we analyze the table graph to check whether, by removing the optional links from the cyclic kernel, the circuits disappear. The procedure is given in Script 28.9, where statement [1] reduces the graph to its mandatory links.

We will call ***hard cyclic kernel*** the set of nodes obtained in this way. While a non empty cyclic kernel may not lead to loading problems, hard cyclic kernel definitely will do so.

```
create table LINK(LinkID   integer not null primary key,
                  FromNode char(12) not null,
                  ToNode   char(12) not null,
                  Opt      char(1),
                  foreign key (FromNode) references NODE
                          on delete cascade,
                  foreign key (ToNode) references NODE
                          on delete cascade);
```

Script 28.8 - Extending LINK table with optionality property **Opt**

```
delete from LINK_W where Opt = 'Y';                                    [1]
while (True);
   extract Nroot = select count(*) from ROOT;
      if ($Nroot$ = 0) exit;
   delete from ROOT;
endwhile;
while (True);
   extract Nleaf = select count(*) from LEAF;
      if ($Nleaf$ = 0) exit;
   delete from LEAF;
endwhile;
extract Nnode = select count(*) from NODE_W;
if ($Node$ = 0) write The schema DEFINITELY is acyclic;
if ($Node$ > 0) write The schema includes circuit(s) with
                      $Nnode$ tables.
```

Script 28.9 - Reducing the *cyclic kernel* of a graph to its *hard cyclic kernel*

28.7 Contracting the circuits of the hard kernel

When translating them into a valid data loading procedure, the circuits of the hard kernel require a special treatment we will examine later. For now, it suffices to isolate them in the graph and to consider the set of nodes of each of them as an independent virtual *super table* (a *super node* in general) made up of the components of the circuit. Mathematicians would say that the nodes of a circuit have been *contracted*. In this way, the cyclic graph becomes acyclic, so that it can be sorted as described above. Figure 28.10 shows the contraction of a circuit into a super table denoted by DEFG*.

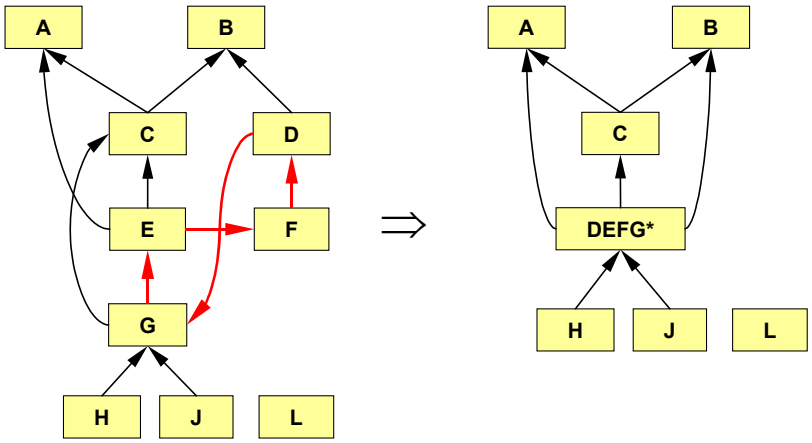


Figure 28.10 - Reducing circuits in a cyclic graph

28.8 Finding the independent circuits of a graph

We will show that the concept of cyclic kernel, be it hard or not, must be refined to design efficient loading scripts.

28.8.1 The independent circuits of a graph

A more careful examination of cyclic table graphs leads to the interesting observation that merely contracting the hard cyclic kernel may lead to too complex loading scripts.

Let us consider the graph (ABCDE) at the left of Figure 28.11, which is supposed to include mandatory links only. Clearly, it is a (hard) cyclic kernel since it has no *root* nor *leaf* nodes. However, it comprises two circuits (.A.B.) and (.D.E.) that can be contracted independently into super tables AB* and DE*, leading to a pure acyclic graph, as shown in the right part of this figure. Table C, which formerly belonged to

the hard cyclic kernel, can now be processed independently, as any table in an acyclic table graph.

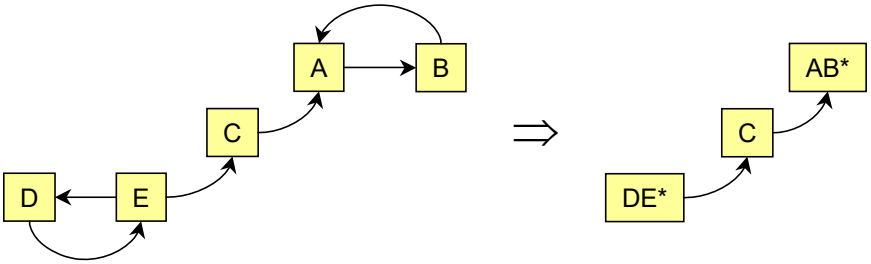


Figure 28.11 - The cyclic kernel may comprise several independent circuits

So, the abstract script derived from the cyclic kernel:

```
insert rows into table circuit .A.B.C.D.E.
```

can be developed into the more refined script:

```
insert rows into table circuit .A.B.
insert rows into table C
insert rows into table circuit .D.E.
```

It appears that the concept of *cyclic kernel* is not sufficient to meet our needs in that it provides too coarse loading solutions. We need to identify all the **independent circuits** in the hard cyclic kernel.

If extracting the hard cyclic kernel of a graph is quite easy, finding the independent circuits in a cyclic graph is not as straightforward as we could think at first glance.

Before developing an appropriate algorithm to find these independent circuits, let us have a more in-depth look at the very nature of circuits.

28.8.2 What is a circuit after all?

In general, a circuit in a directed graph is a path comprising distinct nodes, except the first and last ones which are the same. Figure 28.12 shows a series of five cyclic kernels. Below each of them, we suggest a contraction in which each circuit collapses into a super node. While the transformation of kernels (A) and (BC) is fairly straightforward, that of the other fragments is less obvious. Graph (DEF) for example includes two circuits, namely (.D.F.E.D.) and (.D.E.D.). Since these circuits share common nodes, they cannot be separated and must be treated as a single ***super node***.

Similarly, graph (GHJKL) also comprises 2 circuits that must be treated as a whole. On the contrary, graph (MNOPQR) can be split in two parts, (MNO) and (PQR), giving two super nodes connected by link (P,O).

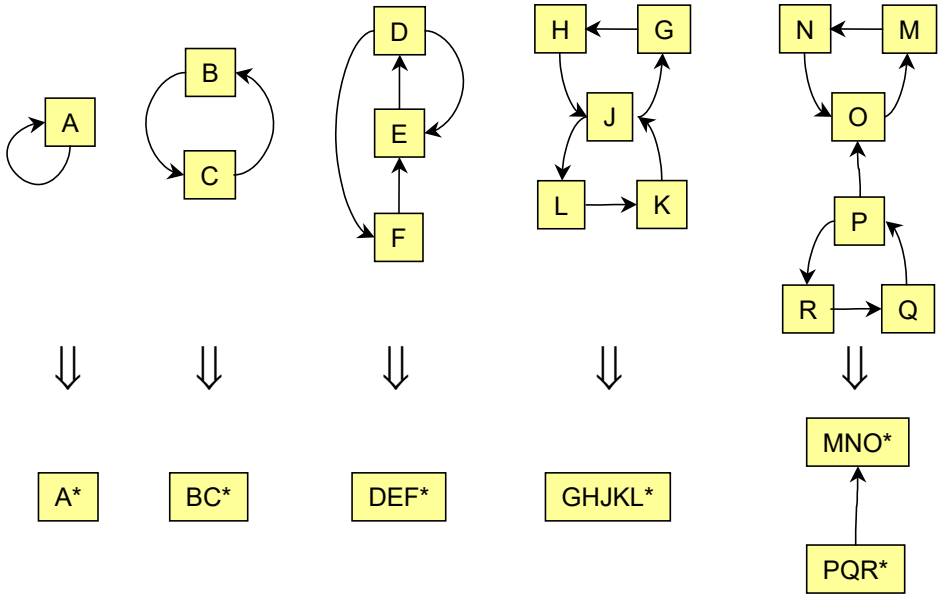


Figure 28.12 - The concept of *super circuit* and its contraction [script GRAPH-Cyclic-[17-20].sql]

To distinguish this special kind of circuits from the standard concept recalled above, we call them ***super circuits***.[6] We define them as follows:

A subset S of directed graph G is a ***super circuit*** *if,*

1. *for any pair (A,B) of not necessarily distinct nodes of S, there exists a path from A to B;*
2. *S is maximum, that is, whatever node A of S, there is no node C outside S such that A and C belongs to the same circuit.*

Let us spot two useful properties of super circuits:

– any node of a super circuit is reachable from any other node of this super circuit;

– each node belongs to at most one super circuit.

6. The technical name for *super circuits* is ***strongly connected component*** in graph theory. When all super circuits of graph G have been contracted, the resulting graph, which is acyclic, is called the *condensation* of G. [https://en.wikipedia.org/wiki/Strongly_connected_component, last visited August 2017]

28.9 Finding the super circuits of a graph

Now, we will develop an algorithm to identify the super circuits of a (hard) cyclic kernel. It is based on the closure of the hard cyclic kernel.

28.9.1 Computing the closure of a table graph

The **closure** of a directed graph G is the set of all the couples (n1,n2) such that there exists a path in G from node n1 to node n2, We compute the closure of the cyclic kernel and we store it in table CLOSURE (Script 28.10). The logic of the script is the same as that developed in case study *Kings of France - Part 2*, with some additional constraints due to the presence of circuits. In this script, we limit the generation of paths as follows:

– when a circuit has been found, it is no longer extended (`C.FromNode <> C.ToNode`),

– we prevent a new node to be appended to a path under construction if it is already present in the current state of this path, unless it appears in the first position, in which case this path becomes a circuit. Considering the right most fragment of Figure 28.12, path .P.R.Q. will be extended with node P, to produce circuit .P.R.Q.P., while path .P.O.M.N. will not be extended with O, since O is not the starting node of the path. This rule is translated into the following SQL condition:

```
C.Path not like '.%.'||L.ToNode||'%'
```

in which prefix `'.%.'` prevents the starting node of path `C.Path` to be compared with the current node `L.ToNode`.

```
create table CLOSURE(FromNode varchar(12),
                     ToNode   varchar(12),
                     Path     varchar(1024));
with CTE_CLOSURE(FromNode,ToNode,Path) as
(select select FromNode,ToNode,'.'||FromNode||'.'||ToNode||'.'
 from LINK_W
   union all
 select C.FromNode,L.ToNode,C.Path||L.ToNode||'.'
 from   CTE_CLOSURE C, LINK_W L
 where  C.ToNode = L.FromNode
 and    C.FromNode <> C.ToNode
 and    C.Path not like '.%.'||L.ToNode||'%')
insert into CLOSURE select * from CTE_CLOSURE;
```

Script 28.10 - Computing the closure of a graph

Applied to fragment DEF of Figure 28.12, that comprises two circuits forming, together, one super circuit, the script produces the result of Figure 28.13.

CLOSURE		
FromNode	**ToNode**	**Path**
D	E	.D.E.
D	D	.D.E.D.
D	F	.D.F.
D	E	.D.F.E.
D	D	.D.F.E.D.
E	D	.E.D.
E	E	.E.D.E.
E	F	.E.D.F.
E	E	.E.D.F.E.
F	E	.F.E.
F	D	.F.E.D.
F	F	.F.E.D.F.

Figure 28.13 - Table CLOSURE: the closure of fragment DEF

Note that this table includes duplicates since more than one distinct path may exist between two nodes. For example, couple (D,E) is defined by paths .D.E. and .D.F.E.. Actually table CLOSURE represents all the paths of the graph instead of its closure. We will discard duplicates on (FromNode,ToNode) later when needed.

28.9.2 Finding the *reachable set* of a node

The closure makes it easy to extract interesting information about the source graph: the *reachability* of its nodes. The *reachable set* of a node **n** is the set of nodes of the graph that are *reachable* from **n** through at least one path. This concept is not strictly required by topological sorting but it is an interesting by-product that can be used in similar contexts. This information is derivable from the closure with the simple query (computed for a node id in variable **n**):

```
select distinct ToNode from CLOSURE where FromNode = '$n$';
```

Script 28.11 is a more *professional* variant of this query. It provides, for the DEF fragment, the result of Figure 28.14, showing that DEF is a super circuit since all its nodes have the same reachable set.

```
select FromNode, group_concat(ToNode) as ToNodes
from  (select distinct FromNode,ToNode from CLOSURE)
group by FromNode;
```

Script 28.11 - Computing the reachable sets of a set of nodes

FromNode	ToNodes
D	E,F,D
E	D,E,F
F	E,D,F

Figure 28.14 - The reachable sets of the nodes of fragment DEF

28.9.3 How to denote the super circuits of a graph?

There is a minor but annoying problem in SQL: creating unique identifiers for sets. Of course, we could build a character string made up of the concatenation of the names of the members of the set.[7] For example, set {E,D,F} could be named 'E-D-F', provided '-' cannot be used in member names. This is fine for small sets but unworkable for thousands of members.

In addition, whether 'E-D-F' and 'F-E-D' denote the same set is not obvious from the comparison of these character strings. We could check this through an SQL query that compares two sets, but this operation is fairly time consuming (sets A and B are equal if each member of A is in B and both sets have the same size). Hence the idea to build set names from the ordered list of the names of their members: 'D-E-F' instead of 'E-D-F' or 'F-E-D'.

Actually, assigning unique names to super circuits is much simpler. Considering that a node may not belong to more than one super circuit, the name of any member of a super circuit can be used to denote the latter. For example, the data of Figure 28.14 can be extended with the unique name of the reachable sets (that happen to be super circuits), chosen as the **first member name** in alphanumeric order (Figure 28.15). This result is produced by Script 28.12. Note that there is no need to eliminate duplicates if we only want to compute the set names.

Figure 28.16 shows the result of the application of the script to the set of fragments of Figure 28.12.

FromNode	ToNodes	SetID
D	E,F,D	D
E	D,E,F	D
F	E,D,F	D

Figure 28.15 - Assigning unique names to (disjoint) reachable sets of the nodes of fragment DEF

7. We could represent sets by bit strings, where each bit indicates the presence of a member. However, this technique is far from the database culture!

```
select FromNode, group_concat(ToNode) as ToNodes,
       min(ToNode) as SetID
from  (select distinct FromNode,ToNode from CLOSURE)
group by FromNode;
```

Script 28.12 - Computing the reachable sets of a set of nodes and assigning them unique identifiers

FromNode	ToNodes	SetID
A	A	A
B	C,B	B
C	B,C	B
D	E,F,D	D
E	D,E,F	D
F	E,D,F	D
G	H,J,G,L,K	G
H	J,G,L,H,K	G
J	G,L,H,K,J	G
K	J,G,L,H,K	G
L	K,J,G,L,H	G
M	N,O,M	M
N	O,M,N	M
O	M,N,O	M
P	O,R,M,Q,N,P	M
Q	P,O,R,M,Q,N	M
R	Q,P,O,R,M,N	M

Figure 28.16 - The reachable sets of the graph of Figure 28.12 with unique identifiers

28.9.4 Identifying the super circuits of a graph

Let us consider another definition of super circuits in a graph G:

A super circuit is a maximum subset S of G such that S is a part of the reachable set of each of its members.

This suggests an intuitive algorithm to compute a super circuit once we have identified one of its member. Let us consider node **n** that is known to belong to a circuit of G (actually any member of the hard cyclic kernel of the graph, but this is irrelevant here). Let R be the reachable set of **n**. The super circuit to which **n** belongs, if any, is the set of all the members **r** of R such that (**r**,**n**) belongs to the closure of G.

Script 28.13 extracts, for each node of a graph, its super circuit and assigns it a unique identifier. Its application of the graph of Figure 28.12 is shown in Figure 28.17.

The particular form of the `from` clause provides a sorted list of distinct (FromNode,ToNode) couples required by the `group_concat` function.

```
create table SUPER_CIRCUIT(Node         varchar(12),
                           SuperCircuit varchar(1024),
                           CircuitID    varchar(12))
insert into SUPER_CIRCUIT(Node,SuperCircuit,CircuitID)
select C.FromNode,group_concat2(C.ToNode,1,C.ToNode,1,',')[a],
      '['||min(C.ToNode)||']'
from   CLOSURE C
where  exists (select 1 from CLOSURE CI
               where  CI.ToNode = C.FromNode
               and    C.ToNode = CI.FromNode)
group by C.FromNode;
```

a. `group_concat2` is a UDF that simulates aggregate function `group_concat` of MySQL. See SQLfast documentation.

Script 28.13 - Computing the super circuit of each node of a graph

SUPER_CIRCUIT		
Node	**SuperCircuit**	**CircuitID**
A	A	[A]
B	B,C	[B]
C	B,C	[B]
D	D,E,F	[D]
E	D,E,F	[D]
F	D,E,F	[D]
G	G,H,J,K,L	[G]
H	G,H,J,K,L	[G]
J	G,H,J,K,L	[G]
K	G,H,J,K,L	[G]
L	G,H,J,K,L	[G]
M	M,N,O	[M]
N	M,N,O	[M]
O	M,N,O	[M]
P	P,Q,R	[P]
Q	P,Q,R	[P]
R	P,Q,R	[P]

Figure 28.17 - Table SUPER_CIRCUIT: the super circuit of each node of the graph of Figure 28.12

The derivation of the set of super circuits and their composition is straightforward, as shown by Script 28.14. Figure 28.18 gives the list of super circuits of the graph of Figure 28.12.

```
select CircuitID as SuperCircuit,
       group_concat(Node,',') as Composition
from   SUPER_CIRCUIT
group by CircuitID;

select distinct CircuitID as SuperCircuit,
                SuperCircuit as Composition
from   SUPER_CIRCUIT;
```

Script 28.14 - Two equivalent queries to compute the super circuits of a hard cyclic kernel

SuperCircuit	Composition
[A]	A
[B]	B,C
[D]	D,E,F
[G]	G,H,J,K,L
[M]	M,N,O
[P]	P,Q,R

Figure 28.18 - The super circuits of the graph of Figure 28.12

28.9.5 Structure of the solution

We know that the condensation of an arbitrary source graph is acyclic. Therefore, when all super circuits of the source have been identified and contracted, all the links internal to super circuits have been removed and all the links in which one member of a super circuit is involved have been updated, we obtain an acyclic graph that can be topologically sorted. We suggest to store this resulting acyclic graph in tables NODE_A and LINK_A (Script 28.15).

```
create table NODE_A(NodeID   char(12),
                    NodeName varchar(64));

create table LINK_A(LinkID   integer,
                    FromNode char(12),
                    ToNode   char(12),
                    Opt      char(1));
```

Script 28.15 - Tables NODE_A and LINK_A contains the final condensation of the source schema

28.9.6 Revisiting the optional links

Script 28.9 reduces the cyclic kernel into the hard cyclic kernel by removing remaining optional links (statement [1]). It is time to wonder what to do with them, instead of just throwing them away. Since they will be translated into update operations, we suggest to store them in a special table, called LINK_UPDATE, from which we will generate appropriate SQL update statements (Script 28.16).

```
create table LINK_UPDATE(LinkID   integer,
                         FromNode char(12),
                         ToNode   char(12));
```

Script 28.16 - Table LINK_UPDATE contains deleted optional links

However, there is room for optimization with these updates. Let us examine the schema of Figure 28.11 in which link (C,A) is made optional. With our procedure, this link will be moved to table LINK_UPDATE to be translated into `update C` statements executed after `insert` statements.

Actually, this is useless since the foreign key columns can be safely set during `insert into C` statements, be they mandatory or optional. For the moment, we leave them in this table, but, later, we will have to move them back to the `LINK_A` result table.

28.9.7 Intermediate summary

It is time to summarize what we have got so far:

- Tables NODE and LINK contain the description of the source schema.
- Tables NODE_W and LINK_W contain the nodes and links of the hard cyclic kernel. They are empty if the source schema, minus the optional link of the cyclic kernel, is acyclic.
- Table CLOSURE contains all the paths starting from each node of the hard cyclic kernel.
- Table SUPER_CIRCUIT provides, for each node, the super circuit it belongs to, with its composition and its unique identifier.
- Table NODE_A *will* contain the nodes of the final acyclic graph. It is still empty.
- Table LINK_A *will* contain the link of the final acyclic graph. It is still empty.
- Table LINK_UPDATE contains the optional links of the cyclic kernel. Some of them may be misplaced and should be moved to LINK_A.

28.10 Building the final acyclic graph

To build the final acyclic graph, we must distinguish three cases.

1. If the source graph **is acyclic**, the final graph is the exact copy of the source graph.
2. If the source graph **is not acyclic**, but its subset comprising mandatory links only **is acyclic**, then the final graph is an exact copy of the source graph, minus the links that already have been moved to LINK_UPDATE.
3. If the source graph **cannot be reduced to an acyclic graph**, then:
   - all the super nodes identified in SUPER_CIRCUIT are moved to NODE_A,
   - all the source nodes that do not belong to super circuits are moved to NODE_A,
   - the links from LINK connecting nodes that do not belong to a super circuit are moved to LINK_A,
   - the links from LINK connecting nodes one or both of which belong to a super circuit are moved to LINK_A, after the id of these nodes have being replaced by that of their super circuit.

Let us examine the third case.

Filling table NODE_A is easy: the super nodes are drawn from CircuitID of SUPER_CIRCUIT (Figure 28.17) and the ordinary nodes are those from NODE that are not in Node of SUPER_CIRCUIT (Script 28.17).

```
insert into NODE_A
select NodeID,NodeName from NODE
where  NodeID not in (select Node from SUPER_CIRCUIT);
insert into NODE_A
select distinct CircuitID,CircuitID from SUPER_CIRCUIT;
```

Script 28.17 - Loading the nodes of the final acyclic graph

Filling table LINK_A is a bit more tricky. To simplify the procedure, we create a new table, TRANSLATION, that translates the id of each source node as follows: if a node belongs to a super circuit, its id is replaced by the id of this circuit, otherwise, it keeps its id. Script 28.18 produces the translation mapping of Figure 28.19 for the source schema of Figure 28.10. Column Super indicates whether the source node is a super node or not.

The final links are then inserted into LINK_A from source links in LINK, in which the nodes ids have been converted through table TRANSLATION (Script 28.19).

```
create table TRANSLATION(Source     varchar(12),
                         Substitute varchar(12),
                         Super      char(1));
insert into TRANSLATION
select NodeID,NodeID,'N' from NODE
where  NodeID not in (select Node from SUPER_CIRCUIT)
  union
select Node,CircuitID,'Y' from SUPER_CIRCUIT;
```

Script 28.18 - Table TRANSLATION mapping source node ids into their super circuit ids.

TRANSLATION		
Source	**Substitute**	**Super**
A	A	N
B	B	N
C	C	N
D	[D]	Y
E	[D]	Y
F	[D]	Y
G	[D]	Y
H	H	N
J	J	N
L	L	N

Figure 28.19 - Translation table for the schema of Figure 28.10

```
insert into LINK_A
select distinct L.LinkID,T1.Substitute,T2.Substitute,L.Opt
from   TRANSLATION T1, LINK L, TRANSLATION T2
where  L.FromNode = T1.Source and L.ToNode = T2.Source
and    T1.Substitute <> T2.Substitute
and    L.LinkID not in (select LinkID from LINK_UPDATE);
```

Script 28.19 - Loading the links of the final acyclic graph

28.11 Optimizing the final acyclic graph

When we convert the cyclic kernel into the hard cyclic kernel, we remove all the optional links to check the acyclicity of the result and to identify, if any, independent super circuits. The result is an acyclic graph that will be converted, once topologically sorted, into a sequence of SQL **insert** statements, plus a series of optional

links that will be converted into a sequence of SQL **`update`** statements, as shown in Script 28.7.

As far as operation cost is concerned, this may raise a performance problem. When a row is the source of an optional link stored in LINK_UPDATE, it will be **inserted**, then, afterwards, **updated** to create this optional link. On the contrary, all the other optional links that have not been discarded are created at insertion time, therefore saving the cost of accessing again the inserted rows to update them.

So, it would be useful to check the links of table LINK_UPDATE to examine whether they actually require additional update operations. Figure 28.20 illustrates the point. The left side fragment is a part of a cyclic kernel. It comprises three optional links (in dotted lines). These links are removed when we produce the hard cyclic kernel and moved in table LINK_UPDATE (center fragment). Actually, any two of them could be restored in the acyclic graph, thereby avoiding their update cost (right side fragment). The more links we can move back, the lowest the global loading time.

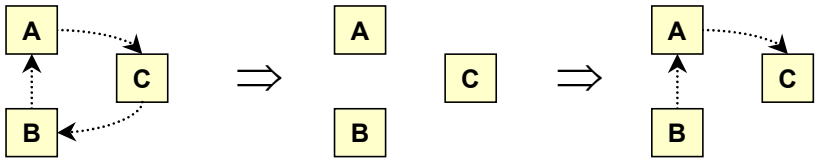


Figure 28.20 - Restoring optional links of the cyclic kernel (left) into the final acyclic graph (right)

The optimization problem can be formulated as follows: *which of the optional links stored in table LINK_UPDATE can be moved back in the acyclic schema without making the latter loose its acyclicity property?*

Unfortunately, there is no clear structural property that could allow us to select the links to move. Therefore, we will adopt a *brute force* method consisting in moving the optional links, one at a time, and keeping those that preserve the acyclicity property of the graph.

The work of acyclicity checking is done in working tables NODE_W and LINK_W. The *innocent* links from LINK_UPDATE (those which leave LINK_A acyclic) are removed and inserted into RESTORED_LINK. The complete algorithms is shown below and its translation in SQLfast is proposed in Script 28.20.

```
table RESTORED_LINK is initially empty
for each row L from LINK_UPDATE]:
   copy NODE_A rows into NODE_W
   copy LINK_A rows into LINK_W
   add (translated) RESTORE_LINK to LINK_W
   add (translated) L to LINK_W
   if graph NODE_W,LINK_W is acyclic:
      insert L into RESTORED_LINK
delete from LINK_UPDATE the rows copied in RESTORED_LINK
insert into LINK_A the (translated) rows of RESTORED_LINK
```

The distribution of optional links between tables LINK_A and LINK_UPDATE depends on the order in which they are examined. Therefore, several loading patterns may be derived from table LINK_UPDATE.

```
for lid,fro,to,opt = [select * from LINK_UPDATE];
   delete from NODE_W;
   insert into NODE_W select * from NODE_A;
   insert into LINK_W select * from LINK_A;
   insert into LINK_W
      select L.LinkID,T1.Substitute,T2.Substitute,L.Opt
      from   TRANSLATION T1, RESTORED_LINK L, TRANSLATION T2
      where  L.FromNode = T1.Source and L.ToNode = T2.Source
      and    T1.Substitute <> T2.Substitute;
   insert into LINK_W
      select $lid$,T1.Substitute,T2.Substitute,'Y'
      from   TRANSLATION T1, LINK_UPDATE L, TRANSLATION T2
      where  L.FromNode = T1.Source and L.ToNode = T2.Source
      and    L.LinkID = $lid$;
   execSQL Remove-Roots.sql;
   extract Nnode = select count(*) from NODE_W;
   if ($Nnode$ = 0);
      insert into RESTORED_LINK
         values($lid$,'$fro$','$to$','$opt$');
      delete from LINK_UPDATE where LinkID = $lid$;
   endif;
endfor;
delete from LINK_UPDATE
where  LinkID in (select LinkID from RESTORED_LINK);
insert into LINK_A
   select L.LinkID,T1.Substitute,T2.Substitute, L.Opt
   from   TRANSLATION T1, RESTORED_LINK L, TRANSLATION T2
   where  L.FromNode = T1.Source and L.ToNode = T2.Source
   and    T1.Substitute <> T2.Substitute;
```

Script 28.20 - Restoring optional links that can be created through **insert** statements

28.12 Loading data: a complex abstract example

We have developed the building blocks allowing us to solve the problem of generating a valid data loading script for any relational schema, with some touch of optimization. Let us now apply them to a more complex example, including several super circuits and optional links, the schema of which is depicted in Figure 28.21. The yellow-colored nodes are members of circuits while dark yellow ones are, in addition, members of the hard cyclic kernel. Solid arcs represent mandatory links while dotted arcs represent optional links.

The elimination of root and leaf nodes provides the *cyclic kernel* depicted in Figure 28.22. Discarding optional links, then removing again root and leaf nodes lead to the *hard cyclic kernel* of Figure 28.23.

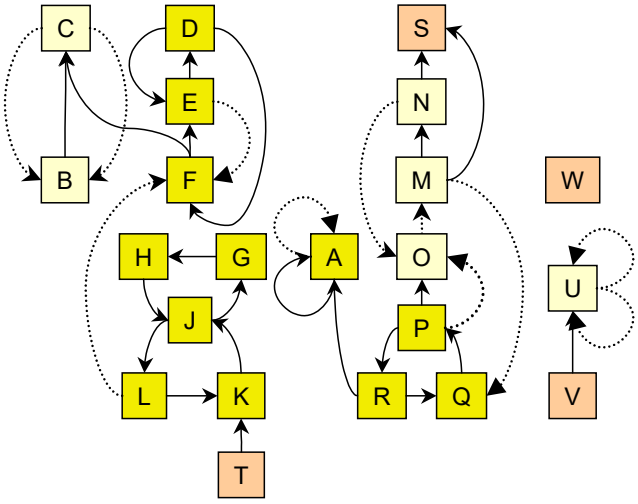


Figure 28.21 - A complex source schema [script GRAPH-Cyclic-[22-23-11].sql]

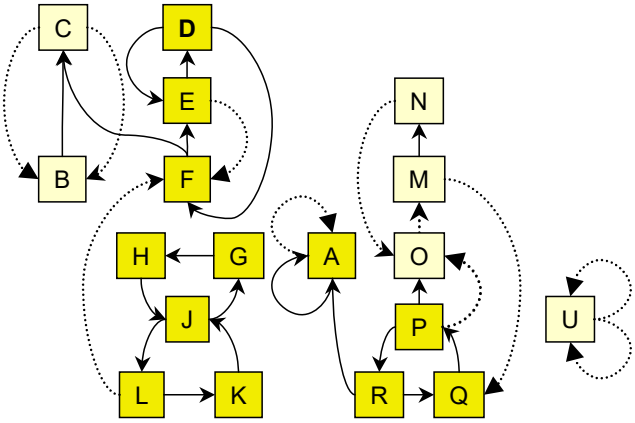


Figure 28.22 - The cyclic kernel

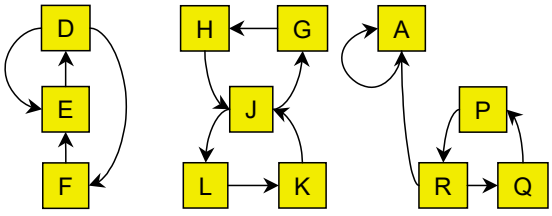


Figure 28.23 - The hard cyclic kernel

From the hard cyclic kernel, we identify four super circuits. They are derived from the contents of table SUPER_CIRCUIT (Figure 28.24) and presented in Figure 28.25).

SUPER_CIRCUIT		
Node	**SuperCircuit**	**CircuitID**
A	A	[A]
D	D,E,F	[D]
E	D,E,F	[D]
F	D,E,F	[D]
G	G,H,J,K,L	[G]
H	G,H,J,K,L	[G]
J	G,H,J,K,L	[G]
K	G,H,J,K,L	[G]
L	G,H,J,K,L	[G]
P	P,Q,R	[P]
Q	P,Q,R	[P]
R	P,Q,R	[P]

Figure 28.24 - Table SUPER_CIRCUIT

SuperCircuit	Composition
[A]	A
[D]	D,E,F
[G]	G,H,J,K,L
[P]	P,Q,R

Figure 28.25 - Super circuits and their composition

Each super circuit is contracted into a super node (Figure 28.26). Figure 28.27 is the condensation of the source schema, in which super circuits have been replaced by super nodes. In addition, all the optional links that preserve acyclicity of the final graph have been added (dotted, blue arcs).

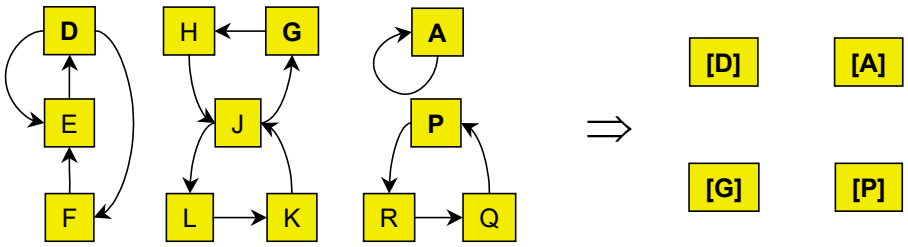


Figure 28.26 - The super circuits (left) are contracted into super nodes (right)

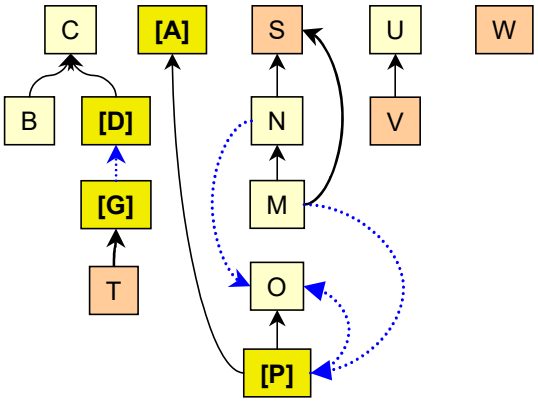


Figure 28.27 - The final acyclic graph, comprising both mandatory (solid, black) and optional (dotted, blue) links

Figure 28.28 represents the contents of tables LINK_A and LINK_UPDATE, which show the distribution of optional links. Four optional links from the cyclic kernel can be created through the `insert` statements, hence much more efficiently than through additional `update` statements. Links described in LINK_UPDATE must be created through `update` statements.

LINK_A

LinkID	FromNode	ToNode	Opt
3	B	C	N
11	[D]	C	N
19	T	[G]	N
20	M	N	N
21	M	S	N
24	N	S	N
27	[P]	O	N
30	[P]	[A]	N
34	V	U	N
17	[G]	[D]	Y
22	M	[P]	Y
23	N	O	Y
28	[P]	O	Y

LINK_UPDATE

LinkID	FromNode	ToNode	Opt
2	A	A	Y
4	C	B	Y
5	C	B	Y
9	E	F	Y
25	O	M	Y
32	U	U	Y
33	U	U	Y

Figure 28.28 - Links of table LINK_A can be created through **insert** statements while those of table LINK_UPDATE must be created through **update** statements

The final schema can then be sorted through the usual topological sorting algorithm, which provides the 4-level sequence depicted in Figure 28.29.

From this sorted sequence and the contents of table LINK_UPDATE, we derive a valid loading script such as that of Script 28.21.

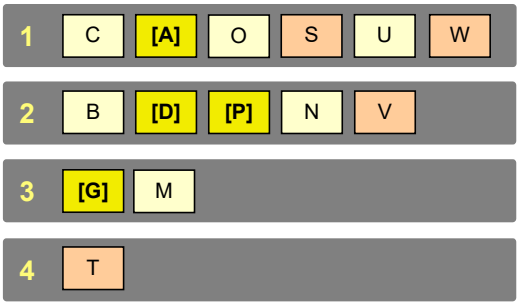


Figure 28.29 - A topological sorting of the final acyclic schema

```
execSQL insert_C.sql;
execSQL insert_[A].sql;
execSQL insert_O.sql;
execSQL insert_S.sql;
execSQL insert_U.sql;
execSQL insert_W.sql;

execSQL insert_B.sql;
execSQL insert_[D].sql;
execSQL insert_[P].sql;
execSQL insert_N.sql;
execSQL insert_V.sql;

execSQL insert_[G].sql;
execSQL insert_M.sql;

execSQL insert_T.sql;

execSQL update_A.sql
execSQL update_C.sql
execSQL update_E.sql
execSQL update_O.sql
execSQL update_U.sql
```

Script 28.21 - A valid data loading script for the schema of Figure 28.21

28.13 The complete data loading generation procedure

The sorting application comprises a main script (Script 28.22) and a series of secondary scripts. It works as follows.

– The script asks the user to select a graph, that is, a script file with name "GRAPH-xxx", such as "GRAPH-Acyclic-[10-11].sql" [1].

– The sorting database is created [2], together with its tables [3] then the selected source graph is loaded in tables NODE and LINK [4].
– The source graph is copied in working tables NODE_W and LINK_W [5].
– The root and leaf nodes are deleted to check the acyclicity of the source graph [6]. Tables NODE_W and LINK_W now contain the *cyclic kernel*.
– If the cyclic kernel is not empty [7], optional links are copied into table LINK_UPDATE [8], then deleted from NODE_W [9]. Then, root and leaf nodes are again deleted [10]. Tables NODE_W and LINK_W now contain the *hard cyclic kernel*.
– If the hard cyclic kernel **is empty** [11], the source graph is copied in tables NODE_A and LINK_A (minus the links contained in LINK_UPDATE) [12]. NODE_A and LINK_A contain an acyclic subset of the source graph and LINK_UPDATE contains the remaining optional links.
– If the hard cyclic kernel **is not empty** [13], the script computes and contracts the super circuits. First it computes the closure of this kernel in table CLOSURE [14], then it computes the super circuits in table SUPER_CIRCUIT [15] and creates the node id translation table TRANSLATION [16]. Table NODE_A and LINK_A are filled with source nodes that are not in super circuits and with all super nodes, while table LINK_A is filled with source links in which node ids have been converted through table TRANSLATION [17]. Finally, harmless optional links from LINK_UPDATE are moved to LINK_A [18].
– Table NODE_A is topologically sorted, then a valid loading script, including `insert` operations and `update` operations is generated [19].

28.14 Loading the data of a super circuit

Loading data into tables that form a super circuit can be performed in several ways. We mention two of them.

1. **Disabling referential constraint validation**. The simplest one consists in disabling referential constraints during the loading process. Rows are inserted without any integrity validation, which makes the process particularly fast. If the source data are of high quality, for instance if they have been extracted from a consistent database, then they are guaranteed to satisfy *a priori* referential integrity, so that no checking may be required. If there is some doubt about the quality of the source data, then an *a posteriory* checking may be performed, to identified incorrect date, through a query such as the following:

```
select OrdID from CUSTORDER
where  CustID not in (select CustID from CUSTOMER);
```

```
execSQL __Select-a-Graph.sql;                                          [1]
   if ($continue$ = 0) return;
createOrReplaceDB TOPO.db;                                             [2]
execSQL __Create-Data-Structures.sql;                                  [3]
execSQL $graphFile$;                                                   [4]
insert into NODE_W select * from NODE;                                 [5]
insert into LINK_W select * from LINK;
execSQL __Remove-Roots-Leaves.sql;                                     [6]
extract Nnode = select count(*) from NODE_W;
if ($Nnode$ > 0);                                                      [7]
   insert into LINK_UPDATE                                             [8]
       select * from LINK_W where Opt = 'Y';
   delete from LINK_W where Opt = 'Y';                                 [9]
   execSQL __Remove-Roots-Leaves.sql;                                 [10]
endif;
extract Nnode = select count(*) from NODE_W;
if ($Nnode$ = 0);                                                      [11]
   insert into NODE_A select * from NODE;                              [12]
   insert into LINK_A select * from LINK
                     where  LinkID not in (select LinkID
                                           from LINK_UPDATE);
else;                                                                  [13]
   execSQL __Compute-CLOSURE.sql;                                      [14]
   execSQL __Compute-SUPER_CIRCUIT.sql;                                [15]
   execSQL __Compute-TRANSLATION.sql;                                  [16]
   execSQL __Move-to-Acyclic-Tables.sql;                               [17]
   execSQL __Restore-Optional-Links.sql;                               [18]
endif;
execSQL __Generate-Loading-Script.sql;                                 [19]
closeDB;
```

Script 28.22 - Producing the final acyclic solution into tables NODE_A and LINK_A [scripts _Topological_Sorting.sql and _Topological_Sorting_(with-trace).sql]

2. **Deferring referential constraint validation**. Another approach consists in encapsulating the loading of data of these tables into a global transaction in which referential checking is deferred until commit time. Rows can be inserted in arbitrary order, providing the data eventually satisfy referential constraints at transaction closure. This technique is of course DBMS dependent. In the version of SQLfast based on SQLite, the script will be written as follows:

```
generic PRAGMA defer_foreign_keys = ON;
<insert rows in any order>
commitDB;
```

28.15 Loading data: a real example

The schema of the main database of a French mail order company has been analyzed by the procedures developed in this chapter. This schema has been generated from the conceptual schema of a legacy DATACOM/DB database recovered through a reverse engineering process.

The schema is made up of 691 tables, 5,449 columns, 683 primary keys, 81 unique keys and 1,190 foreign keys, 98 of which are optional. It includes 207 root tables and 362 leaf tables.

The schema includes circuits. The cyclic kernel comprises 16 tables and 28 foreign keys. When the optional foreign keys are removed and the remaining roots and leaves are deleted, the hard cyclic kernel reduces to 8 tables and 11 mandatory foreign keys. This kernel is made up of three independent aggregates, shown in Figure 28.30.

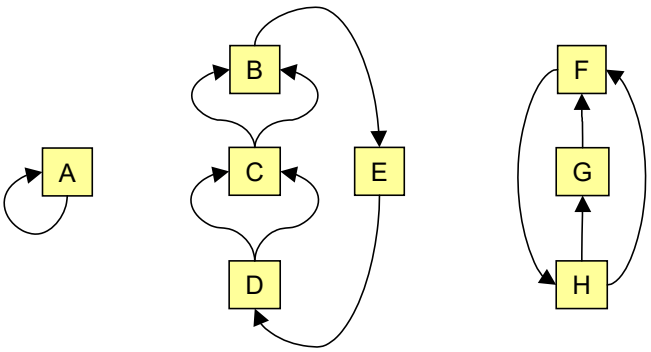


Figure 28.30 - The three aggregates of the draft hard cyclic kernel of the mail order database

The examination of the first aggregate, formed by table A (representing marketing actions) with a self-referencing foreign key, shows that it represents a hierarchical composition relationship: an instance of A is composed of one or several more elementary A instances. This pattern is quite common in database schemas (see the BOM application for example, developed in Case study *Bill of material*. However, this relationship **must be optional**, otherwise, since every A instance references another A instance, building a hierarchy of A instances is impossible, unless table A is empty. Therefore, the self-referencing must be optional.

In the second aggregate, the examination shows that table E represents customers and table D represents the pattern of the mail sent to a customer when the latter has issued a complaint. Considering that some customers should be satisfied, this relationship obviously must be optional, which breaks the circuit.

As to the third pattern, it represents shipments (table F) and the components of shipment (table G). Table H represents the messages attached to components that are gifts included in the shipment. The reference from H to F appears to be redundant

(probably for performance reason) with the references from H to G followed by that from G to F. It can be deleted. The role of the mandatory foreign key (which also is a unique key) from table F to H is not documented, and it is unclear whether it is essential or it is a mere modelling error. Anyway, in the absence of more precise information, we will keep it as such.

As a conclusion, the schema includes a single hard circuit made up of three tables F, G and H.

The cyclic kernel comprises 16 nodes and 27 links (one foreign key has been discarded), 16 of which are optional. Conditioned as described above, the hard cyclic kernel comprises 3 nodes and 3 links. The analysis of the optional links of the cyclic kernel allows 8 of these links to be transferred to the acyclic graph, leaving 8 of them to be executed by `update` statements. The final acyclic schema is structured into 17 levels, the first of which collecting 207 root tables.

The table graph of this database is available in script GRAPH-Mail-Order.sql. However, for confidentiality reason, table names have been replaced by anonymous character strings.

28.16 Brute force data loading technique

The technique we have used to prevent the DBMS from checking referential constraints when loading rows in cyclic structures can be generalized to all the tables. Indeed, deferring foreign key checking allows us to load rows in any orders, since the actual checking will be performed at commit point. In addition, this may reduce the loading time since checking is not carried out for each row but for all the rows as a whole.

However, this remark does not invalidate the procedure we have developed in this case study. First, not all DBMS allow foreign keys to be deferred. Second, loading a large volume of data takes much time. If any error is detected at commit type, all the process must be rerun. The best approach is to load data in topological sort, committing changes at the end of each table, then coping with cyclic parts as suggested above.

28.17 Performance

The production of a valid loading script through the procedure elaborated in this chapter is quite fast. Processing the example schema of Section 28.15, comprising 691 tables and 1,190 foreign keys, provides the following statistics:

– computing the cyclic kernel (16 nodes and 37 links): 0.2 seconds

– computing the hard cyclic kernel (3 nodes and 3 links): 0.01 seconds

– building the final acyclic schema: 3.05 seconds

– generating the loading script: 0.21 seconds

Total: 3.5 seconds.

Considering this excellent time, there is no need to try to optimize the procedure.

28.18 Complement: extracting the table graph of a database

For small schemas, tables NODE and LINK can be filled manually. For larger schemas, the description of the graph will best be derived from the dictionary of the database. Actually, the dictionary of the current database already includes a view, called SYS_INTER_TABLE, quite similar to the LINK table (Figure 28.31), so that the filling of table LINK is straightforward.

SYS_INTER_TABLE

Source-Table	Source-TableID	SourceKey	TargetKey	Target-TableID	Target-Table
CUSTORDER	3	5	1	1	CUSTOMER
DETAIL	4	6	2	2	PRODUCT
DETAIL	4	7	3	3	CUSTORDER

Figure 28.31 - View SYS_INTER_TABLE of the dictionary of database ORDERS

To avoid modifying the source database to be analyzed, we decide to process table graphs in a specific independent database, TOPO.db. Its initial tables NODE and LINK are loaded through scripts named "GRAPH-xxxxx", comprising a series of `insert into NODE` statements, followed by a series of `insert into LINK` statements. Scripts for short abstract table graphs have been written manually. Graphs derived from real databases can be generated through Script 28.23.

28.19 Topological sorting: application to book development

Let us suppose that a team of colleagues plan to write a book on SQL. They identify sixteen topics that are worth being developed in this book. To organize this material into chapters, they list, for each topic, the prerequisite topics, that is, the topics that must be read before. For instance, the chapters on *DML introduction*, *Relational algebra* and *Relational calculus* must be read before starting the reading of the chapter explaining the `select` *query*.

```
openDB ORDERS.db;
createDictionary;
outputOpen SCHEMA-ORDERS.sql;
execSQL UTIL-SELECT-parameters-for-Lists.sql;
select 'insert into NODE values (''N'
       ||cast(TableID as char)||''','''||TableName||''');'
from SYS_TABLE;
select 'insert into LINK values ('
       ||cast(SourceKey as char)||','
       ||'''N'||cast(SourceTableID as char)||''','
       ||'''N'||cast(TargetTableID as char)||''','
       ||'''''||K.KeyNull||''');'
from   SYS_INTER_TABLE I, SYS_Key K
where  I.SourceKey = K.KeyID;
execSQL UTIL-SELECT-parameters-standard-1.sql;
outputAppend window;
closeDB.
```

Script 28.23 - Creating the table graph of a database as a loading script for tables NODE and LINK of database TOPO.db

The result of this preliminary analysis is shown in the following table and in the equivalent graph of Figure 28.32. Since the concept of *optional link* is irrelevant for this application, all the links are declared mandatory.

This first version of the graph can be loaded from script SCHEMA-Cyclic-SQL-book.sql.

Topic	Prerequisite topics
Check	Relational calculus, DDL intro, select query
Data structures	SQL history, Set theory
Database languages	Relational calculus, Relational algebra
DDL intro	SQL history, Data structures
delete query	Triggers, DML intro, select query, update query
DML intro	SQL history, Database languages, DDL intro
insert query	Triggers, DML intro, select query
Logic	*none*
Relational algebra	Data structures, Set theory, Relational calculus
Relational calculus	Data structures, Logic, Relational algebra
select query	Relational calculus, Relational algebra, DML intro
Set theory	*none*
SQL history	*none*
Tables	DDL intro

Triggers	DDL intro, Check, select query, insert query, update query, delete query
update query	Triggers, DML intro, select query, insert query

The application of the procedure developed for the data loading problem identifies two super circuits:

- Relational algebra, Relational calculus
- delete query, insert query, Triggers, update query

By contracting these circuits, named respectively *Relational algebra and calculus* and *Data modification & triggers*, we produce the acyclic graph of Figure 28.33.

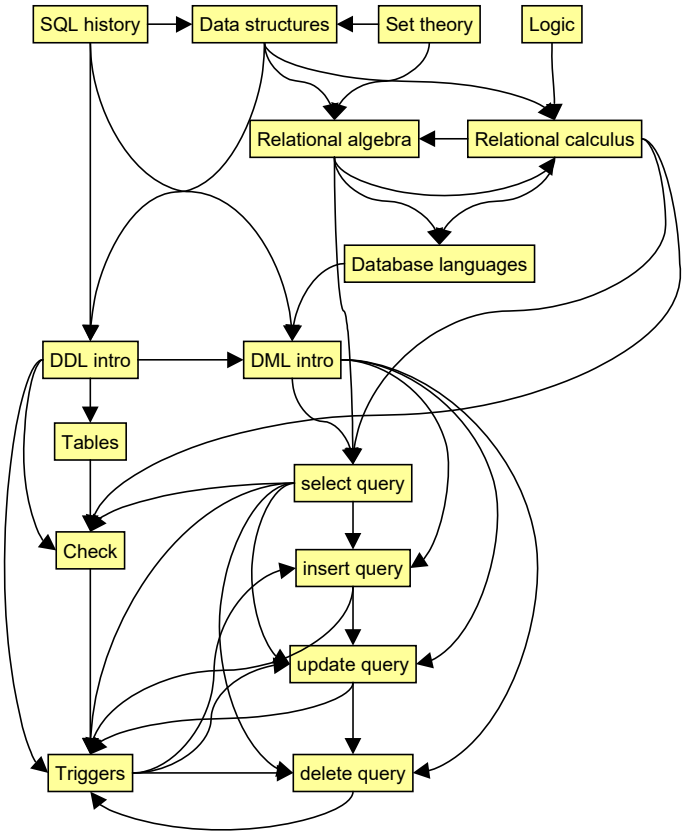


Figure 28.32 - The prerequisite relation between topics

Obviously, this result is not quite satisfying, since it hides essential topics that should appear explicitly in the book structure.

Let us analyze the case of super topic *Relational algebra and calculus*, contracting Relational algebra and Relational calculus. The author in charge of the chapter *Relational algebra* will show that this algebra is as powerful as first order logic, notably

in its particular form of *relational calculus*. So, the future reader of this chapter is expected to have some knowledge in *relational calculus*. Conversely, the author of chapter *Relational calculus* will show that the semantics of each calculus expression can be defined as an algebraic expression. Similarly, the future reader of this chapter is expected to have some knowledge in *relational algebra*.

However, in both case, the required preliminary knowledge is fairly superficial and does not cover, by far, all the material of the other chapter. Therefore, instead of merging these chapters, we define a new chapter which comprises a short introduction to relational calculus and algebra entitled *Relational algebra and calculus - Introduction*. This introduction is sufficient to tackle the more in-depth material of chapters Relational algebra and Relational calculus.

This new version of the graph can be found in script GRAPH-Acyclic-SQL-book.sql.

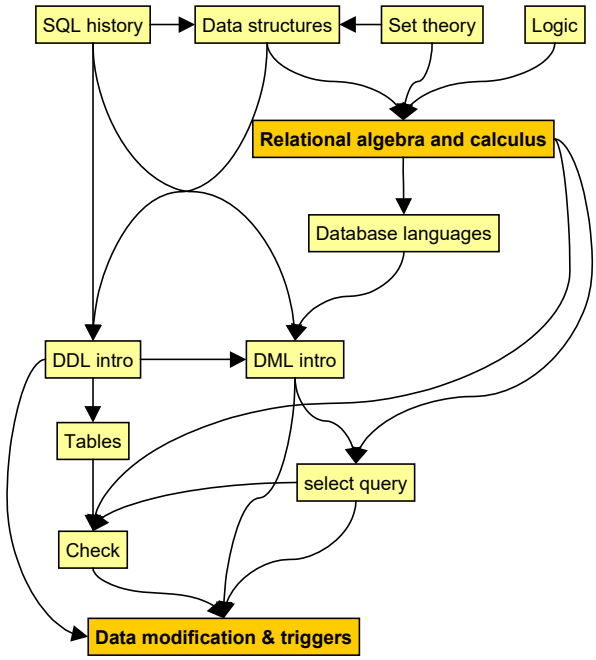


Figure 28.33 - An acyclic transformation of the prerequisite relation through the technique of *super nodes*

The super topic *Data modification & triggers* is transformed in the same way. New introductory chapter *Data modification - Introduction* shortly describes the data modification operators and their relationship with the trigger mechanism.

Through these transformations, we obtain the acyclic topic structure of Figure 28.34 that preserves the initial structure proposition and can be sorted in the usual way.

The table of Figure 28.35 shows the aggregated contents of table SORTED, from which we derive the table of contents of the book (Figure 28.36). Some topics have been located at a higher level, where they seem more natural (DDL intro for example). In addition, some topics exhibiting strong complementarity have been grouped under a common title. Finally, titles have been homogenized and developed a bit to make them more expressive, as in real books!

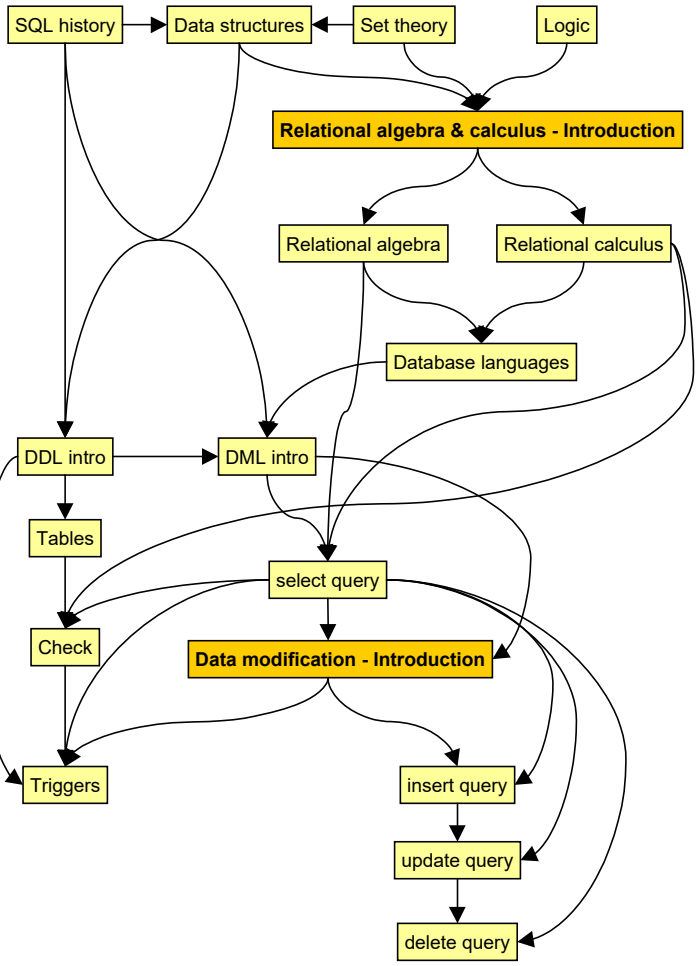


Figure 28.34 - An acyclic transformation of the prerequisite relation through the technique of *introductory chapters*

Level	Members
1	SQL history, Set theory, Logic
2	Data structures
3	DDL intro, Relational algebra & calculus - Introduction
4	Relational calculus, Relational algebra, Tables

```
| 5      | Database languages                                          |
| 6      | DML intro                                                   |
| 7      | select query                                                |
| 8      | Check, Data modification - Introduction                     |
| 9      | Triggers, insert query                                      |
| 10     | update query                                                |
| 11     | delete query                                                |
+--------+-------------------------------------------------------------+
```

Figure 28.35 - Contents of table SORTED

1. History of the SQL language
2. Theoretical reminder
   - Elements of set theory
   - Elements of first order logic
3. Relational data structures
4. Relational algebra and calculus
   - Introduction to relational algebra and calculus
   - Relational calculus
   - Relational algebra
5. Database languages
6. Introduction to the DDL sublanguage
7. Definition of relational tables
8. Introduction to the DML sublanguage
9. The select query
10. The check predicate
11. Data modification
    - Introduction to data modification
    - Trigger mechanism
    - The insert query
    - The update query
    - The delete query

Figure 28.36 - Tentative table of contents of the future book on SQL

28.20 Other applications

Topological sorting has many applications in a wide variety of domains. Let us mention some of them:

- *Scheduling and planning*: how to order a set of tasks.
- *Diplomatic protocol*: how to place persons in a row according to their relative status.

- *Software production*: how to assemble modules in such a way that all the modules the current module depends on are already assembled (*Make* procedure).
- *Software engineering*: declaring classes according to their *is-a* relation.
- *Generating procedural code* from an equational models, in which each object is defined in terms of other objects (see for example, "Hainaut, J.-L., *Bases de données et Modèles de calcul*, Dunod, 2005").

28.21 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Topological_Sorting**. They can be run from main script **Topo-MAIN.sql**, that displays the report selection box of Figure 28.37.

Two report styles are proposed: short, just providing the solution (Short report) and detailed, in which each step is described (Extended report). Then, a graph selection box opens (Figure 28.38).

A graph is defined by a script the name of which includes the name of the graph. For instance, graph **Cyclic-[10-12]** is created by script **GRAPH-Cyclic-[10-12].sql**.

For most of them, the name itself provides some information on the graph composition. For instance, graph **Cyclic-[22-23-11]** comprises circuits and is made up of 22 nodes, 23 mandatory links and 11 optional links (it defines the graph of Figure 28.21). For other graphs, the name refers to their nature: Acyclic-ORDERS, Cyclic-SQL-book, Mail-Order, etc.

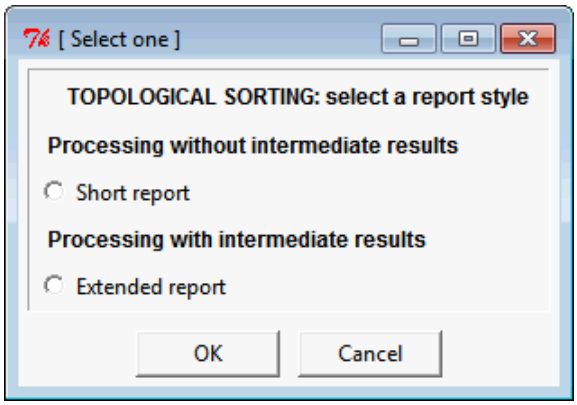


Figure 28.37 - Topological sorting - Selecting a report

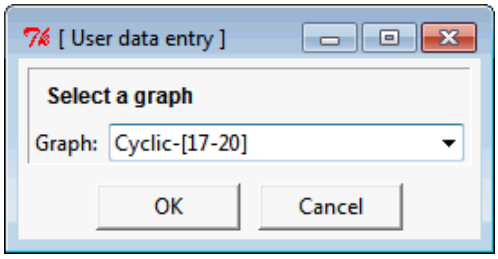


Figure 28.38 - Topological sorting - Selecting a graph

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

Case study 30

Classifying objects

Objective. In this study, we explore a particular way of classifying objects based on their attributes. This technique, called *Formal Concept Analysis*, or *FCA* for short, examines the composition of these objects and extracts *concepts*, that is, classes of objects that share the same set of attributes. By considering the inclusion relationship of the concept object sets, the concepts can be organized as a hierarchy.

Several techniques have been designed to extract concepts from a set of source objects and to build their hierarchy. We analyze the reasoning underlying these techniques and we develop one of the most popular of them, the Chein algorithm. We first translate this iterative algorithm into a Python procedure then we express it as an SQL script.

We propose a third, much simpler and faster technique that produces a remarkable subset of the Chein concept hierarchy. It appears that this technique, which can be coded as a single SQL query or in a small Python procedure, is more appropriate to database schema processing, specifically to conceptual schema normalization and to reverse engineering legacy databases.

The study develops four parametric applications to experiment with these algorithms and to evaluate their performance in time and space.

Keywords. symbolic classification, Formal Concept Analysis (FCA), Galois lattice, set operators, performance evaluation, database optimization, algorithm optimization

Table of content

1. Introduction and motivation
    A short illustration of FCA
    The solution is not always as straightforward
    Naming the concepts generated
    Some words of conclusion
    What next?
    About the scope of this study
2. Principles of FCA - Finding the concepts
    Rectangles
    Extending a rectangle
    Maximal rectangles and concepts
    Finding concepts by visual inspection
    Merging two rectangles
    Degenerated merging
    Processing non-binary relations
3. Principles of FCA - Building the concept hierarchy
4. Looking for a systematic procedure
    4.1 Finding the concepts
    4.2 Building the concept hierarchy
5. The Chein algorithm
    5.1 Analysis of the algorithm
    5.2 Procedural expression of the Chein algorithm
6. An SQL implementation of the Chein algorithm
    6.1 Operations on sets in SQL
        Solving a basic problem: identifying a set
        Set operations revisited
    6.2 SQL implementation of the Chein algorithm
7. A Python implementation of the Chein algorithm
8. Building the concept hierarchy
    8.1 An SQL implementation
    8.2 A Python implementation
9. An alternative FCA implementation: ISA recovery
    9.1 The basic algorithm
    9.2 An SQL translation of the ISA recovery algorithm
    9.3 The case of empty concepts
        Preserving the source objects with no proper attributes
        Empty intermediate concepts
        Creating the empty intermediate concepts
    9.4 Python translation of ISA recovery algorithm
10. Experimentation
    10.1 The scripts
    10.2 Sample formal contexts
    10.3 The context generator
11. Evaluation of the FCA implementations
    11.1 Simplicity of algorithms
    11.2 Preservation of the source concepts
    11.3 Nature of the attributes of the concepts

11.4 Creation of empty intermediate concepts
11.5 Space requirements
11.6 Runtimes
12. Improving the performance of the algorithms
12.1 Chein algorithm (Python implementation)
12.2 Chein algorithm (SQL implementation)
12.3 Implementation of the ISA recovery algorithm
12.4 Building of the concept hierarchy (SQL implementation)
12.5 Building of the concept hierarchy (Python implementation)
13. On the representation of concept hierarchies
14. Querying the concept hierarchy
15. Applications
16. A short bibliography

30.1 Introduction and motivation

Let us first read Wikipedia on the main subject of this study [Wikipedia, 2023]:

> *Formal concept analysis (FCA) is a principled way of deriving a concept hierarchy or formal ontology from a collection of objects and their properties. Each concept in the hierarchy represents the objects sharing some set of properties; and each sub-concept in the hierarchy represents a subset of the objects (as well as a superset of the properties) in the concepts above it. The term was introduced by Rudolf Wille in 1981, and builds on the mathematical theory of lattices and ordered sets that was developed by Garrett Birkhoff and others in the 1930s.*
>
> *Formal concept analysis finds practical application in fields including data mining, text mining, machine learning, knowledge management, semantic web, software development, chemistry and biology.*

The technique we will examine in this case study, called *Formal Concept Analysis*, or *FCA*, aims at extracting pertinent concepts from large data sets. Contrary to statistical analysis, it does not rely on numerical computing, such as the extraction of average and standard deviation, or regression analysis. Rather, it applies symbolic transformations on simple graphs, based on the mathematical theory of *Galois lattice* [Huchard, 2000].

We will explore this concept in various application domains related to knowledge extraction and processing, notably structuring large sets of elementary objects and database schema manipulation[1].

A short illustration of FCA

We will illustrate FCA by one of its historical applications: eliciting *subtype/supertype* hierarchy in a database schema.

The basic idea is that, when two or more entity types appear to have attributes in common, these common components can be extracted to form a new entity type, that is declared a supertype of the former.

Let us have a look at the schema of Figure 30.1 that describes two data sets, named ENGINEER and SECRETARY. The very nature of these data sets is irrelevant in this presentation: they can be structured data files, tables in a standard relational database, table types in SQL3 object-relational databases, document types, Java classes, object classes of an ontology or entity types of a database we intend to develop (what one generally calls its *conceptual schema*). Let us call them **concepts**, in that they denote abstract categories of concrete objects.

1. The first domain concerns untyped instances (such as messages) for which we try to derive the most relevant type(s). In the second domain, we deal with objects that are already types, of which we ignore the instances. Only the first level of abstraction distinguishes these two domains.

To the readers who may feel uncomfortable with this idea of *abstraction*, we suggest to consider Figure 30.1 as the schema of a database comprising two tables. The data rows are the concrete objects while the schema represents their abstraction.

We consider that the concept named ENGINEER represents the class of employees of a company in charge of identifying and solving technical problems. Each of them is characterized by an *employee Id* (represented by attribute *Emp#*), a *name*, a skill *level*, a *salary* and a *specialty*. The other concept is that of SECRETARY, designating the class of the employees in charge of administrative and logistical tasks. Each secretary has a similar set of attributes: *employee Id*, *name*, *salary* and the *language* in which they are most fluent.

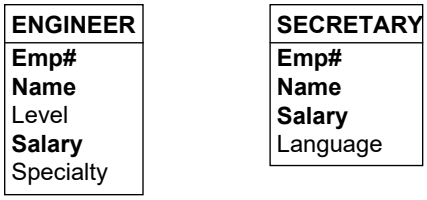


Figure 30.1 - An elementary source schema: two concepts have some attributes in common

The property of this schema we are interested in is that these concepts *share common attributes*, namely Emp#, Name and Salary (in bold face in Figure 30.1). This naturally suggests that all the engineers and all the secretaries are also instances of a more general concept the attributes of which are these common attributes. We should assign a name to this new concept. It seems natural to call *Employees* the category of objects

– that all have an *employee number*, a *name* and a *salary*?

– and that comprises two subcategories, *engineers* and *secretaries*?

Hence the concept EMPLOYEE.

Now, our schema comprises three concepts, EMPLOYEE, ENGINEER and SECRETARY. EMPLOYEE is more general than both ENGINEER and SECRETARY. Conversely, the latter are more specific than EMPLOYEE. They often are called the **subconcepts** of EMPLOYEE, while the latter is the **superconcept** of ENGINEER and SECRETARY.

The new schema is represented in Figure 30.2. Considering the concrete level, it states that,

– each engineer is also an employee

– each secretary is also an employee

and, conversely,

– some employees are engineers

– some employees are secretaries.[2]

A star structure (a triangle plus three branches) connects the general concept (the thick branch) with its specific concepts (the thin branches). As to the attributes, we have **migrated** the common ones to the general concepts while the attributes specific to a concept are kept with it.

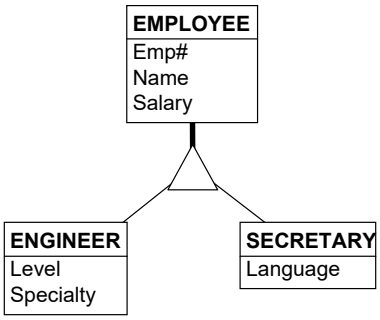


Figure 30.2 - Extracting common attributes as a new concept

This schema must be read as follows (example of ENGINEER):

- ENGINEER has five attributes,
- among them, two are **specific** (also called **proper**) attributes: Level and Specialty,
- and three are **inherited** attributes: Emp#, Name and Salary.

Usually, one represents only the proper attributes of a concept, as shown in the schema of Figure 30.2. We could also have chosen to show all the attributes, proper and inherited, of each concept but this would have made the schema less readable. In a sense, each attribute is located where it is the most relevant.

Four important observations to keep in mind for the following discussion:

- The restructuring operation creates a *hierarchy* of concepts.
- This restructuring operation is only valid if attributes with the same name in the specific concepts convey the same meaning. Should SECRETARY attribute Language be named Specialty, the resulting schema would have been quite different and most probably less informative.
- Being based on purely symbolic (more precisely *character string*) manipulations, this operation can be automated[3].
- In set theoretic terms, the new concept is built as the *union* of existing concepts and its attributes are the *intersection* of those of the existing concepts.

2. If Figure 30.2 were to represent a database schema, we would have to specify two additional properties: (1) whether an employee can be neither an engineer nor a secretary, and (2) whether employees and secretaries form separate sets. We will leave these properties aside.
3. Except of course for the naming of the new concepts, which will be discussed in a further section.

Note

The *subconcept/superconcept* relation that holds among the set of concept is an important feature of all the knowledge representation systems (the conceptual schema of databases being one of them). The linguistic interpretation of this relation often takes the form *each instance of A is an instance of B*, abbreviated to *each A **is a** B*. Hence the synthetic name ***is-a***, or any variation such as ***ISA***, that is commonly used to denote this type of relation.

The solution is not always as straightforward

Sometimes, things can get more tricky. Consider for example the initial schema of Figure 30.3, which is an extension of that of Figure 30.1. While ENGINEER and SECRETARY have common attributes, it appears that ENGINEER also has some common attributes with MANAGER, but those are different from the first set: {Emp#,Name,Salary} for the first two concepts and {Emp#,Name,Level} for the last two concepts.

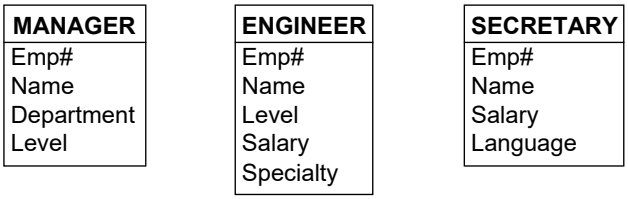


Figure 30.3 - A source schema comprising three concepts

A first attempt would consist in completing the transformed schema of Figure 30.2 with the concept MANAGER. We then observe that MANAGER and EMPLOYEE have two attributes in common, namely Emp# and Name. Applying the reasoning we used above, we create a new concept, PERSONNEL, that collects these common attributes. PERSONNEL has two subconcepts: MANAGER and EMPLOYEE, as shown in Figure 30.4.

The result is not quite satisfying. Indeed, MANAGER and ENGINEER have common attribute Level, that cannot be moved to PERSONNEL nor to EMPLOYEE and that is left unprocessed.

The correct transformation relies on the introduction of a new concept of which MANAGER and ENGINEER are the subconcepts. Let us call it OPERATIONAL and let it be assigned attribute Level. Since MANAGER also have attributes Emp# and Name, we specify OPERATIONAL as a subconcept of PERSONNEL: in this way, OPERATIONAL inherits these attributes that also are inherited by MANAGER (Figure 30.5).

This schema shows only the proper attributes of each concept. In schema 30.6, we have also introduced the inherited attributes. For each concept, it shows the proper (in blue) and inherited attributes. This view is less concise but it makes the composition of all concepts explicit.

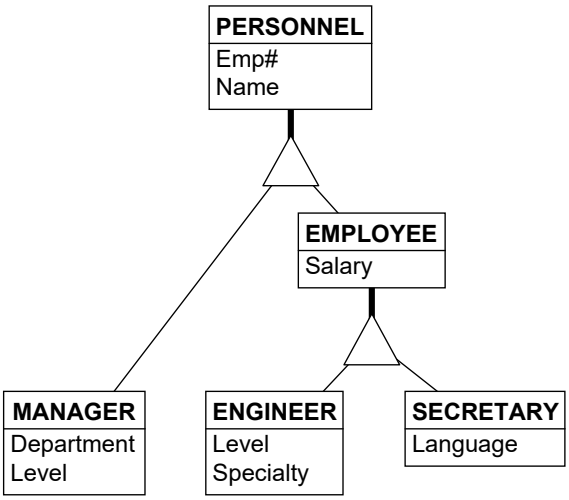


Figure 30.4 - Extracting common attributes: a first tentative

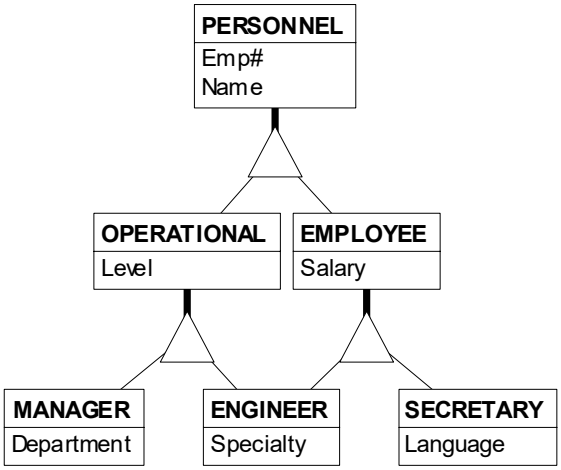


Figure 30.5 - Extracting common attributes: a more satisfying solution

Naming the concepts generated

The assignment of a semantically expressive name to each new concept is usually based on common sense and on domain knowledge. In some cases the nature of the proper attributes of a concept may suggest its name. For example, in the schema of Figure 30.5, considering that the concept EMPLOYEE represents people who receive a salary, we may have more appropriately renamed it SALARIED.

The use of AI technologies could be considered, such as domain ontologies or intelligent conversational systems such as ChatGPT or currently emerging competi-

tors (2023). However, such techniques go well beyond the scope of this (modest) study.

On this issue, it should be observed that naming new concepts is not always necessary or even useful. In the applications that illustrate FCA techniques in the scientific literature, a concept is defined entirely by a set of objects and a set of attributes. Thus, what we have called EMPLOYEE in Schema 30.5 will simply be identified as the unnamed concept

```
(ENGINEER,SECRETARY)x(Emp#,Name,Salary)
```

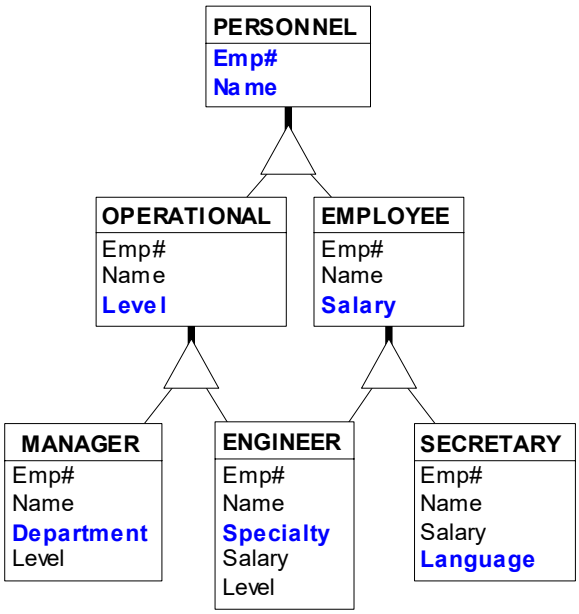


Figure 30.6 - The final schema showing both the proper and inherited attributes

Some words of conclusion

Starting from the composition (the set of their attributes) of initial concepts MANAGER, ENGINEER and SECRETARY, we have discovered, through symbolic (or *formal*) techniques, three new concepts of interest. This is the essence of the *Formal Concept Analysis* and the explanation of its name.

Though we have not distinguished them explicitly, the construction of the final schema comprises two distinct phases, that require specific reasonings and to which we will devote different algorithms:

– finding the concepts,

– building the subconcept/superconcept hierarchy.

The way we have built the final schema is fairly intuitive and obviously cannot solve more complex problems that may comprise thousands of initial objects[4]. So, we have to design a systematic procedure to discover the implicit concepts, which is our goal in the next sections.

What next?

In the next section (Section 30.2), we analyze the general principles of FCA that underlie the discovery of the concepts. In Section 30.3 we study the building of the concept hierarchy. In Section 30.4, we develop a systematic procedure to discover the concepts.

Section 30.5 describes the Chein algorithm, one of the most popular technique that translates this procedure. This algorithm is implemented in SQL in Section 30.6 and in Python in Section 30.7. The implementation in SQL and Python of the building of the concept hierarchy is described in Section 30.8. A alternative algorithm, specially designed for database schema processing, is described in Section 30.9.

Section 30.10 is devoted to the presentation of experimental material (a portfolio of typical problems and four SQL scripts to solve them) while Section 30.11 evaluates the performance of the four implementations according to qualitative and quantitative criteria. Section 30.12 suggests some ways to improve the execution times of the algorithms.

About the scope of this study

It is important to recall the scope of the series of case studies in which this one is part. Their objective is to examine and evaluate the expressive power of database concepts and technologies in modeling and solving a wide variety of problems. They do not constitute a theoretical statement or a state of the art of the problem domain they address. Based on one or more representative examples, each study attempts to derive modeling and solving principles that can help readers and practitioner feel more comfortable in dealing with such problems.

In this study, we describe a very small subset of the FCA domain that is far from encompassing all aspects of it. In particular, the mathematical basis of FCA will be limited to the part required to describe the selected problems and to develop solutions and their implementation.

4. Databases often include hundreds of tables (the figure in Figure 30.3 includes only three). The analysis of emails or tweets, tagged with keywords (their attributes), can lead to the processing of several thousands of messages.

30.2 Principles of FCA - Finding the concepts

The techniques of *Formal Concept Analysis* contribute to two objectives: finding the concepts that underlie the source data set and ordering these concepts in order to build the concept hierarchy. In this section, we explore the first objective.

In the FCA world, the initial problem is specified as a *set objects*, each object being *assigned* a *set of attributes*. In the examples examined in the previous section, these initial objects already were concepts. This is due to the particular nature of the problem domain, that of data structures. FCA considers a more general domain, in which a concept is a subset of objects that share common attributes. Some objects may become concepts while others will be absorbed by new concepts and therefore will disappear.

This initial problem is called the **Formal context**. It comprises three parts, `G`, a set of objects, `M`, a set of attributes[5] and `I(G,M)`, the attribute assignment relation. The fact `(g,m) ∈ I` indicates that object `g` has attribute `m`. Actually, the only interesting part of this specification is relation `I` (in this section we ignore objects without attributes and attributes that do not appear in any object). As this relation is binary, it can be represented by a matrix[6] as illustrated by Figure 30.7, that expresses the schema of Figure 30.3 (we ignore the gray cells for now). The object and attribute names have been abbreviated to fit in the limited width of this page.

Each row represents a source object and its attributes while each column represents a source attribute and all the objects it is associated with. A cross at the intersection of an object row and an attribute column (i.e., a *cell*) indicates that this object has this attribute. The matrix shows graphically that the SEC object has the attributes Emp, Nam, Sal and Lan. It also shows that the Sal attribute is assigned to the ENG and SEC objects.

	Emp	Nam	Dep	Lev	Sal	Spe	Lan
MAN	x	x	x	x			
ENG	x	x		x	x	x	
SEC	x	x			x		x

Figure 30.7 - Matrix representation of a formal context

We postulate that the order of the rows and that of the columns are irrelevant. If we swap two rows or two columns, the resulting matrix provides exactly the same information.

Relation `I` can be given various equivalent representations. With database structure in mind, we could think of a direct translation of the mathematical expression:

5. These symbols come from their German names: G = Gegenstände; M = Merkmale.
6. Called *incidence matrix* by mathematicians.

```
create table I(G varchar(256),M varchar(256));
insert into I values ('MAN','Emp'),('MAN','Nam'), ...;
```

We will sometimes use a more concise form in which we associate with each object name the **list** of the attribute names ; a little more tricky to manipulate by algorithms but better suited to manual data entry:

```
create table I(G varchar(256),M varchar(256));
insert into I values ('MAN','Emp,Nam,Dep,Lev'), ...;
```

Rectangles

The key concept of FCA is the **rectangle**. A rectangle is made up of a set of objects **A** and a set of attributes **B** such that all the cells defined by their intersection include a cross. It will be noted **A**x**B**. Since the order of the rows and the columns are meaningless, the rows (and the columns) of a rectangle need not be contiguous. For example, in the matrix of Figure 30.7, `(ENG,SEC)x(Nam,Sal)` is a rectangle. `(SEC)x(Emp,Nam,Sal,Lan)` and `(MAN)x(Dep)` are also valid rectangles. Two operators will help us reason about the properties of rectangles:

- **`att(A)`**, where `A` is a set of objects, denotes the set of attributes shared by all the objects in `A`
- **`obj(B)`**, where `B` is a set of attributes, denotes the set of objects which share all the attributes in `B`.

Some examples:

```
att(MAN) = (Emp,Nam,Dep,Lev)
att(MAN,SEC) = (Emp,Nam)
obj(Nam,Sal) = (ENG,SEC)
```

Extending a rectangle

Sometimes, a rectangle can be **extended** by adding to it either an object or an attribute. For example, if we add attribute `Emp` to rectangle `(ENG,SEC)x(Nam,Sal)`, we get the new valid rectangle `(ENG,SEC)x(`**`Emp`**`,Nam,Sal)` shown on grey background in Figure 30.7. The condition is that the additional attribute appears in (at least) all the objects of the rectangle. Conversely, we can add an object to a rectangle if it is assigned (at least) all the attributes of this rectangle. Following these rules, we could not add object MAN (which has no Sal attribute) nor attribute Lan (which is not assigned to ENG) to this rectangle.

Maximal rectangles and concepts

Among the (many) rectangles of the matrix, the **maximal rectangles** are particularly remarkable. A rectangle is maximal if there is no way to extend it. Considering `A` a set of objects and `B` a set of attributes, rectangle `AxB` is maximal if, and only if,

`att(A) = B` and `obj(B) = A`

Explanation:

The first part tells that all the attributes of set `B` are shared by all the objects of set `A`; in other words, there is no attribute outside of `B` that would be shared by all the objects of `A`. According to the second part, there is no object outside of `A` that would be assigned all the attributes of `B`. Therefore, there is no object and no attribute that can be added to a maximal rectangle.

Now, the fundamental definition of FCA:

> *in a definite context, each **concept** is a maximal rectangle and each **maximal rectangle** is a concept.*

The intuitive idea behind this definition is that a set of objects that all share the same set of attributes (and this, exclusively), surely reveals a semantic entity hidden among all the objects of the context.

So, we are able to state our first objective, *finding the maximal rectangles in a formal context*.

Finding concepts by visual inspection

Carefully examining the matrix of Figure 30.3 to identify its maximal rectangles is a nice exercise to check if we have understood the definition of a concept. If we try, we should find this set of concepts,

```
(MAN)x(Emp,Nam,Dep,Lev)
(SEC)x(Emp,Nam,Sal,Lan)
(ENG)x(Emp,Nam,Lev,Sal,Spe)
(MAN,ENG)x(Emp,Nam,Lev)
(ENG,SEC)x(Emp,Nam,Sal)
(MAN,ENG,SEC)x(Emp,Nam)
```

of which we derive the graphical representation of figure 30.8.

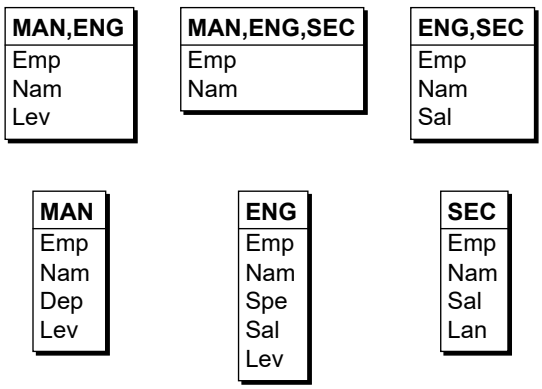


Figure 30.8 - Graphical representation of the concepts

Merging two rectangles

To formalize a procedure to find the concepts of a context, we need a new basic operator: **merging** two concepts.

If we find two rectangles A1xB1 and A2xB2 such that sets B1 and B2 have some common attributes (i.e., their intersection is not empty), then we can **merge** them to derive a new valid rectangle defined as follows:

$$(A1 \cup A2) \times (B1 \cap B2)$$

We can find several examples of this pattern in the set of concepts we have found in the previous section. Let us consider this one,

(ENG)x(Emp,Nam,Lev,Sal)
(SEC)x(Emp,Nam,Sal,Lan)

Since their attribute parts have three common attributes, we can merge them to derive this concept:

(ENG,SEC)x(Emp,Nam,Sal)

If we take a look at the matrix of Figure 30.7, we observe that this derived concept is precisely the gray rectangle.

As another example, the following concepts, that share attributes Emp and Nam,

(MAN,ENG)x(Emp,Nam,Lev)
(ENG,SEC)x(Emp,Nam,Sal)

can be merged to produce this one:

(MAN,ENG,SEC)x(Emp,Nam)

Degenerated merging

Interestingly, if $B1 = B2$, the merging operator reduces to

$$(A1 \cup A2) \times B1$$

which is the precise description of the *extension of a rectangle by adding an object*, or, more generally, *a set of objects*.

This rule can be extended to $B1 \subseteq B2$, with the same result: a new rectangle defined by $(A1 \cup A2) \times B1$. We do not know whether this new rectangle is maximal, but we know for sure that $A1 \times B1$ definitely **is not** a maximum rectangle and will not appear in the final schema.

Processing non-binary relations

The FCA is based on a binary relation, **I(G,M)**, indicating whether the object $g \in G$ is characterized by the attribute $m \in M$. For example, in a context describing the characteristics of a set of people, we distinguish those who have a university degree from those who do not. However, we would find it more interesting to also know the

faculty in which they obtained this degree. In other words, we are interested not only in the existence of an attribute but also in its value for a definite object. Ideally, we would need a ternary extension of `I` defined as `I'(G,M,V)` where `V` is a set of values. So, the context would tell us that *Smith* has no *university degree* and *Johnson* has a *university degree* in *Humanities*.

This can be done by replacing the initial single attribute by a series of attributes the names of which include a value[7]. Of course, this only works if the set of values is not too large. In the example described above, the attribute

UDegree

could be expanded into this series of attributes

UDegree:Arts, UDegree:Classics, UDegree:Humanities, *etc*.

30.3 Principles of FCA - Building the concept hierarchy

The second step of FCA aims to create the subconcept/superconcept hierarchy. We will see that it is the easy part of the FCA procedure.

If we examine very carefully the schema of figure 30.8, and if we keep in mind the concept hierarchy we built manually in figure 30.6, we can suggest a rule to automatically derive this hierarchy. Actually, two equivalent rules work equally well. Considering the pair of concepts $C1 \equiv A1 \times B1$ and $C2 \equiv A2 \times B2$, $C1$ is a subconcept of $C2$ if either condition is met:

$A1 \subset A2$

$B1 \supset B2$

These rules rely on the assumption that the inclusion relation is asymmetric, i.e., two concepts cannot have the same object part or the same attribute part. In other words, the concepts have distinct object parts as well as distinct attribute parts.

Let us remind the concepts of figure 30.8:

```
(MAN)x(Emp,Nam,Dep,Lev)
(SEC)x(Emp,Nam,Sal,Lan)
(ENG)x(Emp,Nam,Lev,Sal,Spe)
(MAN,ENG)x(Emp,Nam,Lev)
(ENG,SEC)x(Emp,Nam,Sal)
(MAN,ENG,SEC)x(Emp,Nam)
```

From the application of the rules, we conclude that,

`(MAN)` is a subconcept of `(MAN,ENG)`

`(MAN)` is a subconcept of `(MAN,ENG,SEC)`

7. This technique is known as *One-hot Encoding* (https://en.wikipedia.org/wiki/One-hot) or *Conceptual Scaling* [Kuznetsov 2015].

`(SEC)` is a subconcept of `(ENG,SEC)`
`(SEC)` is a subconcept of `(MAN,ENG,SEC)`
`(ENG)` is a subconcept of `(MAN,ENG)`
`(ENG)` is a subconcept of `(ENG,SEC)`
`(ENG)` is a subconcept of `(MAN,ENG,SEC)`
`(MAN,ENG)` is a subconcept of `(MAN,ENG,SEC)`
`(ENG,SEC)` is a subconcept of `(MAN,ENG,SEC)`

Being transitive, the inclusion relation includes many uninteresting couples. For instance, knowing that *`(MAN)` is a subconcept of `(MAN,ENG)`* and *`(MAN,ENG)` is a subconcept of `(MAN,ENG,SEC)`*, there is no need to state that *`(MAN)` is a subconcept of `(MAN,ENG,SEC)`*. By removing these pairs, we get the *transitive reduction* of the inclusion relation, that is,

`(MAN)` is a subconcept of `(MAN,ENG)`
`(SEC)` is a subconcept of `(ENG,SEC)`
`(ENG)` is a subconcept of `(MAN,ENG)`
`(ENG)` is a subconcept of `(ENG,SEC)`
`(MAN,ENG)` is a subconcept of `(MAN,ENG,SEC)`
`(ENG,SEC)` is a subconcept of `(MAN,ENG,SEC)`

The resulting schema is shown in figure 30.9. Both their proper and inherited attributes are associated with each concept.

30.4 Looking for a systematic procedure

Let us go back to the source schema 30.3, of which Figure 30.10 is a copy (with abbreviated names). It forms a formal context comprising three objects (we do not yet consider them concepts) that can be represented by these rectangles:

```
(MAN)x(Emp,Nam,Dep,Lev)
(ENG)x(Emp,Nam,Spe,Sal,Lev)
(SEC)x(Emp,Nam,Sal,Lan)
```

30.4.1 Finding the concepts

We will apply a bottom-up procedure that starts from these three objects.

• Step 1

The source objects having different sets of attributes and none of these attribute sets being a strict subset of the others, each of these objects corresponds to a maximal rectangle and can be declared a *concept*.

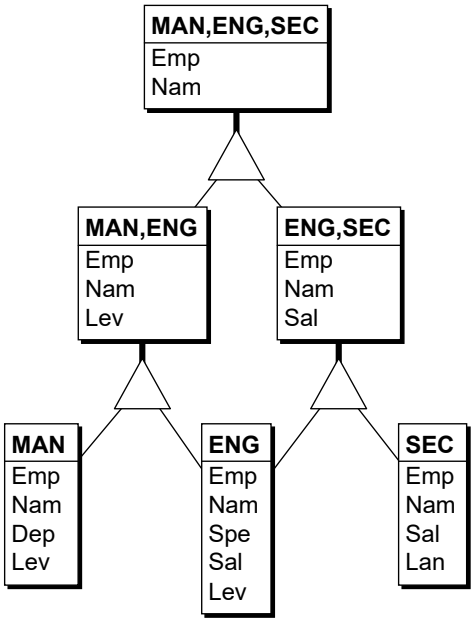


Figure 30.9 - The concept hierarchy is built from the transitive reduction of the inclusion relation of the object parts of the concepts

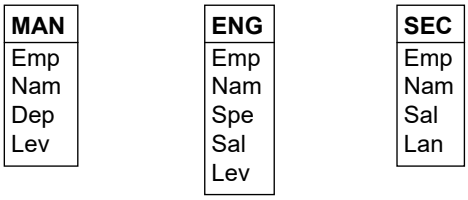


Figure 30.10 - Reminder of the source objects of figure 30.3

• Step 2

Then, we examine all the pairs of these rectangles to identify their common attributes:

- (MAN) and (ENG) : Emp, Nam, Lev
- (MAN) and (SEC) : Emp, Nam
- (ENG) and (SEC) : Emp, Nam, Sal

We create a rectangle by **merging** the rectangles of each of these pairs as follows:

- its name is the *union* of the names of the object sets of the pair

– its attributes are the *intersection* of the names of the attribute sets of the pair, provided this intersection is not empty

So, we create three new rectangles,

```
(MAN,ENG)x(Emp,Nam,Lev)
(MAN,SEC)x(Emp,Nam)
(ENG,SEC)x(Emp,Nam,Sal)
```

They are depicted in Figure 30.11.

Since the attribute parts of these rectangles are different, there is no need to merge them. Furthermore, when generating one of the new rectangles, no source rectangles had an attribute part that was identical to that of the generated rectangle (otherwise we could have discarded this source rectangle).

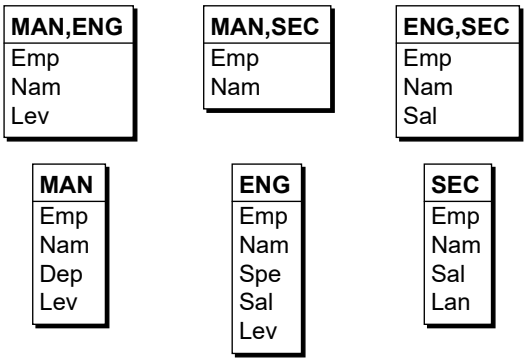


Figure 30.11 - Merging pairs of source rectangles - First level

• Step 3

So far, so good. Now, we examine each pair of the new rectangles and we consider their common attributes:

– `(MAN,ENG)` and `(MAN,SEC)`: `Emp,Nam`

– `(MAN,SEC)` and `(ENG,SEC)`: `Emp,Nam`

– `(MAN,ENG)` and `(ENG,SEC)`: `Emp,Nam`

that are depicted in Figure 30.12.

The attribute parts of each pair intersect, so that a new rectangle can be generated from this pair (Figure 30.12):

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN,SEC,ENG)x(Emp,Nam)
(MAN,ENG,SEC)x(Emp,Nam)
```

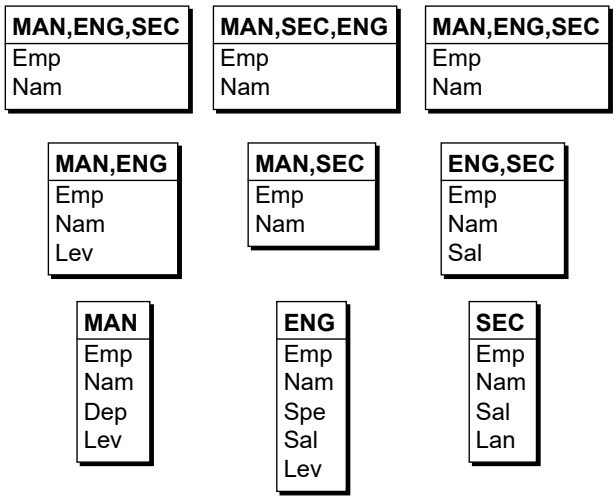


Figure 30.12 - The final schema, before cleaning

• Step 4

These three rectangles happen to be exactly the same. We only need one! We notice that the attribute part of `(MAN,SEC)`, i.e., `(Emp,Nam)`, is the same as that of the rectangles it helped generate. Therefore, this source rectangle is not maximal and we can discard it.

The top level consists of only one (maximal) rectangle, which completes the process. The final schema comprises all the rectangles that have not been discarded and that are interpreted as the six concepts of Figure 30.13.

30.4.2 Building the concept hierarchy

The objective is to build a binary relation ISA(Sub,Super) on the set of concepts such that if (C1,C2) $\in$ ISA, then C1 is a subconcept of C2. We suggest a simple and intuitive technique in two steps.

In the first one, we examine each pair (C1 $\equiv$ A1xB1, C2 $\equiv$ A2xB2) of concepts and we check whether the object part of one of them is included in the object part of the other. If A1 $\subset$ A2 then (C1,C2) $\in$ ISA.

The second step discards the transitive couples from ISA, that is, it computes its *transitive reduction*. For each couple (Cs,CS) of ISA, we search ISA for any pair of couples ((C1s,C1S), (C2s,C2S)) such that

Cs = C1s and C1S = C2s and C2S = CS

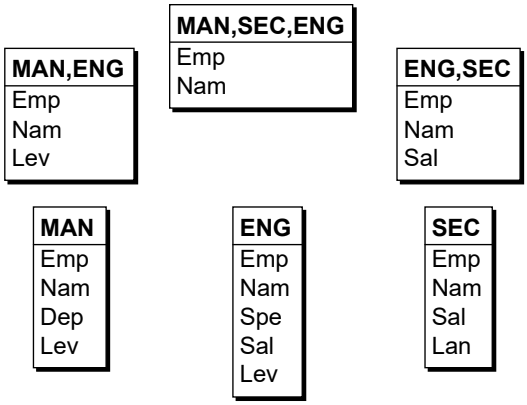


Figure 30.13 - The six concepts of the final schema

If we find such a pair of couples, then (Cs,CS) is transitive and is discarded. The ISA couples that remain form the desired transitive reduction. Deriving the subconcept/superconcept relation of the demonstration case of Figure 30.13 is immediate (see Figure 30.9).

30.5 The *Chein* algorithm

The operations described in the preceding section can be assembled and synthesized into a formal procedure, which we will call *the Chein algorithm*. The mathematical foundation of this algorithm has been described (more precisely *sketched*) in a research note by M. Chein [Chein, 1969] and has been discussed in many further references, for example in [Sarmah, 2013].

We will develop two implementations of this algorithm. The first one expresses it into a sequence of SQL queries. The second one is a direct translation in Python.

30.5.1 Analysis of the algorithm

The manual procedure has a bottom-up layered structure that is general enough to be formalized as an iterative algorithm. Each iteration creates a new level of rectangles based on the analysis of the set of rectangles generated by the previous level.

Let us call `R1` the set of the rectangles created in the previous level and `R2` the set of rectangles being created in the current level.

The content of `R1` in the **first** level is made up of the rectangles created on the objects of the context.

When the processing of the **current** level starts, the set `R1` comprises rectangles, some of which may be maximal, and the set `R2` is empty. Then, we consider all the

pairs of distinct rectangles of R1. Let $r_i = (g_i) \times (m_i)$ and $r_j = (g_j) \times (m_j)$ be such a pair under examination. When we compare their structure, the following cases may arise:

- These rectangles share no common attributes ($m_i \cap m_j$ is empty): we abandon this pair.
- These rectangles have some common attributes, denoted by $m_{ij} = m_i \cap m_j$: there is an opportunity to merge them into a new rectangle $r_{ij} = (g_{ij}) \times (m_{ij})$, where $g_{ij} = g_i \cup g_j$. Then, we consider two situations:
  - there is no rectangle in R2 that has the same attribute part m_{ij}: we merge r_i and r_j to add the new rectangle r_{ij} to R2.
  - on the contrary, R2 happens to already include a rectangle $r_k = (g_k) \times (m_k)$, such that $m_k = m_{ij}$: we augment r_k with r_{ij}, which gives it this new composition: $r_k = (g_k \cup g_{ij}) \times (m_k)$.

When rectangle r_{ij} has been either added or merged into r_k, we compare it to its source rectangles r_i and r_j from R1. First, we observe that g_{ij} is always larger than g_i and g_j.[8] Let us suppose that $m_i = m_{ij}$. This means that the new rectangle r_{ij} extends the source rectangle r_i. Therefore, r_i is not maximal and will not be part of the solution. To state this fact, we mark r_i as *discarded*. Otherwise, if $m_i \neq m_{ij}$, r_i still is a potential maximal rectangle. At the end of the current level, all the rectangles of R1 that have not been marked as discarded definitely are maximal rectangles.

When, at the completion of a level, R2 is empty or contains one rectangle only, the analysis is completed. This last rectangle, if any, is maximal.

All the maximal rectangles identified in the successive levels of the algorithm form the concepts that can be extracted from the source context.

Before developing implementations of this algorithm, let us synthesize the main operations carried out during each level of the FCA algorithm:

Let set R, initially empty, contain the solution, that is, all the maximal rectangles.

for each pair (r_i, r_j) of distinct rectangles in R1 that match

- we merge them into rectangle $r_{ij} = (g_i \cup g_j) \times (m_i \cap m_j)$
- we search the current state of R2 for another rectangle r_k with the same attribute part as r_{ij}; if we find one we extend r_k with r_{ij}; otherwise, we add r_{ij} to R2
- if one of the arguments r_i or r_j of the merging has the same attribute part as r_{ij}, we discard it from R1

At the end of the level, all the rectangles in R1 that have not been discarded are maximal rectangles and therefore are saved in R.

In the next level, R2 replaces R1 and is emptied

8. At any time of the analysis process, all the rectangles in R1 have distinct g parts by construction. So, their union is always larger that each of them.

30.5.2 Procedural expression of the Chein algorithm

Now, the Chein algorithm can be precisely structured as an iterative procedure in which each iteration builds a level[9]. During the current iteration, that creates the current level, we consider three sets:

- `R1`: the set of rectangles built by the preceding iteration
- `R2`: the set of rectangles being built by the current iteration
- `R`: the set of maximal rectangles identified so far.

`R = { }`

Initially, `R2` contains the rectangles created from each object of the formal context and its attributes; they are not marked as discarded.

while `R2` comprises at least two rectangles:

`R1 = R2`

`R2 = { }`

for each pair (r_i, r_j) of distinct rectangles of `R1`:

$m_{ij} = m_i \cap m_j$

$g_{ij} = g_i \cup g_j$

if $m_{ij} \neq \{\}$:

if `R2` includes a rectangle $r_k = (g_k) \times (m_{ij})$:

extend r_k with g_{ij} (i.e., $r_k = (g_k \cup g_{ij}) \times (m_k)$)

else

add rectangle $(g_{ij}) \times (m_{ij})$ to `R2`

endif

if $m_i = m_{ij}$: discard r_i from `R1` **endif**

if $m_j = m_{ij}$: discard r_j from `R1` **endif**

endif

endfor

add non discarded rectangles of `R1` to `R`

endwhile

add `R2` to `R`

Script 30.1 - Pseudo-code of the Chein algorithm

9. This version of the Chein algorithm is a bit different from that that can be found in the literature, where the matching pairs are built from undiscarded rectangles of `R1`. Unfortunately, this algorithm is erroneous. A rectangle that has just been tagged as 'discarded' cannot be ignored since it may still generate other matching rectangles.

30.6 An SQL implementation of the *Chein* algorithm

Rewriting the Chein algorithm into an SQL procedure would be a very bad idea. SQL being a *set-oriented* language, a modification query (`update`, `delete`, `insert-select`) is designed to work at its best on a set of rows, not on individual rows. Such an SQL algorithm, qualified in the database jargon of *row-at-a-time*, is likely to be a variant of the Python procedure in which each access to, and modification of, `R1` and `R2` rows has been replaced by a single-row SQL query, which would be terribly inefficient!

Converting the *row-at-a-time* algorithm into a set-oriented algorithm is fairly straightforward provided we look at the problem from a slightly different angle.

Within a definite level, we will have to carry out four basic operations:

1. extracting matching pairs from `R1` to form new rectangles and storing them into `R2`
2. updating the rectangles of `R2` that have the same attribute part as some new rectangles
3. discarding the rectangles of `R1` that were one of the two sources of some new rectangle of `R2`
4. saving the undiscarded rectangles of `R1` in `R`.

The main difference between procedural and SQL algorithms is that the processing of matching pairs, which is **immediate** as soon as each of them has been discovered in the procedural algorithm, is **delayed** until all of them have been discovered in the set-oriented algorithm. Both versions yield the same result.

Before developing the script based on the set-oriented algorithm, we must discuss the best way to represent and process sets of values in SQL.

30.6.1 Operations on sets in SQL

Let us recall that a rectangle is defined as a pair of sets: a set of objects and a set of attributes. The Chein algorithm applies three operations on these sets:

- intersection (of attribute sets in R1)
- union (of object sets in R1 and R2)
- checking equality (of attribute sets in R2 or between R2 and R1)

SQL provides native set operators to derive the union, the intersection and the difference of two sets.

Let us consider that the relation `I` of the source context is implemented by the table OBJ_has_ATT(ObjID,AttID). The set of attributes of object `'o'` is expressed by:

```
select AttID from OBJ_has_ATT where ObjID = 'o'
```

and the intersection of the attribute sets of objects `'o1'` and `'o2'`:

```
select AttID from OBJ_has_ATT where ObjID = 'o1'
   intersect
select AttID from OBJ_has_ATT where ObjID = 'o2'
```

Finally, objects `'o1'` and `'o2'` can be merged if:

```
exists(select AttID from OBJ_has_ATT where ObjID = 'o1'
         intersect
       select AttID from OBJ_has_ATT where ObjID = 'o2')
```

Now, extracting the pairs of objects that can be merged is straightforward:

```
select O1.ObjID, O2.ObjID
from  OBJ_has_ATT O1, OBJ_has_ATT O2
where exists(select AttID from OBJ_has_ATT
             where  ObjID = O1.ObjID
               intersect
             select AttID from OBJ_has_ATT
             where  ObjID = O2.ObjID);
```

This query is fine to cope with individual objects but does not help clearly if we want to process rectangles: objects have a unique identifier (ObjID) but rectangles have no obvious way to uniquely identify them. For examples, these rectangles are distinct:

```
('o1','o2')*('a1','a2') and ('o1','o2')*('a1','a2','a3')
```

similarly, these ones are also distinct:

```
('o1','o2')*('a1','a2') and ('o1','o2','o3')*('a1','a2')
```

So, in general, neither the object part nor the attribute part alone can be used as rectangle identifier. We will experience similar problems with the union of sets.

Checking the equality of two sets is a bit tricky. Given sets S1 and S2, one of the most popular equality expression checks two properties: there is no element of S1 that is not in S2, and S1 and S2 have the same size. Let us check if table `OBJ_has_ATT` includes objects with the same set of attributes:

```
select distinct O1.ObjID,'equal',O2.ObjID
from   OBJ_has_ATT as O1,
       OBJ_has_ATT as O2
where  O1.ObjID < O2.ObjID
and  not exists (select *
                 from   OBJ_has_ATT
                 where  ObjID = O1.ObjID
                 and    AttID not in (select AttID
                                      from   OBJ_has_ATT
                                      where  ObjID = O2.ObjID))
and  (select count(*) from OBJ_has_ATT where ObjID = O1.ObjID)
   = (select count(*) from OBJ_has_ATT where ObjID = O2.ObjID);
```

The query examines the self-join of OBJ_has_ATT (denoted by aliases O1 and O2) and extracts its ObjID values if:

- these values are in increasing order, an optimization that ensures that (1) no object is compared to itself and (2) if two objects are equivalent, only one of them is provided,
- all the attributes of object O1 also are attributes of object O2,
- objects O1 and O2 have the same number of attributes.

This query has a theoretical complexity of $O(N^3)$,[10] where N is the size of OBJ_has_ATT, which is not particularly scalable. Executed on table OBJ_has_ATT with 660 rows (60 objects with exactly 11 attributes), this query runs in 31 s., to find the single equality. However, an index created on column ObjID allows this time to drop to 3 s, which still is very high.

Solving a basic problem: identifying a set

Basically, the problem lies in finding a convenient way to identify a set of values among a set of such sets. A particular aspect of this problem is to determine whether two sets are the same, that is, whether each element of one of them also belongs to the other one. So, we must find a unique value[11] that is representative of a set and that unambiguously denotes this set among other similar sets.

To clarify the discussion, we call *value-set* any set of values and **S** the set of *value-sets*. In addition, we admit that the values of a set can be ordered, which is the case of character, numeric and date/time values for instance. By definition, all the value-sets are distinct in **S**.

We distinguish two cases.

- The sets in **S** are disjoint. So, any element of a value-set can be used to identify it. We just need a simple rule to unambiguously select this value, for instance, the first one in ascending order within the value-set.
- The sets in **S** are not disjoint. The only reliable identifier of a value-set is its own composition or a bijective function of it, such as its secure hash[12]. To allow scalar operations on value-sets, we choose to derive a character string from the composition of each set by concatenating the ordered values of the set, separated by a character that cannot appear in the values.

  Consider for example attribute set {`Emp,Nam,Lev`} that we represent by value-set {`'Emp','Nam','Lev'`}. If no attribute name may include symbol `','`, this value-set can be represented by character string `'Emp,Nam,Lev'`. Comparing two value-sets represented in this way is not easy. For instance, one must determine that identifiers `'Emp,Nam,Lev'` and `'Nam,Emp,Lev'` denote

10. The table is joined twice with itself.
11. Also called the *signature* of the set.
12. Hashing techniques and their properties are discussed in the case study *Blockchains*.

the same set. A simple way to solve this problem is to order the values in the identifier. So, both identifiers are converted into `'Emp,Lev,Nam'` and denote the same set.

Since the sets we will compare in FCA algorithms generally are not disjoint, we will adopt the second representation technique that we will call *string-list*.

Applying the latter conventions, we translate the rectangles of Figure 30.8 into couples of set identifiers as follows:

```
('MAN','Emp,Dep,Lev,Nam')
('ENG','Emp,Lev,Nam,Sal,Spe')
('SEC','Emp,Lan,Nam,Sal')
('ENG,MAN','Emp,Lev,Nam')
('ENG,SEC','Emp,Nam,Sal')
('ENG,MAN,SEC','Emp,Nam')
```

Set operations revisited

The identifier of a set as a *string-list* explicitly provides all the content of the set, in such a way that it can be used instead of the set itself. As a consequence we can base the FCA implementation on this set representation. All we need is a collection of specific set operators applied on these representations.

SQL does not include standard string manipulation functions that would easily simulate such operators as union and intersection applied to *string-lists*[13] Fortunately, all RDBMS allow users to define custom functions, the so-called UDF (for *User-Defined Function*). SQLfast offers a set of *string-list* manipulation UDF. We will use the following (`s`, `s1`, `s2` are *string-lists*; `sep` is the separator symbol):

- `group_concat2(s,unique,o,dir,sep)`: replacement for the native elementary `group_concat` aggregate function of SQLite. Values `v` are concatenated, separated by symbol `sep`. Parameter `unique` specifies the `distinct` modifier (`0` = duplicates are preserved, `1` = they are discarded). Parameter `o` defines the order key and `dir` specifies the order direction (`1` = ascending, `2` = descending). This form is equivalent to MySQL `group_concat` function.
- `itemUnion(s1,s2,sep)`: returns a *string-list* of all the distinct values belonging to `s1` or `s2` (or both).
- `itemInter(s1,s2,sep)`: returns a *string-list* of all the distinct values belonging to `s1` and `s2`.
- `itemLen(s,sep)`: returns the number of values of `s`.

13. Some SQL engines offer column types structured as arrays of values. The technique explained here, through UDF, can be implemented in all engines.

30.6.2 SQL implementation of the *Chein* algorithm

The algorithm proceeds in seven steps, named S1 to S7, among which steps S3 to S7 create the current level and are iterated until all the concepts have been extracted.

S1: Creating the data structures

The table OBJ_has_ATT(ObjID,AttID) will receive the couples of relation I(G,M). Sets R1, R2 and R are implemented as the eponymous tables R1, R2 and R, in which each row describes a rectangle through the columns G and M that store the *object part* and the *attribute part* of rectangles, both coded in *string-list* format. The table R1 also comprises the column Disc, that indicates whether the rectangle is discarded. The table R2 includes two additional columns, Source1 and Source2, that reference, through their G part, the source rectangles in R1[14]. The last table R3 will be justified later.

```
create table OBJ_has_ATT (ObjID,AttID);
create table R1 (G,M,Disc);
create table R2 (G,M,Source1,Source2);
create table R3 (G,M,Disc);
create table R (G,M);
```

Script 30.2 - Creating the data structures (column types ignored)

S2: Initializing the process

The relation I(G,M) of the context is loaded in the table OBJ_has_ATT. Then, the query of Script 30.3 creates the initial state of the table R1 by storing the rectangles converted from OBJ_has_ATT into *string-list* format.

```
insert into R1
   select ObjID,group_concat2(AttID,1,AttID,1,',',),0
   from   OBJ_has_ATT group by ObjID;
```

Script 30.3 - Initialization: loading in R1 the rectangles of size 1 (one rectangle for each object)

S3: Searching R1 for matching pairs

This is the critical operation of the current level. For each pair (rec1,rec2) of rectangles from R1, the M parts of which have some attributes in common (i.e., rec1.M ∩ rec2.M is not empty), a new rectangle is inserted into R2. The G part of the

14. It may be surprising that we adopt the G part of rectangles to identify them, which seems to contradict the discussion on rectangle identification developed above. Actually, the Chein algorithm ensures that all the rectangles in R1 have distinct G parts.

source rectangles rec1 and rec2 are stored in columns Source1 and Source2 for further use.

The condition `rec1.G < rec2.G` ensures that each pair is examined only once.

```
insert into R2
   select itemUnion(rec1.G,rec2.G,','),
          itemInter(rec1.M,rec2.M,',') as MM,
          rec1.G,
          rec2.G
   from   R1 rec1,
          R1 rec2
   where  MM <> ''
   and    rec1.G < rec2.G;
```

Script 30.4 - Searching R1 for pairs of rectangles that match

S4: Extending rectangles in R2

Now, we analyze the rectangles of R2 to identify those that can be merged. All the rectangles that share the same M parts are replaced with a new rectangle the G part of which is the union of the G parts of these source rectangles.

Instead of updating the rectangles in R2 (deleting the source rectangles and inserting the new one), a supposedly costly operation, we store in table R3 the merged rectangles as well as the other rectangles that need not to be merged (Script 30.5). The query builds groups of rectangles with the same M value and concatenates the G parts within each group. We observe that there is no need to explicitly cope with *single* rectangles: they just form groups with one element only and are automatically copied to R3.

```
insert into R3
   select group_concat2(G,1,G,1,',') as "G",M,0
   from   R2
   group by M;
```

Script 30.5 - Merging the rectangles of R2 with the same attribute part

S5: Discarding rectangles in R1

For each rectangle g in table R2, we have memorized the G part of the source rectangles (columns Source1, Source2). If one of the latter has the same M part as g, it is marked as discarded.

```
update R1
set   Disc = 1
where (G,M) in (select Source1,M from R2
                  union
                select Source2,M from R2);
```

Script 30.6 - Marking in R1 the rectangles that can be discarded

S6: Saving the maximal rectangles

The rectangles of R1 that have not been marked are saved in table R.

```
insert into R
       select G,M
       from R1
       where Disc = 0;
```

Script 30.7 - Saving in R the maximal rectangles of R1

S7: Preparing the next iteration

The consolidated rectangles of R3 are stored in R1 then tables R2 and R3 are cleared. The iteration of the next level can then start.

```
delete from R1;
insert into R1
       select * from R3;
delete from R2;
delete from R3;
```

Script 30.8 - Preparing tables R1, R2 and R3 for the next step

The complete algorithm

Script 30.9 shows the final algorithm assembled from Steps S2 to S7[15].

15. We note that the last operation `"add R2 to R"` of the code of 30.1 has not been translated in this script. Actually, this operation is implicitly performed by the last iteration.

```
insert into R1
   select ObjID,group_concat2(AttID,1,AttID,1,',',),0
   from   OBJ_has_ATT group by ObjID;
while (True);
   extract N1 = select count(*) from R1;
      if ($N1$ = 0) exit;
   insert into R2
      select itemUnion(rec1.G,rec2.G,',',),
             itemInter(rec1.M,rec2.M,',') as MM,
             rec1.G,rec2.G
      from   R1 rec1, R1 rec2
      where  MM <> ''
      and    rec1.G < rec2.G;
   insert into R3
      select group_concat2(G,1,G,1,',') as "G",M,0
      from   R2
      group by M;
   update R1
   set Disc = 1
   where (G,M) in (select Source1,M from R2
                     union
                   select Source2,M from R2);
   insert into R select G,M from R1 where Disc = 0;
   delete from R1;
   insert into R1 select * from R3;
   delete from R2;
   delete from R3;
endwhile;
```

Script 30.9 - The final SQL algorithm

Step S7 revisited

As it is formulated, step S7 is not particularly elegant. Instead of copying the rows of R3 in table R1, an operation that may be costly, we could simply exchange their names, at no cost.

Let us call R1 and R3 two variables that contain the physical names of the tables that play the roles of tables R1 and R3 in each iteration. In the first iteration, the variable R1 denotes the table R1 and R3 the table R3. In the second iteration, the variables are swapped: the variable R1 denotes the table R3 (therefore avoiding copying the rectangles of R3 into R1) and R3 the table R1. And so on in the following iterations. This is shown in Script 30.10.

```
insert into R1
   select ObjID,group_concat2(AttID,1,AttID,1,',',),0
   from   OBJ_has_ATT group by ObjID;
set R1,R3 = R1,R3;
while (True);
   extract n1 = select count(*) from $R1$;
      if ($n1$ = 0) exit;
   insert into R2
      select itemUnion(rec1.G,rec2.G,',',),
             itemInter(rec1.M,rec2.M,',') as MM,
             rec1.G,rec2.G
      from   $R1$ rec1, $R1$ rec2
      where  MM <> ''
      and    rec1.G < rec2.G;
   insert into $R3$
      select group_concat2(G,1,G,1,',') as "G",M,0
      from   R2
      group by M;
   update $R1$
      set Disc = 1
      where (G,M) in (select Source1,M from R2
                      union
                      select Source2,M from R2);
   insert into R select G,M from $R1$ where Disc = 0;
   delete from $R1$;
   delete from R2;
   set R1,R3 = $R3$,$R1$;
endwhile;
```

Script 30.10 - Improved manipulation of tables R1 and R3 in successive iterations

It is important to note that the Chein algorithm associates with each object not only its own attributes, but also all its inherited attributes. This remark will make sense in the comparison between the different implementations of the algorithm.

30.7 A Python implementation of the *Chein* algorithm

Though the focus of this study is on the use of database concepts and tools to solve a wide variety of problems, it is interesting to compare this approach to more traditional problem solving paradigms, in particular those that rely on procedural languages. To this aim, we have written a Python procedure, FCA_Engine.py, which is a direct translation of the pseudo-code of Script 30.1[16].

We consider that the rectangles of sets R1, R2 and R can be indexed, that is, they can be accessed by their rank in their set, according to an arbitrary but deterministic order. The `for-endfor` loop is then developed into two embedded loops:

```
for j in range(1,len(R1)):
    # get rectangle rj
    for i in range(0,j):
        # get rectangle ri
        # examine and process the couple (ri,rj)
```

The main design decision is the way the sets R1, R2 and R are implemented. The first and simplest idea that comes in mind is to represent them by *lists*. R1 and R2 become lists of (g,m,d) tuples and R a list of (g,m) couples, where g is the object string-list, m the attribute string-list and d the discard indicator.

The structure of the `while-endwhile` loop can then be translated as shown in Script 30.11.

```
# list R2 is initialized with the source object:
while len(R2) > 1:
    R1 = R2
    R2 = []
    for j in range(1,len(R1)):
        (gj,mj,d) = R1[j]
        for i in range(0,j):
            (gi,mi,d) = R1[i]
            gij = union(gi,gj)
            mij = inter(mi,mj)
            if mij == '':
                continue
            gk,ik = getRectOfM(R2,mij)
            if gk is not None:
                R2[ik] = (union(gk,gij),mij,0)
            else:
                R2.append((gij,mij,0))
            if mi == mij:
                R1[i] = (gi,mi,1)
            if mj == mij:
                R1[j] = (gj,mj,1)
    addUndiscarded(R,R1)
if len(R2) > 0:
   addUndiscarded(R,R2)
```

Script 30.11 - The Chein algorithm - A Python implementation (excerpts)

This code calls these utility functions:

– `union(s1,s2)`: returns (as a string-list) the union of the elements of string-lists `s1` and `s2`.

16. Actually, FCA_Engine.py is a module that comprises a collection of different implementations of the Chein algorithm. More on this in Section 30.12 (*Improving the performance of the algorithms*).

- `inter(s1,s2)`: returns (as a string-list) the common elements of string-lists `s1` and `s2`.
- `getRectOfM(L,m)`: search list `L` for the rectangle whose attribute list is `m`; returns its rank in `L` and its object part; if none found, returns `-1` and `None`.
- `addUndiscarded(L1,L2)`: add to list `L1` the rectangles of `L2` that have not been discarded (`d == 0`).

30.8 Building the concept hierarchy

Building the concept hierarchy by extracting the inclusion relation of the G parts of the concepts has been described in Section 30.4.2. We translate the two steps of the procedure in SQL then in Python.

30.8.1 An SQL implementation

The first step builds in the table ISA the inclusion relation among the G parts of the concepts of table R. The first two columns, Sub and Super, store the G part of the subconcept and the superconcept respectively and the third column, Trans, will be explained later (Script 30.12). Considering two *distinct*[17] rows sub and sup of R, sub is a subconcept of sup if `sup.GE` includes `sub.G`, which is expressed by the UDF `itemInclude(s1,s2,sep)`, where `s1` and `s2` are string-lists and `sep` their value separator.

```
create table ISA(Sub,Super,Trans);
insert into ISA
   select sub.G,sup.G,0
   from   R sub, R sup
   where  itemInclude(sup.G,sub.G,',')
   and sub.G <> sup.G;
```

Script 30.12 - Building into the table ISA the inclusion relation among the concepts of R

Now ISA contains the *closure* of the inclusion relation. To clean it up, we need to identify and discard the transitive couples. This is the purpose of the query of Script 30.13. For each row **r** in ISA, we check whether there are two other rows whose composition is equal to **r**. If we find such rows, then **r** is transitive and should be ignored, which is specified by setting its column `Trans` to `1`. Finally, we delete the rows for which `Trans = 1`.

17. G is an implicit unique key of R (easy to prove), so, the condition `sub.G <> sup.G` ensures that equality is discarded.

```
update ISA
set    Trans = 1
where exists (select *
              from  ISA I1, ISA I2
              where I1.Sub = ISA.Sub
              and   I1.Super = I2.Sub
              and   I2.Super = ISA.Super);
delete from ISA where Trans = 1;
```

Script 30.13 - Computing the transitive reduction of the hierarchy (SQL version)

30.8.2 A Python implementation

The Python translation shown in Script 30.14 derives the concept hierarchy from the concepts whose definitions are stored in list `R` (built by Script 30.11 for example). The first part stores in list `rawInclusion` the transitive closure of the inclusion relation. The second part examines each couple `(con1,con2)` of `rawInclusion`: if `R` includes another concept `con3` such that both `(con1,con3)` and `(con3,con2)` exists in `rawInclusion`, this couple is transitive and is discarded.

```
rawInclusion = []
for con1 in R:
    for con2 in [con[0] for con in Concepts
                 if con[0] <> con1[0]]:
        if included(con1[0],con2):
            rawInclusion.append((con1[0],con2))

hierarchy = []
for (con1,con2) in rawInclusion:
    status = 'basic'
    for con3 in [con[0] for con in R
                 if con[0] <> con1 and con[0] <> con2]:
        if (con1,con3) in rawInclusion
            and (con3,con2) in rawInclusion:
            status = 'transitive'
            break
    if status == 'basic':
        hierarchy.append((con1,con2))
```

Script 30.14 - Computing the transitive reduction of the hierarchy (Python version)

30.9 An alternative FCA implementation: *ISA recovery*

The literature provides more than a dozen concept extraction techniques, among which the *Chein family*, that formalizes an intuitive manual procedure (see Section 30.4), is the most used in FCA tutorials.

We will now describe an alternative technique that has been applied successfully in the field of database engineering, in particular to *database reverse engineering*[18]. Its objective is to help recover the conceptual schema of a legacy database. Starting from the schema of an existing, sometimes several decades old, database or set of files, this technique identifies a hierarchy of entity types that represents its semantics. It provides a set of entity types (the other name of concepts in the database vocabulary) but is much simpler and faster than Chein's algorithm(s). However, it provides different results, closer to what a database developer expects from a normalized schema. Since it still need to compute the transitive reduction of its inclusion relation (Section 30.8) to extract the ISA hierarchy, it has been called *ISA recovery.*

30.9.1 The basic algorithm

Let us consider the Object/Attribute matrix of Figure 30.7 and pivot it to produce the equivalent matrix of Figure 30.14.

	MAN	ENG	SEC
Emp	x	x	x
Nam	x	x	x
Dep	x		
Lev	x	x	
Sal		x	x
Spe		x	
Lan			x

Figure 30.14 - Another presentation of the matrix of Figure 30.7

Instead of creating a rectangle from *each source object*, as we did in step S2 of Section 30.6.2, we create a rectangle from *each attribute row* of this matrix. These rectangles associate with each attribute the set of objects in which it appears. This operation materializes the **obj(B)** operator described in Section 30.2. Then we merge the rectangles that share the same set of objects, e.g., those built from Emp and Nam. By construction, these rectangles are maximal and therefore form a set of concepts of the source context (Figure 30.15).

18. See [Hainaut, 2002] for example.

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN)x(Dep)
(MAN,ENG)x(Lev)
(ENG,SEC)x(Sal)
(ENG)x(Spe)
(SEC)x(Lan)
```

Figure 30.15 - The concepts extracted from the context of Figure 30.14

We build the concept hierarchy as described in Section 30.4. The result is shown in Figures 30.16 and 30.17.

```
(MAN)x(Dep)       subtype of  (MAN,ENG)x(Lev)
(ENG)x(Spe)       subtype of  (MAN,ENG)x(Lev)
(ENG)x(Spe)       subtype of  (ENG,SEC)x(Sal)
(SEC)x(Lan)       subtype of  (ENG,SEC)x(Sal)
(MAN,ENG)x(Lev)   subtype of  (MAN,ENG,SEC)x(Emp,Nam)
(ENG,SEC)x(Sal)   subtype of  (MAN,ENG,SEC)x(Emp,Nam)
```

Figure 30.16 - Transitive reduction of the subtype/supertype relation

It is important to note that the procedure associates with each object only its proper attributes.

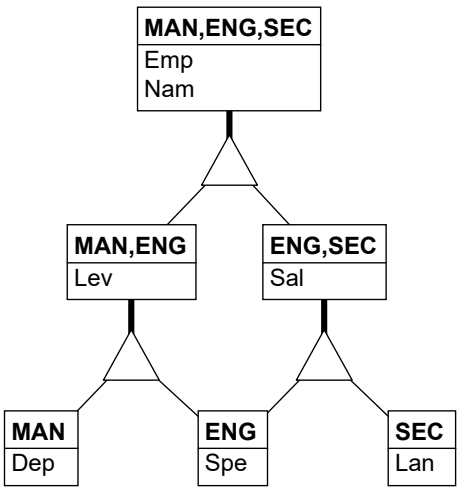


Figure 30.17 - The concept hierarchy derived from the context of Figure 30.7

30.9.2 An SQL translation of the *ISA recovery* algorithm

To implement this technique in SQL, we first define the appropriate data structures (Script 30.15).

```
create table OBJ_has_ATT(ObjID,AttID);
create table CONCEPT(G,M);
```

Script 30.15 - *ISA recovery*: the data structures (column types ignored)

Then, we extract the maximal rectangles in two steps: for each attribute, we compute the list of its objects (the `from` subquery in Script 30.16)[19], thus building elementary rectangles, then we merge the AttID parts of those of these rectangles that have the same G part. These maximal rectangles are stored in the CONCEPT table.

This procedure, particularly simple and intuitive, will be discussed and compared with the *Chein* algorithm in a later section.

30.9.3 The case of empty concepts

The basic algorithm may, in certain circumstances, create or discard concepts that have no proper attributes. We will discuss two different cases.

```
insert into CONCEPT
   select G,group_concat(AttID)
   from (select group_concat2(ObjID,1,ObjID,0,',') as G, AttID
         from   OBJ_has_ATT
         group by AttID)
   group by G;
```

Script 30.16 - Compute the concepts from the maximal rectangles in OBJ_has_ATT

Preserving the source objects with no proper attributes

If we examine in more detail the composition of the matrix of Figure 30.14, we observe that it assigns each source object (`MAN`, `ENG`, `SEC`) at least one proper attribute. The consequence is that all these source objects appear in the final set of concepts and, more explicitly, in the hierarchy of Figure 30.17. Let us suppose that we remove the attribute `Spe`, so that the attribute set of `ENG` is reduced to (`Emp,Nam,Lev,Sal`).

Applying the procedure of Script 30.16, we observe that `ENG` has disappeared. It is included in higher level aggregated concepts but no longer exists as a standalone concept:

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN)x(Dep)
(MAN,ENG)x(Lev)
(ENG,SEC)x(Sal)
```

19. We have used the extended version of `group_concat` to ensure that the components of the object lists are sorted, so that they can be *`grouped by`* correctly in the outermost query.

```
(SEC)x(Lan)
```

The new concept hierarchy is shown in Figure 30.18.

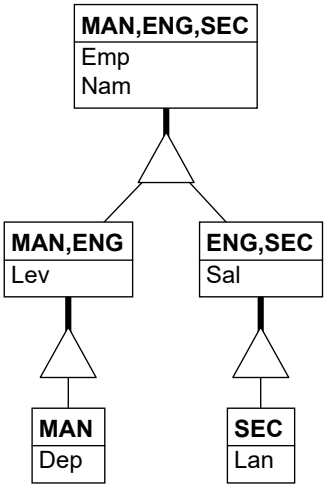


Figure 30.18 - Devoid of any proper attributes, the source object ENG disappears

Whether or not the disappearance of such source objects is desirable depends on the objective of the FCA. In the database domain, source objects most often are meaningful data sets, such as tables, object classes, table types or entity types, that are to be used in application programs. The FCA process consists in normalizing a source schema by eliciting hidden pertinent data sets without getting rid of the source objects, considered the very first concepts of the solution.

To preserve these *empty* concepts, we just add them to the CONCEPT table, as shown in Script 30.17.

```
insert into CONCEPT
   select G,group_concat(AttID)
   from (select group_concat2(ObjID,1,ObjID,0,',') as G, AttID
         from   OBJ_has_ATT
         group by AttID)
   group by G
      union
   select distinct ObjID,''
   from   OBJ_has_ATT
   where  ObjID not in (select G from CONCEPT);
```

Script 30.17 - Extended concept extraction: all the source concepts are preserved

The concept hierarchy now includes these *empty* concepts (Figures 30.19 and 30.20). It must be noted that the Chein algorithm also discards these source empty concepts.

```
(MAN,ENG,SEC)x(Emp,Nam)
(MAN,ENG)x(Lev)
(ENG,SEC)x(Sal)
(MAN)x(Dep)
(ENG)x()
(SEC)x(Lan)
```

Figure 30.19 - Reintroduction of empty source concepts

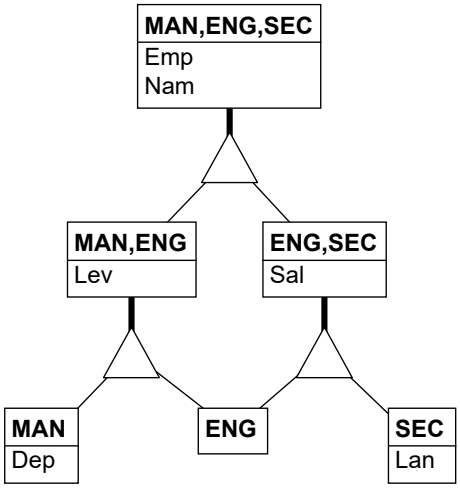


Figure 30.20 - Empty source concepts can also be represented in the concept hierarchy

Empty intermediate concepts

We consider the formal context comprising objects `MAN`, `ENG` and `SEC`, from which we have built the concept hierarchy of 30.17. Now, we add a new object to this context, ASSIST, that represents *technical assistants*, described by four attributes: `Emp`, `Nam`, `Lev` and `Sal`. The modified concept hierarchy is shown in Figure 30.21.

This hierarchy exhibits a pattern that we have not yet encountered: two concepts, `ENG` and `ASSIST`, have the same two superconcepts, `MAN,ENG,ASSIST` and `ENG,ASSIST,SEC`. More generally, *two or more concepts have at least two direct superconcepts in common*.[20]

This pattern suggests the existence of an intermediate concept, here `ENG,ASSIST`, which depends on the two superconcepts and on which the two subconcepts depend. In a sense, this new concept tells that `ENG` and `ASSIST` have a common characteristic, that is, to depend on the same set of superconcepts. It *materializes* the common

20. Or, equivalently, *a set of at least two superconcepts share the same set of at least two direct subconcepts*.

subconcept-superconcept relationship. Naturally, this concept has no proper attribute. This transformation is illustrated in Figures 30.22 and 30.23.

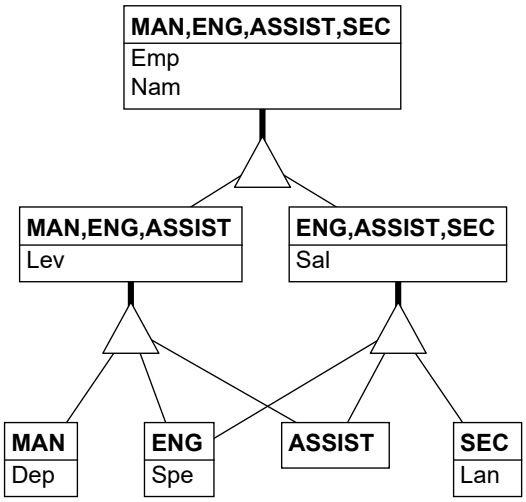


Figure 30.21 - ENG and ASSIST share the same set of superconcepts

```
(MAN,ENG,ASSIST,SEC)x(Emp,Nam)
(MAN,ENG,ASSIST)x(Lev)
(ENG,ASSIST,SEC)x(Sal)
(ENG,ASSIST)x()
(MAN)x(Dep)
(ENG)x(Spe)
(ASSIST)x()
(SEC)x(Lan)
```

Figure 30.22 - Introduction of intermediate concept ENG,ASSIST

It must be noted that the Chein algorithm automatically creates the empty intermediate concepts, whenever the pattern condition is met.

Here again, whether this transformation has to be performed depends on the application domain. In a database schema, where developers tend to avoid too deep inheritance hierarchies, considered less readable, the solution of Figure 30.21 will probably be preferred. On the contrary, when extracting the most pertinent concepts from a large data set, the users might appreciate the elicitation of such concepts as ENG,ASSIST in Figure 30.23.

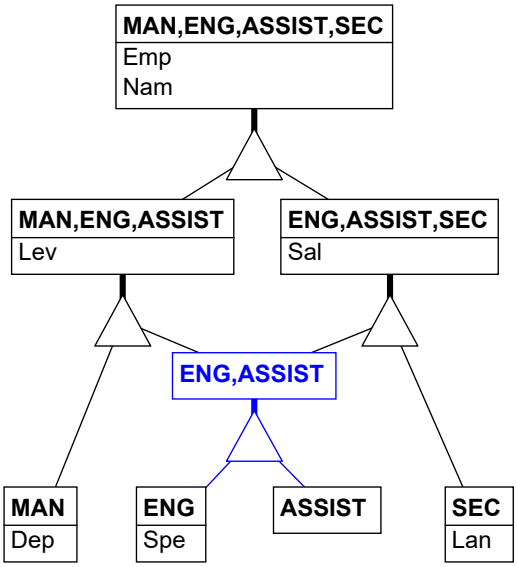


Figure 30.23 - A new empty concept makes explicit the common set of superconcepts of ENG and ASSIST

Creating the empty intermediate concepts

Let us suppose that we want to identify and create these intermediate concepts. To develop such a procedure, we will illustrate its steps by the abstract example of Figure 30.24, in which all the concepts comprise (at least) an attribute. This hierarchy has been built by the procedure developed in Section 30.8.

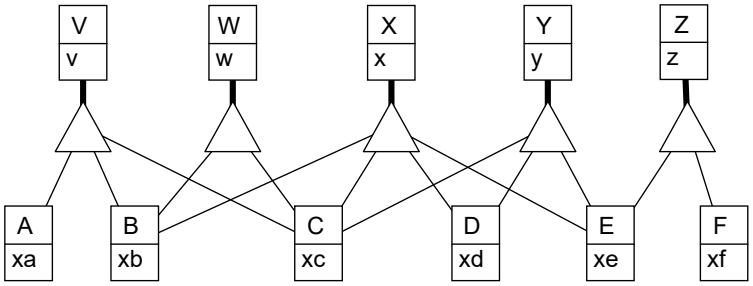


Figure 30.24 - An abstract hierarchy without empty intermediate concepts

Its fourteen *subconcept-superconcept* links are initially stored in table ISA(Sub,Super)[21]. The content of ISA is shown in Figure 30.25. It is produced by the query 30.18[22]:

21. We ignore the column Trans, now useless.

```
select Sub,group_concat(Super,';') as Supers
from   ISA
group by Sub;
```

Script 30.18 - Showing the content of table ISA

Sub	Supers
A	V
B	V;W;X
C	V;W;X;Y
D	X;Y
E	X;Y;Z
F	Z

Figure 30.25 - Synthetic view of table ISA

Now, we select the concepts (column Sub) that depend on *at least two superconcepts* (column Super). Concepts A and F are discarded (Script 30.19 and Figure 30.26).

```
create view ISA2(Sub,Super)
as select Sub, group_concat(Super,';')
   from   ISA
   group by Sub
   having count(*) >= 2;
```

Script 30.19 - Selecting the concepts depending on 2+ superconcepts

Sub	Super
B	V;W;X
C	V;W;X;Y
D	X;Y
E	X;Y;Z

Figure 30.26 - The four concepts that depend on 2+ superconcepts (view ISA2)

We compute in ISA2 the intersection of each pair of sets of superconcepts. If this intersection comprises at least two superconcepts, the subconcepts are concatenated, otherwise, the pair is dropped (Script 30.20 and Figure 30.27).

22. The choice of the `';'` separator preserves the identity of superconcepts when their name is made up of more than one component. On the contrary, subconcepts are merged (without duplicates), which justify the `','` separator, the same as that used in the concept names.

```
create view PAIR(Subs,Supers)
as select I1.Sub||','||I2.Sub,
          itemInter(I1.Super,I2.Super,';') as Supers
   from   ISA2 I1,ISA2 I2
   where  I1.Sub < I2.Sub
   and    itemLen(Supers,';') >= 2;
```

Script 30.20 - Merging pairs of concepts that share at least two superconcepts

Subs	Supers
B,C	X;W;V
C,D	Y;X
C,E	Y;X
D,E	Y;X

Figure 30.27 - Pairs of concepts sharing a set of at least two superconcepts (view PAIR)

Finally, we merge the sets of subconcepts that share the same set of superconcepts (Script 30.20 and Figure 30.27). For instance, three concept sets `C,D`, `C,E` and `D,E` are merged into `C,D,E` (Figure 30.28). The updated concept hierarchy is shown in Figure 30.29.

```
select itemSort(group_concat(Subs,','),0,1,',') as NewConcept,
       Supers as SuperConcepts
from  PAIR
group by Supers;
```

Script 30.21 - Final step: merging the set of subconcepts sharing the same set of superconcepts

NewConcept	SuperConcepts
B,C	X;W;V
C,D,E	Y;X

Figure 30.28 - The two new intermediate concepts and their superconcepts

These script fragments can be wrapped into a single query. This is left as an exercise.

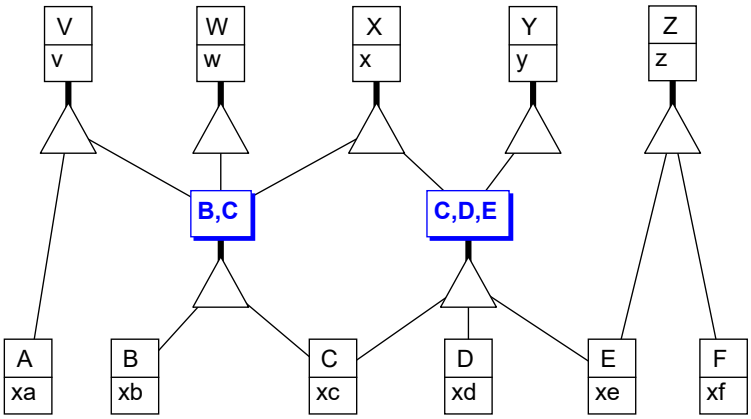


Figure 30.29 - Introducing two intermediate empty concepts

Note

The introduction of intermediate concepts can again create the conditions for the generation of additional intermediate concepts. It is therefore recommended to iterate the process until no additional concepts are created.

30.9.4 Python translation of the *ISA recovery* algorithm

Due to its simplicity, the SQL script 30.17 is easily translated into a small Python program like the one shown in Script 30.22.

```
def groupConcat(T,A,G):
    UG = list(set([t[G] for t in T]))
    AG = []
    for g in UG:
        ag = ','.join([t[A] for t in T if t[G] == g])
        AG.append((g,ag))
    return AG

innerQuery  = groupConcat(I,0,1)
Concepts = groupConcat(innerQuery,0,1)

actObj = [obj for (obj,att) in Concepts
          if len(obj.split(',')) == 1]
for obj in G:
    if obj not in actObj:
        Concepts.append((obj,''))
```

Script 30.22 - A Python translation of SQL Script 30.17

The component `I` of the context is represented by an eponymous list of couples, the first element of which is the name of an object and the second is the name of one of its attributes.

The `groupConcat` function simulates the `group_concat-group by` SQL pattern applied to a list of couples. `T` is this list, `A` is the index [0,1] of the component of the couples to aggregate and `G` is the index [0,1] of the component on which the aggregation is applied (the grouping criterion). The list `UG` comprises the unique values of the `G` components of the couples.

The first application of the function corresponds to the inner query of the SQL expression. It computes the rectangles based on each attributes. So, we aggregate the objects names (index `A = 0`) with the same attribute (index `G = 1`):

```
innerQuery = groupConcat(I,0,1)
```

The result, `innerQuery`, is a list of couples (*attribute*, *object list*). The second application merges the rectangles with the same object list. It aggregates the attribute names (index `A = 0`) with the same object list (index `G = 1`):

```
Concepts = groupConcat(innerQuery,0,1)
```

The result is that of Figure 30.15. The end of the code converts the objects with no proper attributes into concepts.

30.10 Experimentation

The algorithms developed in this study have been translated into four SQLfast scripts. They are included in the SQLfast distribution, along with a collection of sample formal contexts.

30.10.1 The scripts

Chein algorithm, Python implementation

The script **computeFCA (Python).sql** opens the control panel shown in Figure 30.30 and through which the user selects a Python procedure and the way the concept hierarchy is derived (*None*, through a *Python* algorithm, through an *SQL* script).

The Python procedure described in Section 30.7 is labelled "**1. two lists**". The other four procedures are based on alternative representations of the sets R1, R2 and R. They will be discussed in Section 30.12, devoted to performance optimization.

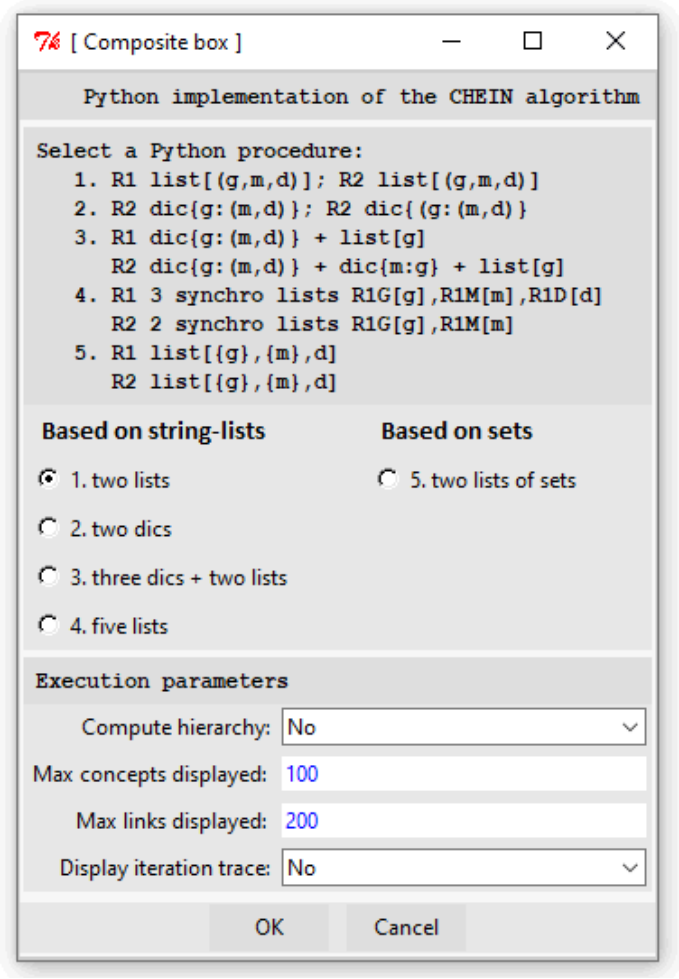


Figure 30.30 - Control panel of the Python implementation of the Chein algorithm

The script calls the **FCA_Engine.py** Python module which includes the five variants of the Chein algorithm and (optionally) the derivation of the concept hierarchy. This module has been added to the SQLfast directory.

Chein algorithm, SQL implementation

The script **computeFCA (SQL).sql** opens the control panel shown in Figure 30.31. It allows the user to specify the information to be displayed at each stage of execution and the final results. In addition, the user can evaluate the effect of various combinations of indexes on the execution time (more on this in Section 30.12.2).

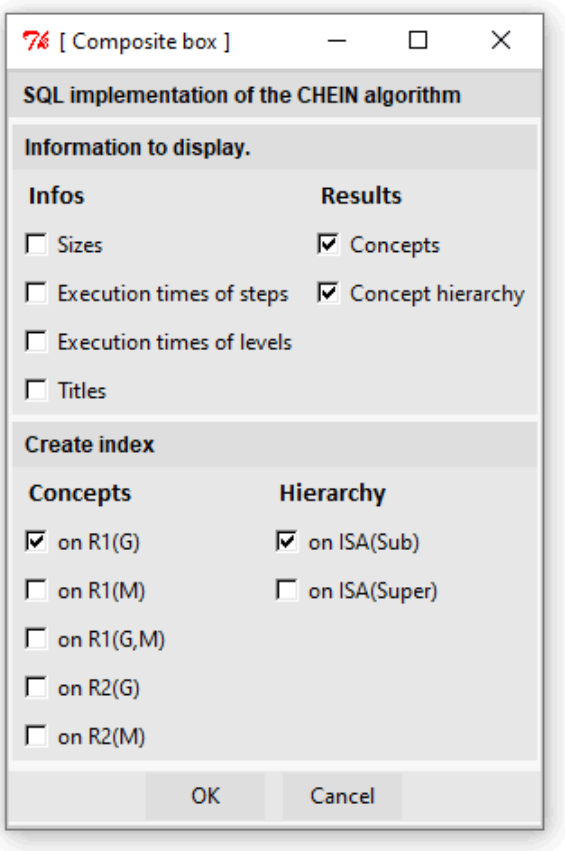


Figure 30.31 - Control panel of the SQL implementation of the Chein algorithm

ISA recovery algorithm, SQL implementation

The control panel of the script **computeISA Recovery (SQL).sql** is shown in Figure 30.32. The first frame specifies the additional concepts to include in the solution (empty source objects and empty intermediate concepts). The second frame controls the number of rows to be displayed and the maximum number of iterations when computing the intermediate concepts.

ISA recovery algorithm, Python implementation

The Python script (one of the function of the **FCA_Engine.py** module) is executed from the SQLfast script **computeISA Recovery (Python).sql**.

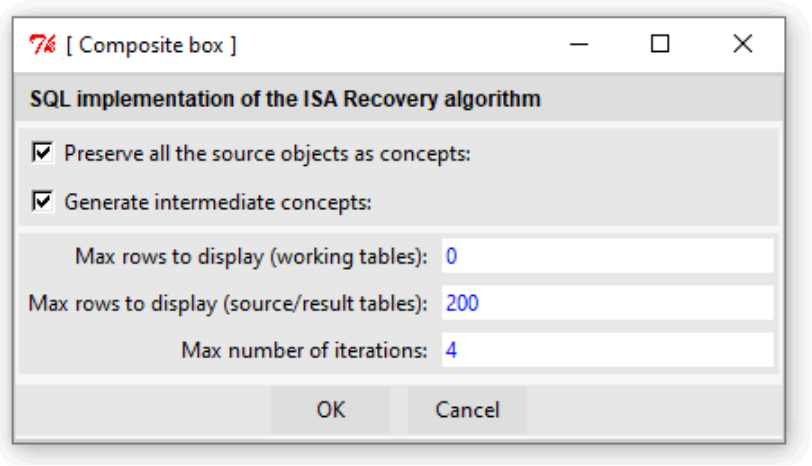


Figure 30.32 - Control panel of the ISA Recovery algorithm

30.10.2 Sample formal contexts

The literature and the tutorials available on the web provide us with many illustrative samples. However they all are quite small, comprising one or two dozen objects and not more attributes. They are fitted to play with the concepts and to illustrate the way the algorithms work, but they are less useful to compare the solution, the time performance and the space requirements of the algorithms. Some of them are included in the SQLfast distribution. They are available in the directory Scripts/Case-Studies/Case_FCA/Contexts.

– PERSONNEL

One of the first contexts used in this study.

Object	Attributes
ENG	Emp#,Name,Lev,Gra,Sal,Spe
MAN	Emp#,Name,Lev,Gra,Dep
SEC	Emp#,Name,Sal,Lang1,Lang2
TRA	Emp#,Name,Sal,Lang1

Scripts: Create-PERSONNEL.sql, FCA-PERSONNEL.csv

– ANIMAL

Seven animal species and their properties.[23]

Object	Attributes
canary	eggs,feather,fly,breath
crocodile	eggs,teeth,swim,breath
duck	eggs,feather,fly,swim,breath
frog	eggs,swim,breath
ostrich	eggs,feather,breath
salmon	eggs,swim
shark	eggs,teeth,swim

Scripts: Create-ANIMAL.sql, FCA-ANIMAL.csv

– WATERS

Description of 17 bodies of water[24].

Object	Attributes
CANAL	con,run
CHANNEL	con,run
LAGOON	con,mar,nat,sta
LAKE	con,nat,sta
MAAR	con,nat,sta
POND	con,nat,sta
POOL	con,nat,sta
PUDDLE	nat,sta,tmp
RESERVOIR	con,sta
RIVER	con,nat,run
RIVULET	con,nat,run
RUNNEL	con,nat,run
SEA	con,mar,nat,sta
STREAM	con,nat,run
TARN	con,nat,sta
TORRENT	con,nat,run
TRICKLE	con,nat,run

Scripts: Create-WATERS.sql, FCA-WATERS.csv

– ERA

Abstract physical relational schema (just tables and columns) derived from the flattening of an Entity-relationship conceptual schema. Comprises 25 objects and 52 attributes. *Question*: can the FCA algorithms recover the exact source conceptual schema?

23. Example often used in tutorials. See for example, https://ijcai-15.org/downloads/tutorials/T23-FCA.pdf

24. See extended description in https://en.wikipedia.org/wiki/Formal_concept_analysis

Object	Attributes
Q01	a1,a2,a3,d1,d2,j1,q01
Q02	a1,a2,a3,d1,d2,e1,j1,k1,k2,q02
Q03	a1,a2,a3,d1,d2,e1,k1,k2,q03
Q04	a1,a2,a3,b1,b2,d1,d2,e1,k1,k2,f1,f2,l1,q04
Q05	a1,a2,a3,b1,b2,e1,f1,f2,l1,q05
...	...
R06	a1,a2,a3,b1,b2,c1,c2,f1,f2,g1,g2,h1,h2,h3,m1,m2,r06
R07	b1,b2,c1,c2,h1,h2,h3,i1,i2,i3,o1,r07
R08	c1,c2,i1,i2,i3,p1,p2
R09	c1,c2,i1,i2,i3,p1,p2,r09
R10	c1,c2,i1,i2,i3,p1,p2,r10

Scripts: Create-ERA.sql, FCA-ERA.csv

– COUNTRY[25]

The formal context comprises 147 objects (countries) and 33 attributes (country indicators). Each object describes a country through 11 attributes. From this source, we have derived subsets of 10 to 147 countries. Below, the context of COUNTRY-10 (country names and attributes names are denoted by letters or numbers)

Object	Attributes
A	2,5,6,9,12,14,19,22,23,29,32
B	0,3,7,10,11,16,18,20,25,28,30
C	2,4,7,9,13,15,18,21,26,28,31
D	2,4,6,9,12,14,19,22,23,29,32
E	1,4,7,9,12,15,17,20,26,27,31
F	1,3,7,9,11,16,18,20,26,27,30
G	2,5,7,9,13,16,17,20,26,27,32
H	0,3,7,9,13,16,17,20,26,27,30
I	1,5,7,10,12,14,18,21,25,27,32
J	2,4,6,9,12,15,18,20,25,27,31

Scripts: Create-COUNTRY-10.sql (10 countries) to Create-COUNTRY-Full.sql (147 countries, FCA-COUNTRY-10.csv to FCA-COUNTRY-100.csv.

– Hypercard

This example is a reduced version of a real project the objective of which was to convert the data of a small Hypercard[26] application into a standard database. The

25. Derived from the data provided in https://github.com/mdaquin/fca.js, referenced by [D'Aquin, 2020]. The full source data are available in script Create-COUNTRY-Full.sql. Countries are coded as numbers from 0 to 146 (except in COUNTRY-10, where they are coded as letters) and attributes from 0 to 32. The interpretation of these codes can be found as comments in the full script.

very first step when converting a database is to extract its conceptual schema, in the form of a hierarchy of entity types. The application comprises eight files, the structure of which can be expressed as the following formal context:

Object	Attributes
BOO	Aut,Dat,ID,Key,Mon,Pub,Tit,Yea
COL	Dat,Edi,ID,Key,Mon,Pub,Tit,Yea
COLP	Aut,Dat,ID,Key,Pag,Sou,Tit
JOU	Dat,Edi,ID,Key,Mon,Num,Tit,Vol,Yea
JOUP	Aut,Dat,ID,Key,Pag,Sou,Tit
PRO	CDa,CTi,Dat,Edi,ID,Key,Mon,Pub,Tit,Yea
PROP	Aut,Dat,ID,Key,Pag,Sou,Tit
REP	Aut,Dat,ID,Key,Mon,Ori,Tit,Typ,Yea

The meaning of these abbreviations is described in the SQL code of the context.

Scripts: Create-HYPERCARD.sql, FCA-HYPERCARD.csv

30.10.3 The context generator

The context generator (Context-Generator.sql) is a small application that generates synthetic (i.e., artificial) formal contexts of up to 2,600 objects[27]. The attributes of each object are randomly selected among a set of attributes of up to the same size. The size of the attribute set of each object is a random number in a definite range.

The dialog box of Figure 30.33 sets the parameters for a formal context of 100 objects and 30 attributes. A set of 2 to 10 attributes will be associated with each object. The size and content of the attribute set of each object are randomly selected.

The description of the context is stored as an *SQL* script or as a *csv* file.

Due to the twofold random nature of the `I` relation, the contexts generated from the same set of parameters can vary, producing, for example, a different number of concepts.

30.11 Evaluation of the FCA implementations

In this section, we will comment and compare the four implementations developed here above on the basis of qualitative and quantitative criteria:

26. Hypercard is a user-friendly software tool (*"programming for the rest of us"*) that was very popular on the Macintosh in the 80's and beyond. Very powerful and remarkably simple, it allowed to build small interactive database applications in an intuitive way. [https://en.wikipedia.org/wiki/HyperCard]

27. The script includes a parameter to set the maximum to 26,000 or 260,000.

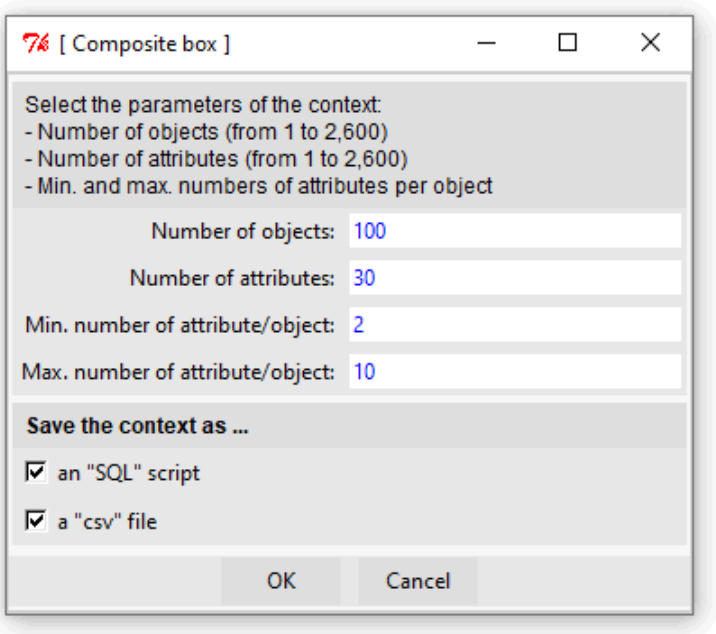


Figure 30.33 - Creating a new formal context

- Python implementation of the *Chein* algorithm
- SQL implementation of the *Chein* algorithm
- SQL implementation of the *ISA recovery* algorithm
- Python implementation of the *ISA recovery* algorithm

It should be clear that different input data sets may produce different results, at least for some metrics[28]. It would therefore be imprudent to generalize the conclusions of this section to all possible uses of the techniques studied. As usual, the reasoning is more important than the conclusions!

30.11.1 Simplicity of algorithms

Of the four FCA implementations, the SQL version of the *ISA recovery* algorithm (Script 30.17), expressed as a single intuitive query, is arguably the simplest and most natural, though some Python programmers may consider its Python translation to be equally so. The comparison between the Python and the SQL implementation of the *Chein* algorithm is a matter of taste: the Python code is shorter but the SQL

28. For example, the simplicity and source object preservation criteria do not depend on the nature of the formal context.

version which performs global and declarative set-oriented operations, may be considered more natural by some.

The comparison can change considerably when we look at *optimized versions* of the algorithms (see Section 30.12). The main technique for optimizing SQL queries does not change the queries themselves but merely adds indexes, which are external constructs that do not require table modification either. So there is no difference between the tables and the queries whether the algorithm is optimized or not. The optimized SQL solution therefore preserves its original simplicity. In contrast, optimizing a procedural program often requires changes to the data structures and algorithmic control structures such that the optimized version can be considered more convoluted and less readable.

30.11.2 Preservation of the source concepts

The *Chein* algorithm may ignore some source objects for two reasons. First, it merges the source objects with the same attribute set (step S2). Then, it discards the rectangles of R1 that have the same attribute set as some rectangles in R2 (step S5).

On the contrary, the *ISA recovery* algorithm preserves all the source objects by converting them into concepts, even those that have no proper attributes.

30.11.3 Nature of the attributes of the concepts

The *Chein* algorithm associates with each concept all its attributes, be they proper or inherited. On the contrary, the *ISA recovery* algorithm keeps their proper attributes only. When there is none, empty concepts are generated. As we have observed, they all translate source objects of the context. Unless the complementary procedure of Section 30.9.3 have been applied, each generated concept has at least one proper attribute.

Each form can be converted into the other one, either by recursively propagating the attributes of each concept to its subconcepts or by recursively deleting in each concept the attributes of its super-concepts. This is left as an exercise.

30.11.4 Creation of empty intermediate concepts

The *Chein* algorithm, through the merging step of R1 rectangles, creates new rectangles, then converts those that are maximal into concepts. Among them many intermediate concepts have no proper attributes.

The first step of the *ISA recovery* algorithm creates concepts that have proper attributes. The second step adds the concepts derived from the source objects that have no proper attributes and therefore appear empty. No empty intermediate concepts are generated. However, the complementary procedure described in Section 30.9.3 generates some empty intermediate concepts that may contribute to the readability of the final hierarchy (compare the schemas of Figures 30.24 and 30.29).

The table of Figure 30.34 shows that the number of empty intermediate concepts may be quite huge. For example, the *ISA recovery* solution of the COUTRY-50 case (labelled ISA recovery) includes 82 concepts compared to 1,639 for the *Chein* algorithm, most of them being empty intermediate concepts. Clearly, the *ISA recovery* algorithm generates less concepts than the *Chein* algorithm.

	Objects	Chein algorithm	ISA recovery
COUNTRY-10	*10*	*68*	*36*
COUNTRY-20	*20*	*299*	*50*
COUNTRY-30	*30*	*708*	*60*
COUNTRY-40	*40*	*1,187*	*72*
COUNTRY-50	*50*	*1,639*	*82*
COUNTRY-60	*60*	*2,171*	*92*

Figure 30.34 - Number of concepts generated by the algorithms. The additional empty intermediate concepts of the *ISA recovery* algorithm have not been accounted for

The figure 30.35 illustrates the same fragment of the hierarchy built from the concepts generated by both techniques.

It should be noted that each of these algorithms may generate intermediate concepts that the other does not. For example, the *ISA recovery* algorithm (through its complementary procedure), applied to the COUNTRY-10 context, generates 4 new empty intermediate concepts[29] but two of them are ignored by the *Chein* algorithm[30].

In a context of database reverse engineering, in which the physical schema of a legacy database comprises 50 tables or data files, generating a conceptual schema made up of more than 1,600 entity types has no sense[31].

In contrast, the *ISA recovery* algorithm is not suitable for discovering, analyzing and querying semantically (potentially) relevant object classes among large sets of objects.

This discussion clearly delineates the application domains of these algorithms.

30.11.5 Memory space requirements

We examine the quantity of memory space the implementation of each algorithm requires at runtime. The measurements are performed on the COUNTRY-50 context resolution.

29. CG, GH, BCI and EGHJ
30. BCI and EGHJ
31. All the more so as each concept will be, in a second phase of the database reengineering process, translated into a table!

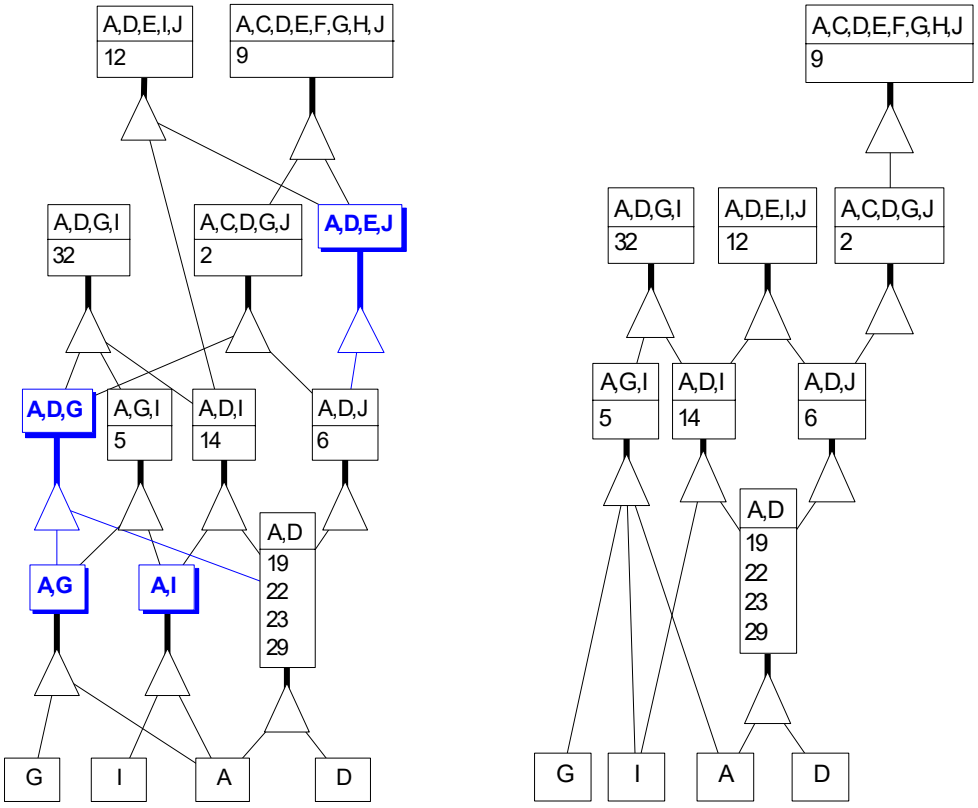


Figure 30.35 - Fragments of the COUNTRY-10 concept hierarchy generated by the standard *Chein* algorithm (left) and the *ISA recovery* algorithm (right)

Python implementation of the *Chein* algorithm

The space used by the Python solver is determined by the maximum size of the lists that implement the rectangle sets R1, R2 and R. These numbers are given in the table of Figure 30.36 at the end of each of the ten levels of the CONTRY-50 context processing. They are obtained by enabling the tracing function (see control panel of Figure 30.30). The Python solver requires 1,305 + 1,305 + 1,639 = 4,249 list elements.

	1	2	3	4	5	6	7	8	9	10
R1	50	856	1,305	955	652	407	232	119	48	10
R2	856	1,305	955	652	407	232	119	48	10	0
Disc R1	0	573	954	652	407	232	119	48	10	0
R	50	333	684	987	1,232	1,407	1,520	1,591	1,629	**1,639**

Figure 30.36 - Maximum number of elements in lists R1, R2 and R. The row **Disc R1** indicates the number of discarded elements in R1.

SQL implementation of the *Chein* algorithm

The algorithm uses three physical tables, namely R1, R2 and R, R3 being another name for R1 at the next level. The maximum size of these tables are obtained by checking the button Size of the control panel of Figure 30.31. The figures shows that the sizes are those of the Python solver, considering that table R3 plays the role of list R2 (Figure 30.37).

	1	2	3	4	5	6	7	8	9	10
R1	50	856	1,305	955	652	407	232	119	48	10
R2	1,211	254,767	455,423	220,484	90,535	31,047	8,967	1,918	162	10
R3	856	1,305	955	652	407	232	119	48	10	0
Disc R1	0	573	954	652	407	232	119	48	10	0
R	50	333	684	987	1,232	1,407	1,520	1,591	1,629	**1,639**

Figure 30.37 - Number of rows of the working tables at each level (COUNTRY-50)

The interesting result is that of table R2, which contains the rectangles created by merging the couples of R1 rectangles that match. Created by a self-join of R1 in Step S3, the space complexity of R2 is $O(|R1|^2)$. Indeed, the first levels show an important increase of the size of R2.

The execution will consume 1,305 + 455,423 + 1,305 + 1,639 = 459,672 table rows. Though the evaluation and the comparison of these figures are approximate, there is little risk to conclude that the space requirement of the SQL implementation is about *one hundred times* that of the Python implementation.

The tables of the SQL implementations (as well as that of the next algorithm) are created in an *in-memory* database so that all the three algorithms execute in RAM, which make their comparison more pertinent.

SQL implementation of the *ISA recovery* algorithm

Expressed as the query of Section 30.17, the implementation does not use any intermediate data structure. The translation of OBJ_has_ATT to CONCEPT is performed in a single step. However, the execution of a `group-by` query usually includes sorting the source table, which itself relies on its own temporary data structures, such as extracts of the table to sort and indexes. Though precisely evaluating their size would need an in-depth knowledge of the internals of the RDBMS, we can try to estimate conservative figures for the COUNTRY-50 context.

The internal `group-by` subquery processes the table OBJ_has_ATT that contains 510 rows. Sorting them requires the storage of a bit more than 510 additional temporary rows. The subquery cannot produces more than 82 rows (32 distinct attributes + 50 objects if none of them include any proper attributes). The external `group-by` query will sort and aggregate these 82 rows.

These sizes can be considered negligible.

Python implementation of the *ISA recovery* algorithm

A similar analysis will also conclude that the memory space used by the Python code is negligible.

30.11.6 Runtimes

The execution time is a critical evaluation criterion. From it, we can derive guidelines for the applicability of each technique depending on the size of datasets. We have measured the runtime of each implementation when processing the COUTRY-10 to COUTRY-60 formal contexts. These implementations do not yet include the optimization techniques we will discuss later.

The table 30.38 shows particularly dramatic results:

– When running the *Chein* algorithm, the Python version turns out to be much faster than the SQL variant. The ratio between these times seems to rapidly increase with the size of G, the set of source objects. All other parameters being equal, it appears that their complexity of the algorithm is $O(|G|^3)$, which leaves little hope for improvement[32].

– The figures relative to the *ISA recovery* algorithm (both Python and SQL implementations) just mean that it computes the concepts in less than 1 millisecond. It must be kept in mind, however, that this algorithm generates a different set of concepts than the *Chein* algorithm.

– The building of the concept hierarchy is much faster when executed by the SQL query than by the Python procedure. Python could do better: optimization desperately needed!

	Chein algorithm		ISA recovery		Hierarchy	
	Python	SQL	Python	SQL	Python	SQL
COUNTRY-10	0.02	0.07	0.00	0.00	0.13	0.03
COUNTRY-20	0.40	1.95	0.00	0.00	34.70	0.81
COUNTRY-30	3.22	15.03	0.00	0.00	702.34	6.74
COUNTRY-40	14.65	83.22	0.00	0.00	5,771.82	23.57
COUNTRY-50	45.00	359.06	0.00	0.00		54.75
COUNTRY-60	128.53	1,322.97	0.00	0.00		115.96

Figure 30.38 - Execution times of the implementations of the FCA algorithms and of the derivation of the concept hierarchy

32. See [Sarmah et al., 2013]. Also observed experimentally by a regression on the series of time values. It should be noted that the Chein algorithm, though some other algorithms have better execution times, is far from the slowest [Kuznetsov, 2001].

It may be interesting to examine in more detail the SQL implementation of the *Chein* algorithm to identify the most expensive steps. The table in Figure 30.39 details the execution times for each step of each level when solving the COUTRY-50 context[33]. It clearly shows that the step S4, which merges the rectangles of R2, is the main culprit. This is in line with the sizes of the working tables.

	1	2	3	4	5	6	7	8	9	Σ	%
S3	0.04	10.51	22.83	11.50	5.40	1.99	0.62	0.16	0.03	53.08	**14.78**
S4	0.02	42.36	194.65	57.25	9.01	1.30	0.15	0.02	0.00	304.76	**84.88**
S5	0.00	0.28	0.53	0.23	0.09	0.03	0.02	0.00	0.00	1.18	**0.33**
S6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0.00**
S7	0.00	0.00	0.01	0.02	0.01	0.00	0.00	0.00	0.00	0.04	**0.01**
Σ	**0.06**	**53.15**	**218.02**	**69.00**	**14.51**	**3.32**	**0.79**	**0.18**	**0.03**	**359.06**	**100**

Figure 30.39 - SQL implementation of the Chein algorithm: execution time of each step of each level in seconds (solving COUNTRY-50),

30.11.7 Other techniques

Besides the Chein algorithm, the literature describes and analyzes more than a dozen FCA techniques that exhibit different performance characteristics. Although describing and comparing them would go far beyond the scope of this case study, it may be interesting to try to classify them according to different criteria. Here are three of them, which make sense after having studied the internals of the Chein algorithm.

– *Batch* or *incremental*. Batch techniques process the formal context as a whole while the incremental techniques consider each object, one at a time, integrating it to the solution built so far. The latter facilitates the evolution of the set of objects. The Chein algorithm belongs to the *batch* category.

– *Bottom-up* or *top-down*. The bottom-up algorithms start from the set of objects of the context as a first draft of the solution, then layer by layer, reduce the set of concepts to the maximal rectangles. The final solution is obtained when the last layer cannot generate new concepts. A top-down technique is an iterative procedure the starting point of which is a root concept comprising the set of all the objects of the context and all their common attributes (this latter set being often empty). Then new, more specific concepts are derived from the current state of the solution until no new concepts can be generated. The Chein algorithm belongs to the *bottom-up* category.

– *Concept set* or *concept hierarchy*. The final solution of some algorithms is a complete concept hierarchy, which comprises the set of concepts and their hierar-

33. The execution times of S1, S2 and Level 10 being less than 1 ms. are not shown.

chical organization. Other algorithms generates only the set of concepts, which is the case of the Chein algorithm. For them, the hierarchy is built by an independent procedure.

30.12 Improving the performance of the algorithms

The development of high performance implementations is not the main objective of this case study. Rather, it focuses, as do all the other studies in this series, on the modeling of the problem and of its solution(s) and on the simplicity of their implementation. Nevertheless, in some problems, the efficiency of the solution can be of crucial importance and is therefore an important part of the solving process. This is clearly the case for the FCA techniques we have designed in this chapter, some of which have impressive and, to put it bluntly, discouraging execution times.

30.12.1 *Chein* algorithm (Python implementation)

In the first implementation, the sets of rectangles R1 and R2 are represented by *lists of tuples* (g,m,d) where g is the object string-list, m the attribute string-list and d the discard indicator (Section 30.7). We ignore the representation of R (a list of couples (g,m)), common to all the techniques we will examine. We symbolize this implementation of R1 and R2 as follows:

- `R1[(g,m,d)]`
- `R2[(g,m,d)]`

The solving of contexts COUNTRY-10 to COUNTRY-60 shows much better execution times than the SQL implementation. Let us try to see whether we can gain even better results by experimenting with other implementations of sets R1 and R2. We will look at four alternative techniques.

– **2 dictionaries**. R1 and R2 are each expressed as a dictionary, symbolically denoted by `R1{g:(m,d)}` and `R2{g:(m,d)}`. A dictionary provides a quick access to an item based on the value of the key (`g`). In addition, it can be parsed as a list in the two embedded loops through the `list` function: `list(R1)`.

– **3 dictionaries + 2 lists**. In the preceding technique, the parsing of dictionary `R1{g:(m,d)}` is likely to be costly, due to its conversion into a list. We propose to complement it with a simple list `R1G[g]` of the `g` values to drive both loops. The extraction of the `m` value of `g` is then done through the dictionary, a very fast operation. The manipulation of `R2` is a bit more complex. First, we need to maintain a dictionary `R2{g:(m,d)}` and a list `R2G[g]` since, at the end of each level, `R2` becomes the new `R1`. Then, when searching `R2` for pairs of rectangles sharing the same `m` part, we need a quick access to `R2` through this `m` part, hence the additional dictionary `R2m{m:g}`. To sum up, the implementation comprises three dictionaries and two lists:

- `R1{g:(m,d)}, R1G[g]`
- `R2{g:(m,d)}, R2m{m:g}, R2G[g]`

– **5 lists**. The *2-list* implementation developed in Section 30.7 and recalled here before is simple and elegant, but does not support well the key-based access *g where m* and *m where g* since these queries must be performed by list parsing coded in Python. The idea is to let the Python engine execute them through the `index` method, likely to be faster than the equivalent Python loop. For instance, considering the list `L` and `v`, one on its element, the expression `L.index(v)` returns the position of the first instance of `v` in `L`. So, we decompose the list `R1` into three synchronized elementary lists and `R2` into two synchronized lists, totalling five lists as shown below:

- `R1G[g], R1M[m], R1D[d]`
- `R2G[g], R2M[m]`

– **2 lists of sets**. In the preceding techniques, we translated the sets of object names and of attribute names into *string-lists* (as discussed in Section 30.6.1). Now, we express these sets as Python sets of names. This could improve the union and intersection set operations. The sets of rectangles `R1` and `R2` are implemented as lists of tuples, each of them comprising a (Python) set of object names, a (Python) set of attribute names and the `d` indicator:

- `R1[(set{g},set{m},d)]`
- `R2[(set{g},set{m},d)]`

The experimental figures obtained when solving the contexts COUNTRY-10 to COUNTRY-100 are quite interesting (Figure 30.40).

The **2 dictionaries** technique (noted **2 dics**) is the most expensive and should be discarded.

The **5 lists** technique clearly appears to be the best, showing an improvement by a factor of 2.5 to 3.35 (measured for the COUNTRY-100 context, **2 dics** technique excluded).

The other three techniques, namely **2 lists**, **3 dics + 2 lists** and **sets** show intermediate figures, the initial **2 lists** technique being the more expensive, though not by far.

	2 lists	2 dics	3 dics + 2 lists	5 lists	sets
COUNTRY-10	0.02	0.02	0.01	0.01	0.00
COUNTRY-20	0.33	0.53	0.29	0.18	0.16
COUNTRY-30	2.54	6.48	2.59	1.50	1.34
COUNTRY-40	9.27	33.72	7.64	4.42	5.38
COUNTRY-50	24.12	104.96	20.56	11.10	12.35
COUNTRY-60	54.24	276.40	45.26	22.90	35.17

COUNTRY-70	121.28	751.88	97.70	46.19	80.17
COUNTRY-80	206.80	1,366.38	158.45	71.79	153.71
COUNTRY-90	481.51	3,094.88	297.64	125.13	294.77
COUNTRY-100	592.18		442.15	176.83	451.48

Figure 30.40 - Execution times (in seconds) of the five Python techniques

The graph of Figure 30.41 shows the relative position of the execution times of the five techniques.

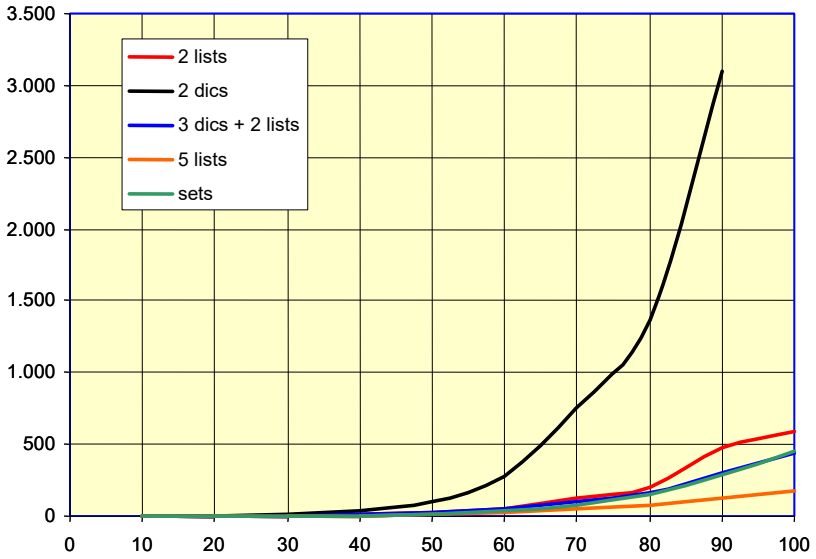


Figure 30.41 - Comparison of the execution times of the five Python techniques

30.12.2 *Chein* algorithm (SQL implementation)

The main optimization technique of relational databases is the creation of indexes. However, just as "*man proposes, God disposes*", the *database designer proposes, the RDBMS disposes*. In other words, the designer creates indexes that should logically improve the execution time of a query but in some circumstances, the SQL engine ignores the suggestion and chooses another way (hopefully better) to execute the query. This fact must be kept in mind in the following.

To try to reduce the execution time of the SQL implementation of the Chein algorithm, we analyze the structure of the queries developed in Section 30.6.2 to identify potential support indexes.[34]

34. Steps S1, S2 and S7, which each cost less than 1 ms., are ignored.

- *Step S3, Query 30.4*: the query is a self-join of table R1 based on the ordering criterion "rec1.G < rec2.G".

  Suggestion: index on R1(G)

- *Step S4, Query 30.5*: the query creates groups on column M of R2. Table R3 is filled sequentially, which does not require any index.

  Suggestion: index on R2(M)

- *Step S5, Query 30.6*: the query selects the rows of R1 that meet an equality condition on their (G,M) values. The couples of the right member of the condition are produced by an unconditional full scan of R2, which requires no index. Another way to spare access to R1 would be to store not only the G parts of the source rectangles but also their M parts, so that the equality checking can be performed *in situ* without additional access to R1. Better, since R3 is much smaller than R2, (Figure 30.37), the step S5 could be performed from the content of R3 instead of that from R2. However, the test described in Figure 30.39 shows that the contribution of step S5 to the total cost is very small, suggesting that any further optimization is worthless.

  Suggestion: index on R1(G,M)

- *Step S6, Query 30.7*: the query selects rows from R1 based on a binary column (Disc). The experiment reported in Figure 30.37 shows that the rows non discarded in R1 account for about 40% of the table. In this condition, an index on Disc would be useless and ignored by the SQL engine. Table R is sequentially filled, which does not require any index.

  Suggestion: none

To sum up, we retain these three hypotheses:

- on table R1,
  - an index on column G
  - a combined index on columns G and M
- on table R2,
  - an index on column M

To evaluate these suggestions, we apply them to the solving of contexts COUNTRY-20 to COUNTRY-60. We observe that index R2(M) brings no improvement and is discarded. On the contrary, indexes R1(G) and R1(G,M) have a positive effect that is represented in the graph of Figure 30.42. The x-axis specifies the number of source objects (20 for COUNTRY-20, 30 for COUNTRY-30, etc.) and the y-axis represents the ratio between the execution time *with index* and the time *without index*. For example, the black curve at x = 30 (COUNTRY-30) indicates that the execution time with index R1(G) is 80% (more precisely 79.6) of the time without index while the time with index R1(G,M) (the magenta curve) is slightly better (78.3%).

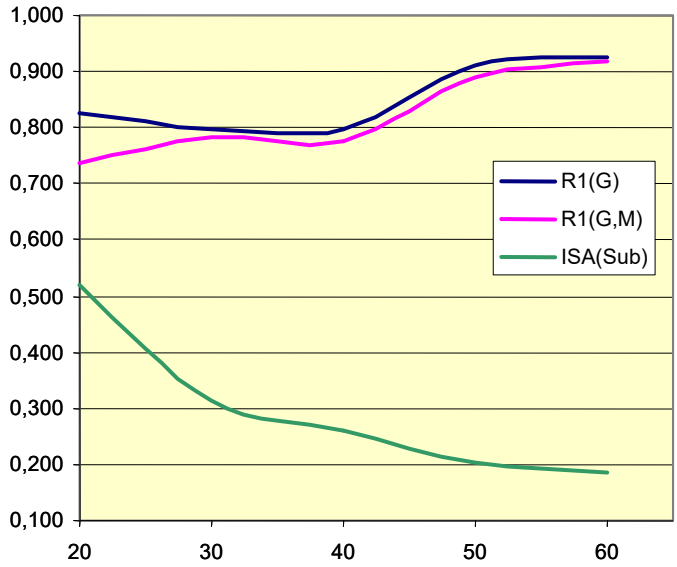


Figure 30.42 - Execution time improvement due to appropriate indexes

What does these curves tell us?

- whatever the size of the context, the improvement is not particularly impressive: from 74% to a meager 93%
- the efficiency of both indexes decreases with the size of the context
- the advantage of index R1(G,M) on R1(G) also decreases with the size of the context

To sum up, these indexes slightly improve the execution time of the SQL implementation of the Chein algorithm. However, their real usefulness is questionable, to say the least.

Note

The execution time of this SQL implementation and the influence of indexes are highly dependent on the structure (`|G|`, `|M|`, `|I|`) and content (`G`, `M`, `I`) of the source context. This can be shown by solving various configurations created by the *Context generator* (Context-generator.sql)

30.12.3 Implementation of the *ISA recovery* algorithm

The very low times measured for both the SQL and Python implementations do not suggest any specific optimization, at least for the type and size of contexts considered in this study. It would be interesting to check these times when processing larger contexts (several thousands of objects), for instance those generated by the context generator (Context-Generator.sql). This is left as an exercise.

30.12.4 Building of the concept hierarchy (SQL implementation)

We limit the discussion to the role of indexes on table ISA. The process executes two queries that create, then update, the table ISA.

The first one creates and stores in ISA the transitive closure of the inclusion relation on column G of table R (Script 30.12). It is a self-join, the join condition of which is based on a UDF (`itemInclude(sup.G,sub.G,',')`). As they are implemented in SQLfast, the UDF are *blind* to any index[35].

The second query computes the transitive reduction by deleting the transitive couples from ISA (Script 30.13). The conditions of the double self-join of the subquery includes explicit references to columns Sub and Super of table ISA, which suggests that an index on one (or on both) of these column may improve the execution time of the query. This hypothesis proves right, as the green curve of Figure 30.42 shows it:

- an index on ISA(Sub) contributes significantly to the reduction of the execution time; same for an index on ISA(Super) and ISA(Sub,Super)
- the index on ISA(Sub) proves better than those on ISA(Super) and on ISA(Sub,Super)
- the improvement is all the more important as the size of the ISA increases (times fall from 52% on COUNTRY-20 to 18% on COUNTRY-60)

Conclusion: create an index on ISA(Sub) before running the second query[36].

30.12.5 Building of the concept hierarchy (Python implementation)

The table in Figure 30.38 shows such a difference between the SQL and Python implementations that the latter seems irretrievably disqualified for a real usage.

First, it appears that the first part of the procedure (computing rawInclusion, the transitive closure of the hierarchy) is by far faster than the second (computing the transitive reduction, i.e., the final hierarchy). Thus, if an optimization is possible, it should be sought in this second part.

Let us recall its principle: we examine each couple of concepts (con1,con2) of the closure, for which we try to find an intermediate concept con3, different from con1 and con2, such that both (con1,con3) and (con3,con2) belong to the closure. If we find such a concept, then the initial couple is transitive and must be rejected, otherwise, it belongs to the reduced hierarchy.

The high cost of the algorithm seems to come from the way con3 is selected: we examine *all the concepts* (but con1 and con2, which can be neglected). Let us reduce the source of con3 to the set of superconcepts of con1. Some experiments in solving

35. In the SQL slang, such a query is called *non-sargable*, that is, non-optimizable [https://stackoverflow.com/questions/799584/what-makes-a-sql-statement-sargable]
36. Why not before the first query?

the COUNTRY family contexts show that, on the average, each concept has from 2.25 to 4 superconcepts, which is a substantial reduction in size! The code of Script 30.23 translates this idea.

```
hierarchy = []
for (con1,con2) in rawInclusion:
    status = 'basic'
    for con3 in [con for (con0,con) in rawInclusion
                 if con0 == con1 and con <> con2]:
        if (con3,con2) in rawInclusion:
            status = 'transitive'
            break
    if status == 'basic':
        hierarchy.append((con1,con2))
```

Script 30.23 - Improved procedure of transitive reduction of concept hierarchy: *reducing the search space*

The decrease of the execution times is also dramatic, as shown in the table of Figure 30.43. The figures of column Python 1 are those of the initial algorithm (copied from Figure 30.38). Those of column Python 2 are obtained by the new algorithm. The last two columns show that the SQL script still remains the best technique, specially when supported by index ISA(Sub) (column SQL 2).

	Python 1	Python 2	SQL 1	SQL 2
COUNTRY-10	0.13	0.02	0.03	0.04
COUNTRY-20	34.70	2.07	0.81	0.44
COUNTRY-30	702.34	19.73	6.74	2.16
COUNTRY-40	5,771.82	80.62	23.57	5.98
COUNTRY-50		208.99	54.75	11.60
COUNTRY-60		452.93	115.96	20.80

Figure 30.43 - Dramatic effect of simple optimizations of the concept hierarchy building (times in seconds)

30.13 On the representation of concept hierarchies

Preliminary remark

The term *Galois lattice* often appears as a synonym for *concept hierarchy*. In fact, Galois lattices are a mathematical domain at the origin of FCA. The lattice structures a set of elements among which a partial order holds. A concept hierarchy is a Galois lattice if it satisfies two properties: any pair of concepts have a unique closest (lowest) common superconcept and a unique closest (highest) common

subconcept. In particular, this implies that the hierarchy has a unique root concept and a unique leaf concept. If the hierarchy has several roots, an artificial common superconcept is added, the G part of which comprises all the objects names and the M part is empty. If the hierarchy comprises several leaf concepts, an artificial common subconcept is added, the G part of which is empty and the M part comprises all the attribute names. The various developments of this study show that not all concept hierarchies are genuine Galois lattices.

Throughout this study, we constructed various forms of equivalent hierarchy representations, in which the G part has been replaced by meaningful names, in which only proper attributes were shown, or in which all source objects have been converted into concepts. In this section, we collect all these forms (and some others) and we show how each of them derives from the others. We illustrate the discussion on the example of the ANIMAL context[37], recalled below (see Section 30.10.2):

Object	**Attributes**
CANARY	eggs,feather,fly,breath
CROCODILE	eggs,teeth,swim,breath
DUCK	eggs,feather,fly,swim,breath
FROG	eggs,swim,breath
OSTRICH	eggs,feather,breath
SALMON	eggs,swim
SHARK	eggs,teeth,swim

... and which can be represented graphically as in Figure 30.44.

CANARY	CROCODILE	DUCK	FROG	OSTRICH	SALMON	SHARK
eggs	eggs	eggs	eggs	eggs	eggs	eggs
feather	breath	feather	breath	feather	swim	teeth
fly	teeth	fly	swim	breath		swim
breath	swim	breath				
		swim				

Figure 30.44 - Graphical representation of a formal context

By application of the standard Chein algorithm, we obtain the concept hierarchy of Figure 30.45, graphically rendered in Figure 30.46.

```
(CROCODILE)x(breath,eggs,swim,teeth)
(DUCK)x(breath,eggs,feather,fly,swim)
(CANARY,DUCK)x(breath,eggs,feather,fly)
(CROCODILE,SHARK)x(eggs,swim,teeth)
```

37. Just note that these graphical representations are useful to reason on small contexts and concept hierarchies!

```
(CANARY,DUCK,OSTRICH)x(breath,eggs,feather)
(CROCODILE,DUCK,FROG)x(breath,eggs,swim)
(CANARY,CROCODILE,DUCK,FROG,OSTRICH)x(breath,eggs)
(CROCODILE,DUCK,FROG,SALMON,SHARK)x(eggs,swim)
(CANARY,CROCODILE,DUCK,FROG,OSTRICH,SALMON,SHARK)x(eggs)
```

Figure 30.45 - "GxM" representation of an FCA solution

If we feel uncomfortable with a schema in which some source object are missing, we can add them as shown in Figure 30.47 (in blue).

Since the subconcept/superconcept relation was derived from the composition of their G part (or, equivalently of their M part), the hierarchy is a redundant structure.

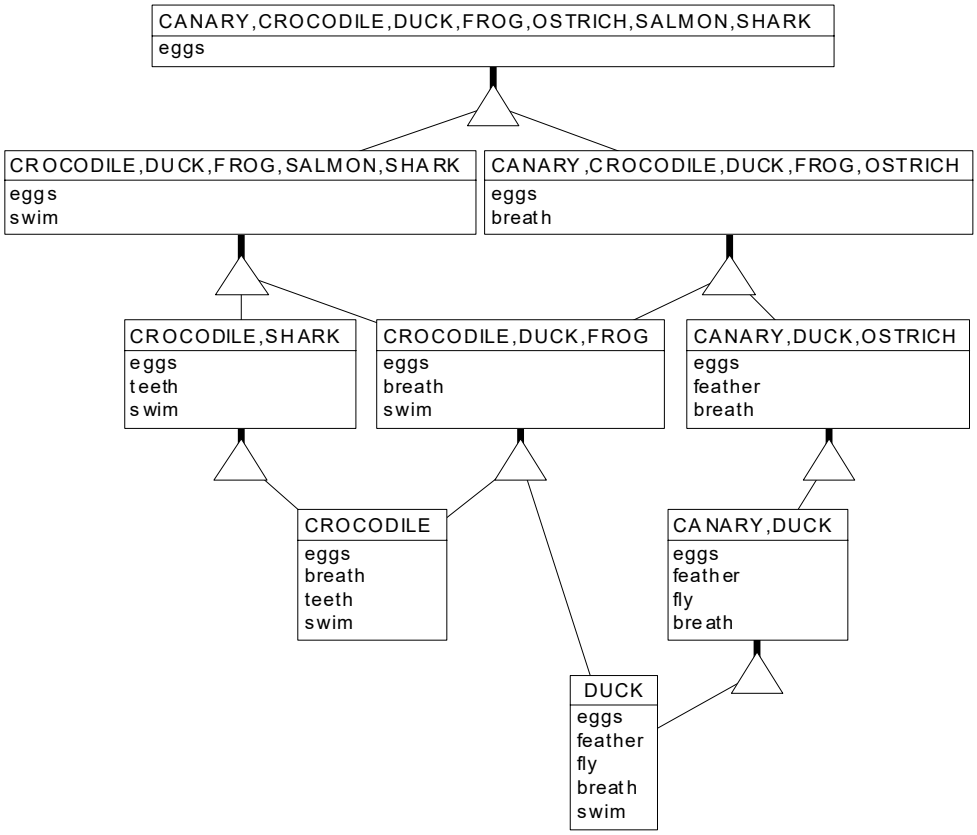


Figure 30.46 - Graphical representation of a concept hierarchy (empty source object omitted, all the attributes shown)

The reduction of the M part of the concepts to their proper attributes is shown in Figure 30.48. Again, the hierarchical structure is derivable and therefore redundant.

In some applications, such as in knowledge engineering, the nature of each concept in the real-world is important, which requires that it be given a meaningful

name. This is what we have tried to do in the schema of Figure 30.49. The hierarchy then takes the form of a taxonomy (or ontology) of a part of the real world. Since the syntactic inclusion relation between the G part of the concepts is gone, the subconcept/superconcept relation must be explicitly stated.

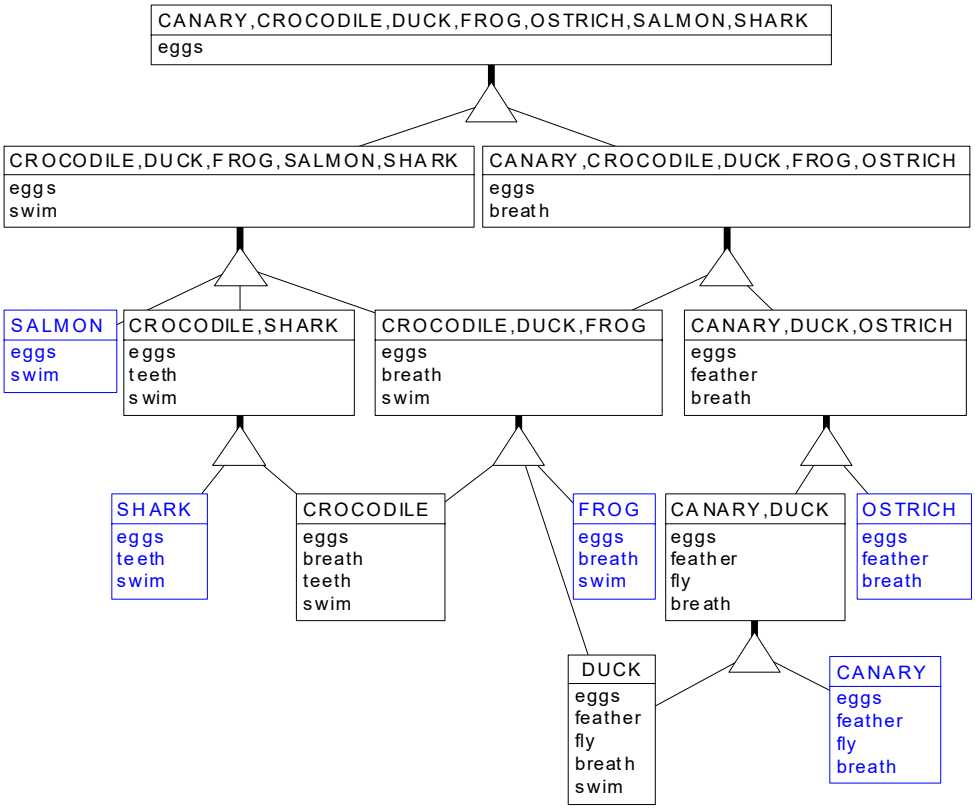


Figure 30.47 - Graphical representation of a concept hierarchy (empty source objects shown as concept)

The recovery of a complete hierarchy such as that of Figure 30.47 from the reduced version of Figure 30.48 is performed by the inheritance mechanism. Since the latter proceeds from the top of the hierarchy downward, we can call it *descending inheritance*.

Now, let us look closely at the hierarchy of Figure 30.48. We observe that the name of a subconcept (its G part) is included in the name of its superconcepts. If we remove the subconcept name from the name of its superconcept, we do not loose any information. In a sense, the superconcept *inherits* the names of its subconcepts, a mechanism that can be called *ascending inheritance*.

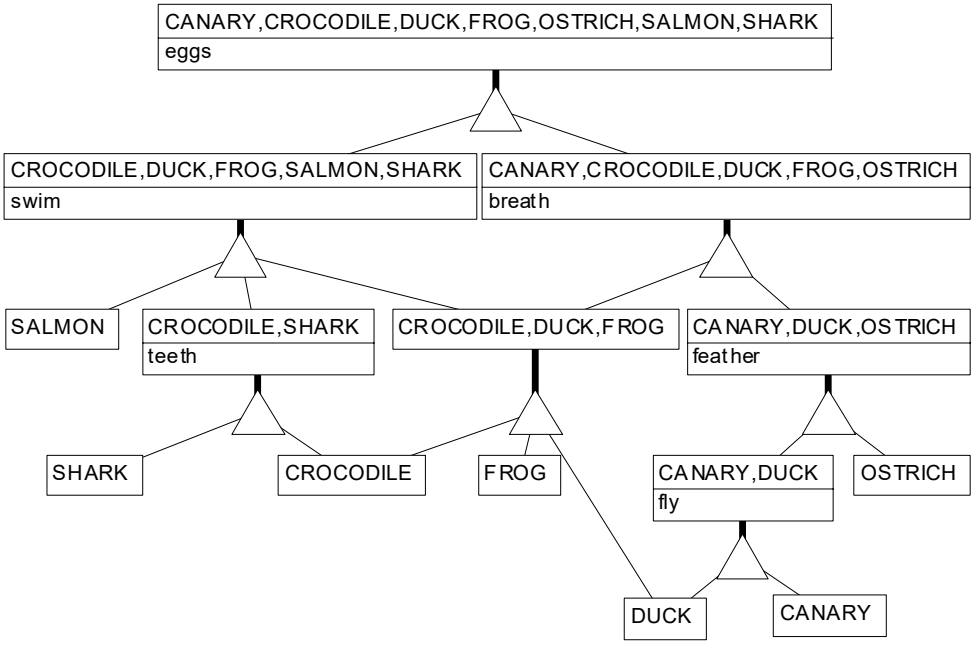


Figure 30.48 - Graphical representation of a concept hierarchy (empty source object and only proper attributes shown)

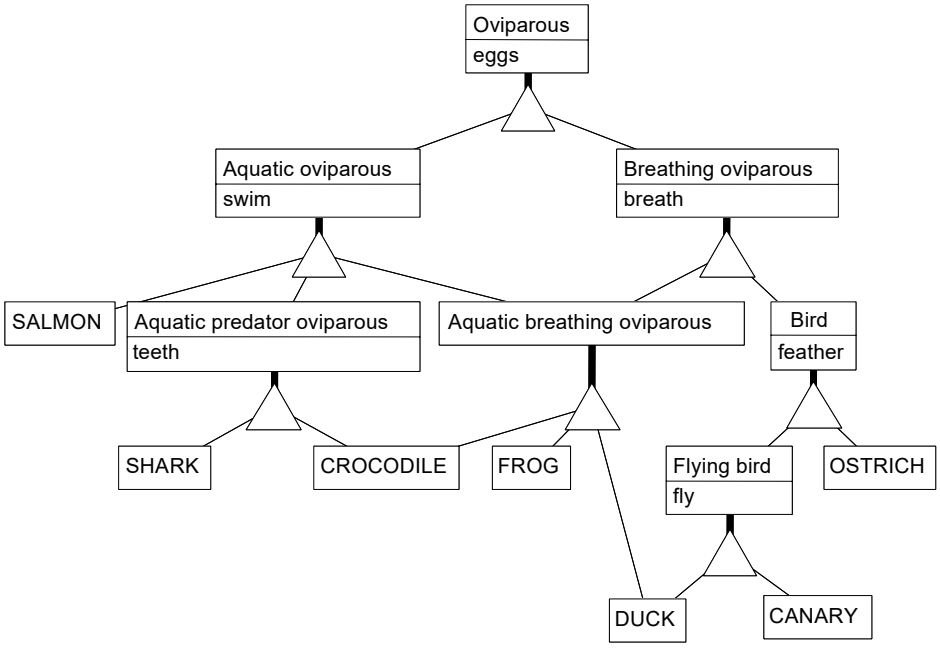


Figure 30.49 - Graphical representation of a concept hierarchy (empty source object and only proper attributes shown, G part renamed)

By applying both the descending inheritance (of the M part) and the ascending inheritance (of the G part), we get a particularly concise form of the concept hierarchy, shown in Figure 30.50. Please note that in this form, the concepts no longer are identified by their G part alone.

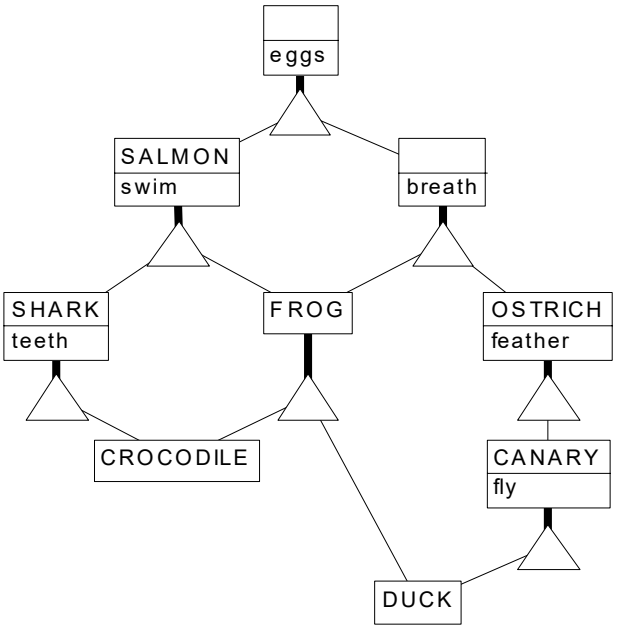


Figure 30.50 - Minimal structure of a concept hierarchy

30.14 Querying the concept hierarchy

The hierarchical structure is an appropriate support to query the concepts. Let us base our reasoning on the schema of Figure 30.35 in which *only the proper attributes* of the COUNTRY-10 hierarchy are shown. We remember that the objects are *countries* and the attributes are socio-economic properties of these countries. Considering the large number of new concepts generated (see the table of Figure 30.34), giving them meaningful names is unrealistic. Considering this hierarchy as a *database*, we can ask such questions as the following (X is a variable containing the G part of a concept):

- how many objects does concept X comprise:

```
select itemLen(G,',') from CONCEPTS where G = :X;
```

- how many subconcepts does concept X have:

```
select count(*) from ISA where Super = :X;
```

- The G part of concept C is a superset of the G part of each of its subconcepts. The larger the subconcept, the more influence it has on the composition of C. What are the most influential subconcepts of each superconcept:

```
select Super,Sub,Size
from  (select Sub,Super,
              itemLen(Sub,',') as Size,
              max(itemLen(Sub,','))
                  over (partition by Super) as MaxSize
       from   ISA
       )
where Size = MaxSize;
```

Inference rules

The analysis of the hierarchy allows us to discover hidden rules that hold in the source data. We briefly describe two of them, drawn from reference [D'Aquin, 2020].

In subconcept C, let A be a proper attribute and B an attribute inherited from one of its direct superconcept D. The descending inheritance mechanism implies that in any direct or indirect subconcept of C, the M part comprises both attributes A and B. This leads to an interesting consequence:

any object with attribute A also have attribute B,

this property can be restated as *A implies B* and formalized by the dependency rule

$A \longrightarrow B$.

To appreciate the importance of this rule, let the objects be the *countries* represented in a COUNTRY context, A be *inflation_rate:high* and B be *life_expectancy:low*. If the rule $A \longrightarrow B$ holds in the hierarchy, this means that all the countries with a *high inflation rate* also have a *low life expectancy*:

inflation_rate:high $\longrightarrow$ life_expectancy:low

The second rule looks like, in a sense, the inverse of the first one. Again, we consider the (C<D) example discussed above. Let us call M_C the M part of C and M_D the M part of D. Let us also call S_C and S_D the sizes of C and D respectively and Mc-d the attributes of C that are not in D (proper or inherited from other superconcepts). If S_C is very close to S_D, say, with a difference of 10 percent, then we can assert that M_D implies Mc-d with a probability of 0.9 ($M_D \longrightarrow_{[0.9]}$ Mc-d). Ok, this rule seems a bit obscure, so let us apply it to the example above. We assume that concept D comprises 50 countries and concept C, 45 countries. Among the 50 countries that have a *low life expectancy*, 45 (or 90%) also have a *high inflation rate*. Hence the rule, that we will call *probabilistic dependency*:

life_expectancy:low $\longrightarrow_{[0.9]}$ inflation_rate:high

The query of Script 30.24 computes the probabilistic dependencies that hold in the ISA table. Applied to the context COUNTRY-20, with a threshold of 0.9, it generates

the dependencies shown in Figure 30.51. We observe that this result can include transitive dependencies (here, 16 $\rightarrow_{[0.9]}$ 9). Their elimination is left as an exercise.

```
select SuperM as Left,
       '-->['||round(1.*SubSize/SuperSize,2)||']' as 'implies',
       itemExcept(SubM,SuperM,',') as Right
from (select Sub,itemLen(Sub,',')     as SubSize,
             Super,itemLen(Super,',') as SuperSize,
             C1.M as SubM,C2.M as SuperM
      from   ISA,R C1,R C2
      where  1.*SubSize/SuperSize >= 0.9
      and    ISA.Sub = C1.G and ISA.Super = C2.G
      );
```

Script 30.24 - Computing the *probabilistic* dependencies (probability 0.9)

Left	implies	Right
16	-->[0.9]	20
16	-->[0.9]	9
20	-->[0.91]	9

Figure 30.51 - The *probabilistic* dependencies in COUNTRY-20

30.15 Applications

The most general application domain of FCA is *knowledge discovery and processing* [Kuznetsov, 2004,2015], a definition that encompasses a very large variety of more specific applications. The examples used in this study suggest some typical objectives of FCA techniques, in particular:

– elicitation of hidden significant patterns, such as concepts, hierarchy of concepts and dependencies, in source data collections

– normalization of database schemas by reconstruction of subtype/supertype structures, both in the elaboration of the conceptual schema of a database under construction and in the reverse engineering of legacy databases [Beeri, 1999] [Hainaut, 2002].

The basic mechanism of FCA being to *classify* uninterpreted objects, it can be applied to any kind of these objects, such as documents and messages [Cigarrán 2016].

In software engineering, a pertinent architecture can be suggested among a collection of programs based on common resources: database tables read or updated, APIs, dialog boxes, user classes, access control (user-privilege relation).

In the database realm, besides the forward and reverse engineering mentioned above, a plausible schema can be extracted from a schema-less table (collection of rows with no common structure). Such application is common in NoSQL databases in which each record defines its own structure[38].

30.16 A short bibliography

[Beeri, 1999] C. Beeri, A. Formica, M. Missiko. *Inheritance hierarchy design in object-oriented databases*, Data & Knowledge Engineering 3 0(1999) pp.191–216

[Castellanos 2017] A. Castellanos, J. Cigarrán, A. García-Serrano. Formal concept analysis for topic detection: A clustering quality experimental analysis, Information Systems, 66(2017) pp. 24–42

[Chein, 1969] M. Chein. Algorithme de recherche des sous-matrices premières d'une matrice, Bull. Math. R.S. Roumanie, 13, 1969

[Cigarrán 2016] J. Cigarrán, A. Castellanos, A. García-Serrano. A step forward for Topic Detection in Twitter: An FCA-based approach, Expert Systems With Applications 57(2016) pp. 21–36

[D'Aquin, 2020] M. D'Aquin. A Demystifying Introduction to Formal Concept Analysis (FCA) [https://towardsdatascience.com/a-demystifying-introduction-to-formal-context-analysis-fca-ab8ce029782e]

[Hainaut, 2002] J.-L. Hainaut. Introduction to Database Reverse Engineering [https://projects.info.unamur.be/~dbm/mediawiki/index.php?title=LIBD:OUVRAGES-WEB]

[Huchard, 2000] M. Huchard, H. Dicky and H. Leblanc. Galois Lattice as a Framework to Specify Building Class Hierarchies Algorithms, RAIRO-Theor. Inf. Appl., 34 6 (2000) 521-548 [https://www.rairo-ita.org/articles/ita/pdf/2000/06/ita0110.pdf]

[Kuznetsov, 2001] S. O. Kuznetsov, S. A. Obedkov. Comparing Performance of Algorithms for Generating Concept Lattices, ICCS'01 Int'l. Workshop on Concept Lattices-based KDD pp. 35–47 [https://ceur-ws.org/Vol-42/paper3_kuznetsov.pdf]

[Kuznetsov, 2004] S. O. Kuznetsov. Machine Learning and Formal Concept Analysis, CFCA 2004, LNAI 2961, pp. 287–312, 2004. Springer-Verlag

[Kuznetsov 2015] S. O. Kuznetsov1 and A. Napoli. Formal Concept Analysis: Themes and Variations for Knowledge Processing, *Tutorial on Formal Concept Analysis at IJCAI 2015*, Buenos Aires, July 27th 2015 [https://ijcai-15.org/downloads/tutorials/T23-FCA.pdf]

38. Please note that the SQLfast environment includes a subsystem allowing users to create and exploit schema-less tables. See *SQLfast-Manual, Chapter 25 - Dynamic columns and schema-less tables*.

[Sarmah, 2013] A.K. Sarmah, S.M. Hazarika, S.K. Sinha. Formal concept analysis: current trends and directions, June 2013, Artificial Intelligence Review 44(1):1-40, June 2013 [https://www.researchgate.net/publication/255999110_Formal_concept_analysis_current_trends_and_directions]

[Wikipedia, 2023] Formal Concept Analysis [https://en.wikipedia.org/wiki/Formal_concept_analysis]

Case study **31**

Path finders, rovers and Ariadne's thread

Objective: This chapter tackles a widespread optimization problem: computing the shortest path between two cities. The solving technique is based on Dijkstra's algorithm. It also is applied to two similar applications domains, namely maze solving and controlling a rover on a hostile planet. A general purpose, application independent, solving tool is developed.

Keywords: optimization, shortest path, Dijkstra's algorithm, maze solving, rover control

31.1 Shortest paths between two cities

Let us consider a set of cities between which roads have been built (Figure 31.1). We associate a *distance* with each road. For instance, the road from city **A** to city **B** is **85** km long. We will write: $\mathbf{d_{AB}} = \mathbf{85}$. To compute the distance between cities **A** and **F** we add distances **85** and **80**, that is, $\mathbf{d_{AF}} = \mathbf{165}$ km.

Computing $\mathbf{d_{AJ}}$, the distance between **A** and **J**, is a bit more complex. Indeed, the figure shows that three different paths with three different lengths connect city **A** to city **J**, namely:

- **A.B.F.I.J** with $\mathbf{d_{AJ}} = \mathbf{499}$ km
- **A.C.H.J** with $\mathbf{d_{AJ}} = \mathbf{487}$ km

– **A.E.J** with $\mathbf{d_{AJ}} = \mathbf{675}$ km

Hence the concept of *shortest path* between two cities. Computing efficiently this shortest path is the main objective of this chapter.

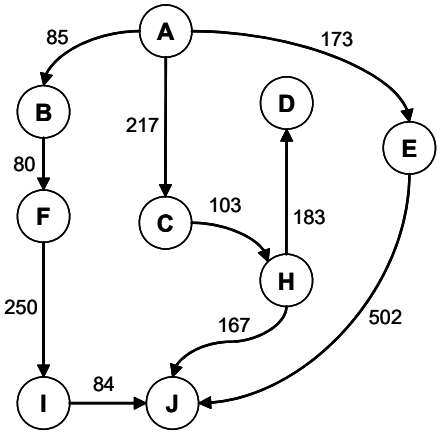


Figure 31.1 - A map showing cities and roads between them

31.2 Dijkstra's Algorithm

Dijkstra's algorithm is the most popular technique to find the *shortest path* between two nodes of a graph[1].

Each edge of the graph is given a *cost* that measures the *effort* required for travelling from the source node to the target node. This cost (a non-negative number) can be the distance or more generally some resource (such as money, time, energy or gasoline), depending on the nature of the problem.

We will develop one of the variants of this algorithm, that computes the shortest paths between a definite node, the *starting node*, and each of the other nodes of the graph (actually, one of the shortest paths, since more than one path with the same length may exist), a structure called the *shortest-path tree*. By querying this tree, we can extract the shortest path between the starting node and any target node.

Let **A** be the starting node of the graph. We assign to each node a property that specifies its distance from this starting node. Before applying the algorithm, we set this property to **0** for node **A** (the distance between **A** and **A** is zero) and **infinite** for all the other nodes (Figure 31.2). The goal of the algorithm is to adjust this distance until we get its minimal value.

1. See https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm for example

We distinguish in this graph two subsets of nodes, the nodes for which the minimal distance has already been computed and the other nodes, for which the minimal distance has yet to be evaluated. Let us call the first category *internal nodes* (they belong to the solution) and the others *external nodes*. Before applying the algorithm, the set of internal nodes is empty. Internal nodes are drawn in dark gray and the external nodes in white.

We will examine each of the external nodes, one by one. When we select a node to examine, we color it in light gray.

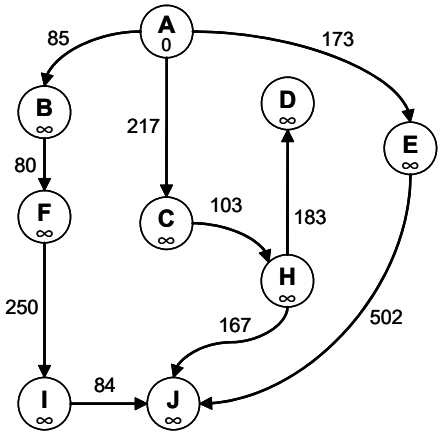


Figure 31.2 - Initial state of the graph

Quite naturally, we examine starting node **A** first, which is then colored in light gray (Figure 31.3/left). The distance of **B** from **A**, $\mathbf{d_B}$, is set to $\mathbf{d_{AB}}$. More precisely, $\mathbf{d_B} = \mathbf{d_A} + \mathbf{d_{AB}}$. The same modification is applied to the other neighbor nodes **C** and **E**. The result is shown in Figure 31.3/left. Node **A** is (trivially) an internal node and is colored in dark gray (Figure 31.3/right).

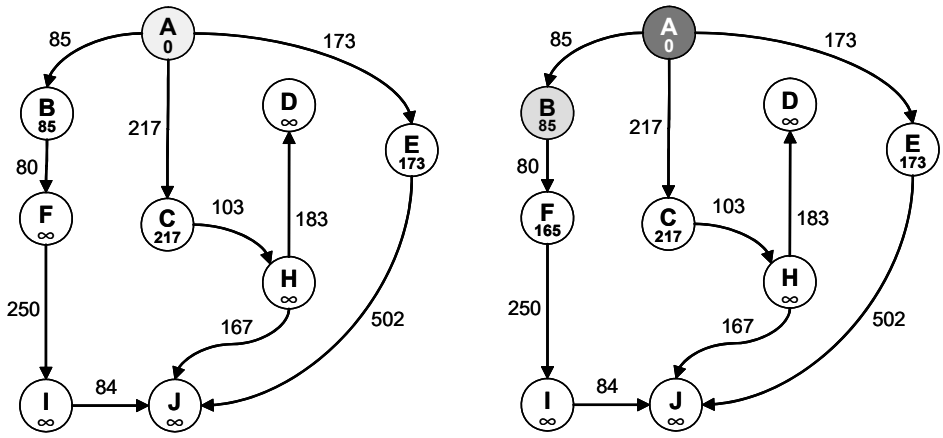


Figure 31.3 - Processing cities **A** and **B**

Here comes the smart part of the algorithm.

Among the external subset, we select the node with the lowest distance and we examine it. At this stage, this node is **B**, with distance $d_B = 85$. Its unique neighbor is **F**, for which we compute new distance $d_F = d_B + d_{BF} = 85 + 80 = 165$. Since this new value is lower than the former one (∞), it is assigned to **F**. The current state of the graph is shown in Figure 31.3/right.

What is the status of node **B**? Could we find, later, a better path from **A** to it? No, we couldn't since this hypothetical path should include at least one external node and all of them have a higher distance than that of **B**. So, d_B really is minimal so that **B** can be declared an internal node, which therefore is colored in dark gray (Figure 31.4/left).

Once again, we select the external node with the lowest distance and we examine it. This node is **F**, with distance **165**. It is colored in light gray and its neighbors are updated. Its unique neighbor is **I**, the new distance of which is $d_I = d_F + d_{FI} = 165 + 250 = 415$. Since $415 < \infty$, the distance of **I** is updated (Figure 31.4/left).

The reasoning about the status of **B** can be applied to **F**, which becomes internal and is colored in dark gray (Figure 31.4/right).

In the next iteration, we select and examine external node **E**, which now has the lowest distance (**173**). It is colored in light gray and its unique neighbor, **J**, is updated: $d_J = d_E + d_{EJ} = 173 + 502 = 675$. Since $675 < \infty$, the distance of **J** is updated (Figure 31.4/right).

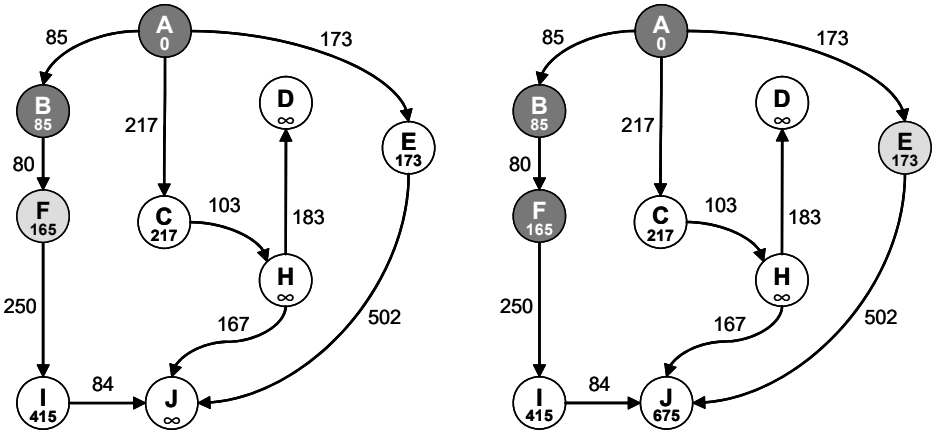


Figure 31.4 - Processing cities **F** and **E**

In the following iterations:

– **E** becomes internal, **C** is selected ($d_C = 217$ is the lowest) and examined, its neighbor **H** is updated with $d_H = d_C + d_{CH} = 320$. Since $320 < \infty$, the distance of **H** is updated (Figure 31.5/left).

– **C** becomes internal, **H** is selected (d_H = **320** is the lowest) and examined, its neighbors **D** and **J** are updated. For **J**, $d_J = d_H + d_{HJ}$ = **487**. Since **487** < **675**, the distance of **J** is replaced (Figure 31.5/right).

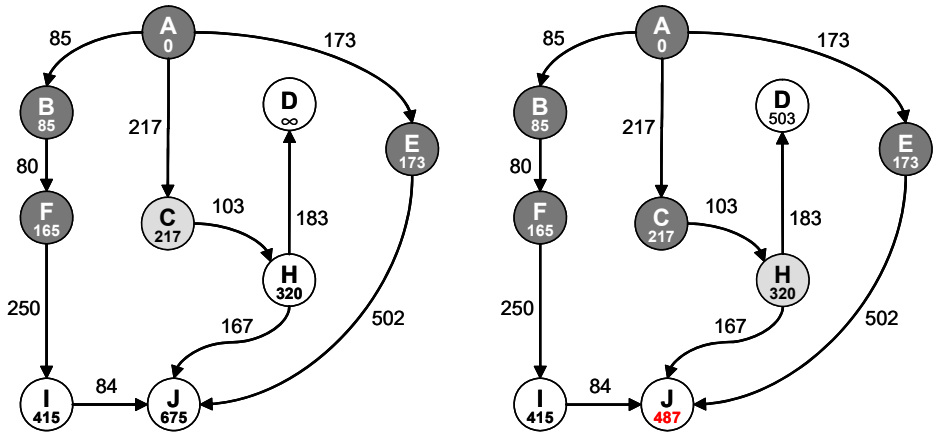


Figure 31.5 - Processing cities **C** and **H**. Improving the path to **J**.

– **H** becomes internal, **I** is selected (d_I = **415** is the lowest) and examined, its neighbor **J** is tentatively updated. $d_J = d_I + d_{IJ}$ = **499**. Since **499** > **487**, the distance of **J** is **not** replaced (Figure 31.6/left).

– **I** becomes internal, **J** is selected (d_I = **487** is the lowest) and examined, it has no neighbor so that its examination stops (Figure 31.6/right).

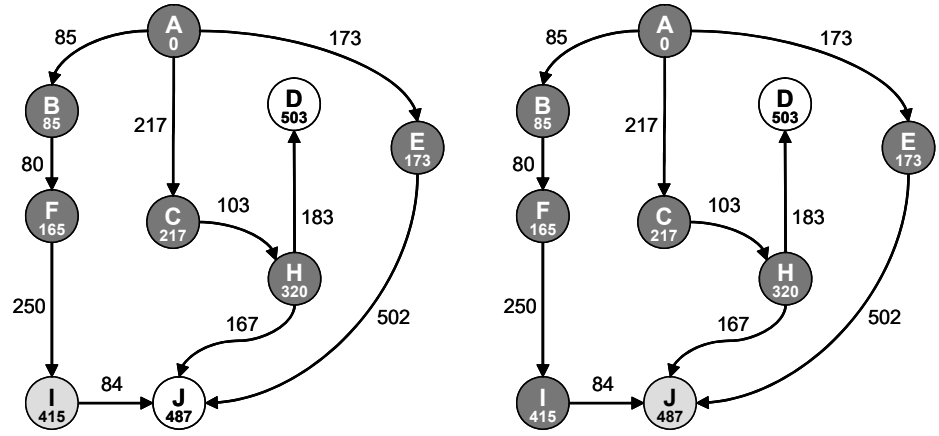


Figure 31.6 - Processing cities **I** and **J**

– **J** becomes internal, the last external node **D** is selected and becomes internal.

The final state is shown in Figure 31.7. The shortest path from **A** to **J** is **A.C.H.J** , with a distance of **487** km.

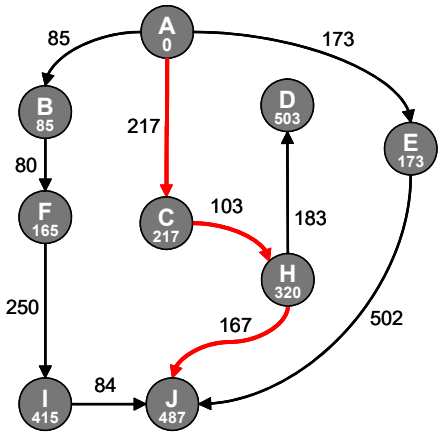


Figure 31.7 - The shortest path from **A** to **J** is **A.C.H.J**

Important notes

It is worth to note that this algorithm still is valid if the graph is not acyclic, that is, includes circuits. It is also valid if it is not directed, i.e., if paths can be followed in both directions.

The algorithm produces the shortest path from the starting city to each of the other cities. Actually, it provides much more! If d_{AK} is the length of the shortest path from **A** to **K**, and if **E** is in this path, then $d_{EK} = d_{AK} - d_{AE}$.

31.3 Representation of the graph

Table CITY stores the description of the nodes (the cities) of the graph: CityID (technical id), Name (city name), Mark (**0** for external nodes and **1** for internal nodes), Path (the current shortest path to this city from the starting city) and Dist (length of the current shortest path).

Tables NPATH and PATH both represents the edges (the paths) of the graph. NPATH expresses them through *city names* while PATH uses *city ID*'s. The first one is more convenient for users while the second one is more efficient for complex calculations (Script 31.1)

To make SQL queries more expressive, we also define views, namely EXTERNAL, that selects external cities, and ExtPATH that provides an extended presentation of PATH (Script 31.2).

Script 31.3 loads the data describing the graph of Figure 31.2.

If the graph is not directed, we create the inverse paths:

```
insert into NPATH(City1,City2,Len)
       select City2,City1,Len from NPATH;
```

The starting state of the graph is defined as follows (999999 denotes *infinity*):

```
update CITY set Mark = 0, Path = '', Dist = 999999;
```

```
create table CITY(CityID integer not null
                       primary key autoincrement,
                  Name   varchar(32) not null,
                  Mark   integer,
                  Path   varchar(256),
                  Dist   integer);
create table NPATH(City1 varchar(32) not null,
                   City2 varchar(32) not null,
                   Len   integer not null);
create table PATH(City1  integer not null,
                  City2  integer not null,
                  Len    integer not null);
```

Script 31.1 - Data structure for City/Path graphs

```
create view EXTERNAL as select * from CITY where Mark = 0;
create view ExtPATH(City1,Name1,Dist1,Len,City2,Name2,Dist2)
       as select P.City1,C1.Name,C1.Dist,P.Len,
                 P.City2,C2.Name,C2.Dist
          from   CITY as C1, PATH as P, CITY as C2
          where  C1.CityID = P.City1
          and    P.City2 = C2.CityID;
```

Script 31.2 - Complementary views

```
insert into CITY(Name) values ('A'),('B'),('C'),('D'),('E'),
                              ('F'),('H'),('I'),('J');
insert into NPATH(City1,City2,Len) values
       ('A','B',85),('A','C',217),('A','E',173),('B','F',80),
       ('C','H',103),('F','I',250),('H','D',183),
       ('H','J',167),('E','J',502),('I','J',84);
```

Script 31.3 - Load the City/Path graph of Figure 31.2

Script 31.4 derives PATH rows from NPATH rows, asks the user the name of the starting city and updates the state of this city.

```
insert into PATH select C1.CityID,C2.CityID,Len
                  from   CITY C1, NPATH P, CITY C2
                  where  C1.Name = P.City1
                  and    P.City2 = C2.Name;
ask start = Starting city:[select Name,CityID
                           from   CITY order by Name];
update CITY set Mark = 0, Dist = 0, Path = Name
where  CityID = '$start$';
```

Script 31.4 - Prepare data and select the starting city

31.4 Computing the shortest paths from a starting city

The structure of the algorithm is a loop each iteration of which selects and examines an external city. Since each external city will be selected once and only once, the loop will look as follows:

```
extract N = select count(1) from CITY;
for step = [1,$N$];
  <A. select S = external city with the lowest distance>
  <B. update the neighbors of S>
  <C. make S an internal city>
endfor;
```

A. Selection of S, an external city

We extract the ID of a city from the EXTERNAL view where the distance (column Dist) is equal to the minimum of the distances (`min(Dist)`) of all the external cities. If more than one city may satisfy this condition, the first one is selected. Hence the query:

```
extract S = select CityID from EXTERNAL
            where  Dist = (select min(Dist) from EXTERNAL);
```

B. Updating the neighbors of S

This task is the core of the algorithm. It will be translated into a single SQL `update` query. To make it more expressive, we first define two shorthands that translate natural concepts:

– **the *neighbors* of S**:

```
set neighbors = select City2 from PATH where City1 = $S$;
```

– **the tentative *new distance* of each neighbor of S**:

```
set newDist = (select Dist1 + Len from ExtPATH
               where  City1 = $S$
               and    City2 = CITY.CityID);
```

This part of the update query computes the distance of the neighbor city being updated, denoted by `CITY.CityID`. Dist1 is the distance of the selected external city and Len is the length of the path between these two cities.

The updating can then be written very naturally, as follows:

```
update CITY
set    Dist = min(Dist,$newDist$)
where  CityID in ($neighbors$);
```

Recording the paths

So far, we have computed the *shortest distance* between two cities but we have not recorded the intermediate cities that constitute this *shortest path*. When we update a neighbor, if the new distance is lesser than its current distance, we also replace the current path of the neighbor by that of the selected external city to which we append the name of the neighbor:

```
update CITY
       Path = case when  Dist > $newDist$
                  then  (select Path from CITY
                         where CityID = $S$)||'.'||Name
                  else  Path
              end
where  CityID in ($neighbors$);
```

By merging both `update` queries we get:

```
update CITY
set    Dist = min(Dist,$newDist$),
       Path = case when  Dist > $newDist$
                  then  (select Path from CITY
                         where CityID = $S$)||'.'||Name
                  else  Path
              end
where  CityID in ($neighbors$);
```

C. Making S an internal city

The SQL expression is straightforward:

```
update CITY set Mark = 1 where CityID = $S$;
```

The complete algorithm is shown in Script 31.5.

```
extract N = select count(1) from CITY;
for step = [1,$N$];
   extract S = select CityID from EXTERNAL
               where  Dist = (select min(Dist)
                              from   EXTERNAL) #1;
   set neighbors = select City2 from PATH
                   where  City1 = $S$;
   set newDist = (select Dist1 + Len from ExtPATH
                  where  City1 = $S$
                  and    City2 = CITY.CityID);
   update CITY
   set    Dist = min(Dist,$newDist$),
          Path = case when  Dist > $newDist$
                      then  (select Path from CITY
                             where CityID = $S$)||'.'||Name
                      else Path
                 end
   where  CityID in ($neighbors$);
   update CITY  set Mark = 1 where CityID = $S$;
endfor;
```

Script 31.5 - Computing the shortest paths of cities from a starting city

The execution of this script for the problem depicted in Figure 31.1 provides this solution:

CityID	Name	Mark	Path	Dist
1	A	1	A	0
2	B	1	A.B	85
3	C	1	A.C	217
4	D	1	A.C.H.D	503
5	E	1	A.E	173
6	F	1	A.B.F	165
7	G	1	A.C.G	403
8	H	1	A.C.H	320
9	I	1	A.B.F.I	415
10	J	1	A.C.H.J	487

If we are only interested by the shortest path between two definite nodes, we can stop the iterations and exit the loop once the target city has been made internal.

31.5 Building a general purpose *shortest path engine*

Joining two cities in an efficient way is just one specific example of a more general family of optimization problems that can be modelled by a graph. Cities are **nodes** of the graph while roads are directed or undirected **paths** between nodes. The length associated with a road can be generalized as the **cost** of the path. Finally, the distance of a city from the starting city (we will call it the **starting node**) also is its **cost**.

Now, we can write a set of procedures with which one can easily solve any problem pertaining to the *shortest path* family.

This set includes generic, problem-independent, procedures:

- **__Shortest-path_DB.sql**: creates tables NODE, PATH and NPATH and the views of the database. See Script 31.6.
- **__Shortest-path_Engine.sql**: applies Dijkstra algorithm to the problem stored in the database. See Script 31.7.
- **__Display_Shortest-path.sql**: displays the solution of the problem. See Script 31.8.
- **__Display_Nodes.sql**: displays the final state of table NODE (which includes the solution). See Script 31.9.

In addition, for any type of problems **X**, we have to write two scripts:

- **_X_Loader.sql**: loads in tables NODE and NPATH the data describing the specific problem **X**, creates, if needed, the inverse paths then derives the rows of table PATH.
- **X_Solver.sql:** main script used to solve any problem of type **X**; each of them has the minimal structure:

```
createDB InMemory;
   execSQL __Shortest-path_DB.sql;
   execSQL _X_Loader.sql;
   execSQL __Shortest-path_Engine.sql
   execSQL __Display_shortest-path.sql;
closeDB;
```

The script that computes the shortest path between two cities can then be rewritten as Scripts 31.10 and 31.11.

```
create table NODE (NodeID integer not null
                           primary key autoincrement,
                   Name    varchar(32) not null,
                   Mark    integer,
                   Path    varchar(256),
                   Cost    numeric);
create table NPATH(Node1   varchar(32) not null,
                   Node2   varchar(32) not null,
                   Cost    numeric not null);
create table PATH( Node1   integer not null,
                   Node2   integer not null,
                   Cost    numeric not null);
create view EXTERNAL as select * from NODE where Mark = 0;
create view ExtPATH(Node1,Name1,Cost1,Cost,Node2,Name2,Cost2)
            as select P.Node1,C1.Name,C1.Cost,P.Cost,
                      P.Node2,C2.Name,C2.Cost
               from   NODE as C1, PATH as P, NODE as C2
               where  C1.NodeID = P.Node1
               and    P.Node2 = C2.NodeID;
```

Script 31.6 - Generic data structures for shortest path problems [script file _Shortest-path_DB.sql]

```
extract N = select count(1) from NODE;
for step = [1,$N$];
   extract S = select NodeID from EXTERNAL
               where  Cost = (select min(Cost)
                              from   EXTERNAL);
   set neighbors = select Node2 from PATH where Node1 = $S$;
   set newCost = (select Cost1 + Cost from ExtPATH
                  where  Node1 = $S$ and Node2 = NODE.NodeID);
   update NODE
   set    Cost = min(Cost,$newCost$),
          Path = case when Cost > $newCost$
                      then (select Path from NODE
                            where NodeID = $S$)||'.'||Name
                      else Path
                 end
   where  NodeID in ($neighbors$);
   update NODE set Mark = 1 where NodeID = $S$;
endfor;
```

Script 31.7 - The shortest path algorithm [script file _Shortest-path_Engine.sql]

```
if ('$end$' = '');
   write-ab -- No target node has been specified.;
   return;
endif;

extract cost,path = select Cost,Path from NODE
                    where  NodeID = '$end$';
extract nam1 = select Name from NODE where NodeID = '$start$';
extract nam2 = select Name from NODE where NodeID = '$end$';
write-b -- Shortest path from "$nam1$" to "$nam2$": $path$;
write-a -- Cost of this path = $cost$.
```

Script 31.8 - Display the solution of the problem [script file _Display_Shortest-path.sql]

```
set maxSelect = $maxSelectWidth$;
extract maxSelectWidth = select max(length(Path))+34
                         from   NODE;
select Name as Node,Path,Cost from NODE;
set maxSelectWidth = $maxSelect$;
```

Script 31.9 - Display the final state of table NODE [script file _Display_Nodes.sql]

```
insert into NODE(Name) values ('A'),('B'),('C'),('D'),('E'),
                              ('F'),('G'),('H'),('I'),('J');
insert into NPATH values
      ('A','B',85),('A','C',217),('A','E',173),('B','F',80),
      ('C','G',186),('C','H',103),('F','I',250),('H','D',183),
      ('H','J',167),('E','J',502),('I','J',84);
insert into PATH select C1.NodeID, C2.NodeID, P.Cost
                 from   NODE C1, NPATH P, NODE C2
                 where  C1.Name = P.Node1
                 and    P.Node2 = C2.Name;
update NODE set Mark = 0, Path = '', Cost = 999999;
```

Script 31.10 - Loading the data for the Inter-City problem depicted in Figure 31.1 [script file _Inter-City_Loader#1.sql]

```
set script =
    $scriptDirectory$/Case-Studies/Case_Shortest_Path;
createDB InMemory;
   execSQL $script$/_Shortest-path_DB.sql;
   execSQL $script$/Inter-City_Loader#1.sql;
   ask start,end = Starting node:[select Name,NodeID
                                  from   NODE order by Name]
                  |Target node:[select Name,NodeID
                                 from   NODE order by Name];
      if ('$DIALOGbutton$' = 'Cancel' or '$start$' = '') stop;
   update NODE set Mark = 0, Cost = 0, Path = Name
   where NodeID = '$start$';
   execSQL $script$/_Shortest-path_Engine.sql;
   execSQL $script$/_Display_Shortest-path.sql;
closeDB;
```

Script 31.11 - The solver for Inter-City problems [script file **Inter-City_Solver.sql**]

31.6 Ariadne's thread: solving mazes

Mazes are popular brain puzzles that ask players to walk across a convoluted path from an entry gate to an exit gate. In many of them, there is only one route from these gates, a pattern called *unicursal maze* or *labyrinth*. Multicursal mazes allow more than one route between gates.

The small multicursal maze of Figure 31.8 comprises two distinct start-to-exit routes, depicted by lines red and blue. The first one (red) requires **20** steps (a step is a unitary square room) while the last one (blue) is **18** step long, and therefore is the shortest one.

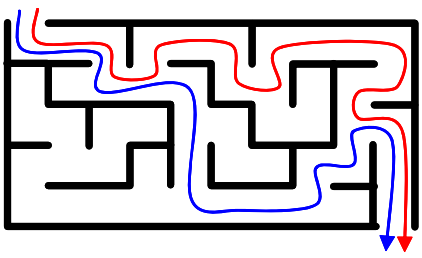


Figure 31.8 - A typical multicursal maze[2]

2. Derived from https://en.wikipedia.org/wiki/Maze

A maze can be described in two equivalent ways. On the one hand, one shows its **walls**, between which the route to find will be drawn. This is the usual way mazes are described (Figure 31.9/left). The second convention consists in showing the available **paths** the maze is made up of, i.e., the space between walls (Figure 31.9/right). We will reason on this latter representation.

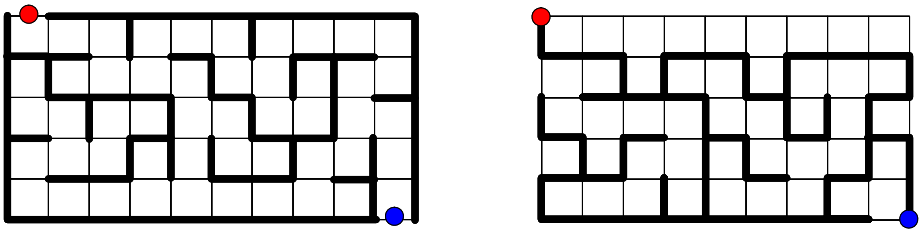


Figure 31.9 - Dual representations of a maze: walls (left) and paths (right)

Considering the grid of Figure 31.10, in which each node is denoted by its row and column identifiers (e.g., B8, E2, etc.), building a maze is simply performed by marking the edges that belong to the maze (Figure 31.9/right).

In this way, solving a maze appears to be a special case of finding the shortest path between cities. The nodes of the grid are some sort of *cities* while the edges are *roads* between cities, all with the same arbitrary *length*, say, 1.

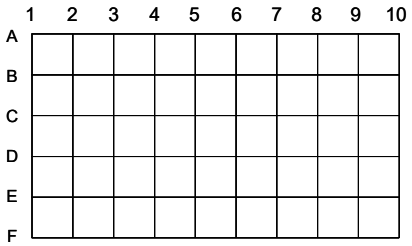


Figure 31.10 - The background grid of a maze

A real maze will comprise quite a lot of nodes and edges. To help players to build them, we suggest to describe them in a text file (named **Maze_map.txt**) where a node is represented by star symbol '`*`', a horizontal edges by '`--`' and a vertical edge by symbol '`|`' (Figure 31.11).

This file is then transformed into a script made up of `insert` queries that load the description of the maze into tables NODE, NNPATH and PATH. This transformation is performed by small Python program **Generate-Maze.py**,[3] which generates file **Maze_Loader.sql**.

3. Its translation into an SQLfast script is left as an exercise.

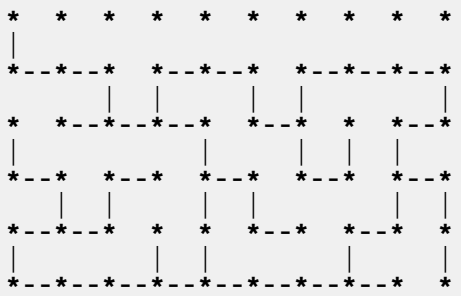


Figure 31.11 - Code of a maze [file **Maze_map.txt**]

```
set script =
    $scriptDirectory$/Case-Studies/Case_Shortest_Path;
   createDB InMemory;
   execSQL $script$/_Shortest-path_DB.sql;
   execSQL $script$/Maze_Loader.sql;
   ask start,end = Starting node:[select Name,NodeID
                                  from    NODE order by Name]
                  |Target node:[select Name,NodeID
                                 from   NODE order by Name];
      if ('$DIALOGbutton$' = 'Cancel' or '$start$' = '') stop;
   update NODE set Mark = 0, Cost = 0, Path = Name
   where  NodeID = '$start$';
   execSQL $script$/_Shortest-path_Engine.sql;
   execSQL $script$/_Display_Shortest-path.sql;
closeDB;
```

Script 31.12 - The maze solver [script file Maze_Solver.sql]

The larger maze of Figure 31.12 is defined in text file **Maze_map_Large.txt**.

Some references on maze algorithm

Some of the main maze solving techniques are described in https://en.wikipedia.org/wiki/Maze_solving_algorithm. The *shortest path* algorithm is briefly mentioned, but does not refer to Dijkstra's algorithm explicitly.

Generating maze is also quite interesting. Reference https://en.wikipedia.org/wiki/Maze_generation_algorithm can be used as a starting point. In particular, it provides Python programs to generate mazes. Cellular automata (see Chapter 32, *Conway's*

Game of Life) with special generation rules can be used to create large mazes. They are described in http://www.conwaylife.com/w/index.php?title=Maze.

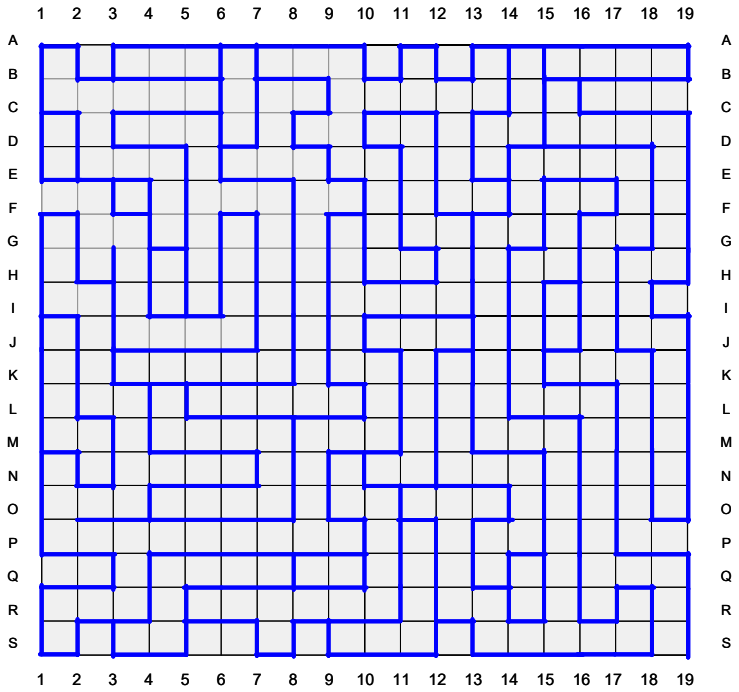


Figure 31.12 - A real size maze (text file **Maze_map_Large.txt**)

31.7 Exploring a planet

Let us imagine that we are to control an exploratory vehicle, sent on a remote planet, to make it move from a starting position to a target position in a hostile 2D space. This vehicle, generally called a *rover*, obeys elementary commands, which are to move one step in one of the four cardinal directions: N, S, E and W. The floor itself is modeled as a regular grid like that of Figure 31.13/left.

In this example, the rover starts at position A5 (the red point) and has to reach target position J8 (the blue point). Each step consumes a unitary electrical power, say, 1 Wh (one watt during one hour). Since the power of the rover is limited, it would be nice to guide the rover so that it consumes the lowest possible energy to accomplish its task.

Clearly, this problem is fairly close to that of the shortest path between cities: the nodes of the grid are kind of cities (named A1, A2, ..., J9, J10) and steps are roads between these cities, with unitary cost.

To make things more realistic, we consider that the rover may have to move across swamp areas that will require more power. Let us say that a step in such an area will cost 3 units. In addition, the floor is scattered with rocks that force the rover to go around them to progress to the target. The grid of Figure 31.13/right shows where the swamp areas (light gray) and the rocks (solid dark gray) are located.

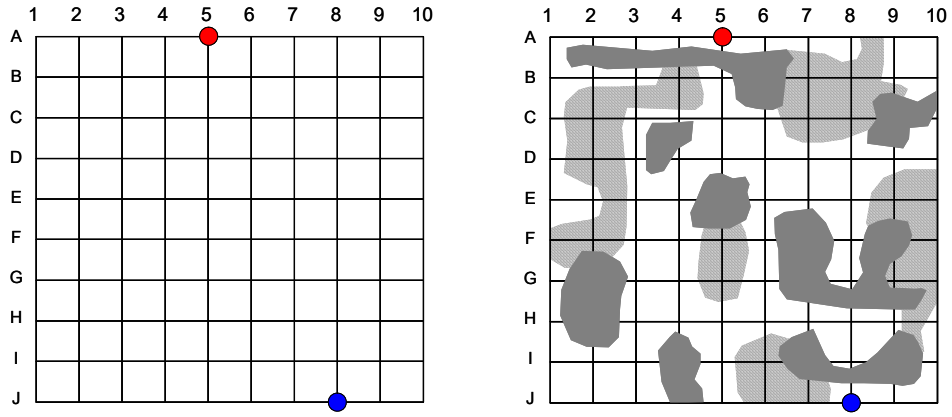


Figure 31.13 - The remote planet the rover has to explore is not as easy as planet earth!

How to represent rocks, which are forbidden areas? The most obvious technique would consist in deleting all the paths partially or completely covered by a rock, as we did in solving the maze problem. We can also keep these paths, to which we assign an infinite cost, that is, practically, 999,999 units, so that they will never be chosen.

The solution for starting position A5 to target position J8 is shown in Figure 31.14/left. Its cost is 26 Wh.

Now, let us consider the second generation rover, which is able to climb rocks securely, but at the higher cost of 6 Wh per rock step. Our algorithm finds another, more sober, path, shown in Figure 31.14/right. Its cost is 22 Wh.

Script 31.13 shows an example of solver for rover problems. The unit cost of plain floor, swamp and rock can be chosen interactively. The script Rover-on-Planet_Loader.sql loads the data for the planet model of Figure 31.13/right.

Some references on rover pathfinding

As we can expect, most references on controlling rovers are classified by the NASA, and therefore are not available. Nevertheless, the reader can find some interesting hints in this reference: https://en.wikipedia.org/wiki/Pathfinding. In particular, it discusses pathfinding algorithms in video games.

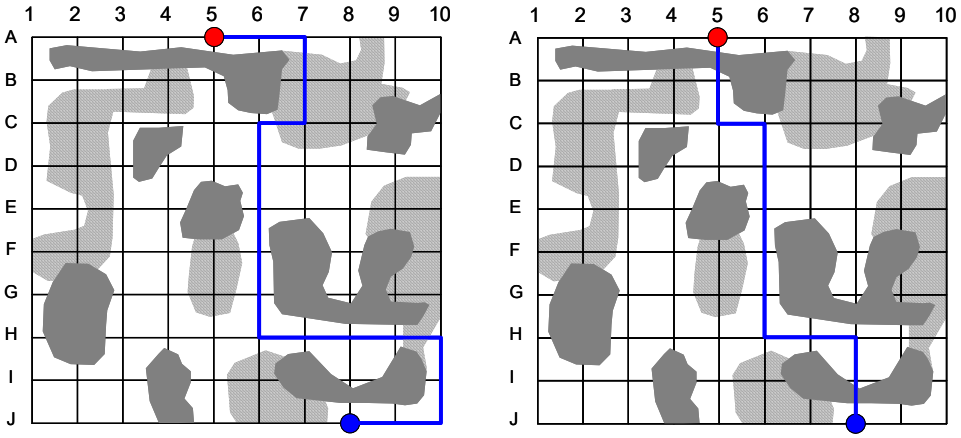


Figure 31.14 - One of the shortest paths between positions A5 and J8 as performed by the first (left) and second (right) generation rovers.

```
set scripts = SQLfast-Tutorials/Chapter-52;
createDB InMemory;
execSQL $scripts$/_Shortest-path_DB.sql;
set c1 = 1;
set c2 = 3;
set c3 = 999999;
   ask-u c1,c2,c3 = Cost of normal step:
                   |Cost of swamp step:
                   |Cost of rock step:;
   execSQL $scripts$/Rover-on-Planet_Loader.sql;
   ask start,end = Starting node:[select Name,NodeID
                                  from    NODE order by Name]
                  |Target node:[select Name,NodeID
                                 from   NODE order by Name];
      if ('$DIALOGbutton$' = 'Cancel' or '$start$' = '') stop;
   update NODE set Mark = 0, Cost = 0, Path = Name
   where  NodeID = '$start$';
   execSQL $scripts$/_Shortest-path_Engine.sql;
   execSQL $scripts$/_Display_Shortest-path.sql;
closeDB;
```

Script 31.13 - The solver for rover problems [script file Rover-on-Planet_Solver.sql]

31.8 Performance and optimization

Finding real country maps comprising a large number of cities **AND** in which the lengths of the roads between them are specified is no easy task. To evaluate the performance of the algorithm developed above we will rely on artificial maps generated as square grids like those used to model mazes. Each node represents a city and each city is linked to its four closest neighbors (except border cities of course). So, an **N**x**N** grid represents a map with $\mathbf{N}^2$ cities and $4 \times \mathbf{N} \times (\mathbf{N} - 1)$ roads.

The goal is to compute the shortest paths from the top left city to all the other cities. Figure 31.15 shows the time needed to get the result for increasing values of **N** (column **Time1**).

N	Cities	Roads	Time1	Time2	Time3
10	100	360	0.124	0.109	**0.001**
20	400	1520	0.624	0.468	**0.156**
30	900	3480	1.872	1.123	**0.812**
40	1600	6240	4.446	**2.215**	2.558
50	2500	9800	9.236	**3.776**	6.240
60	3600	14160	17.175	**6.048**	13.010

Figure 31.15 - Computing time (in seconds) for various map sizes

Column **Time2** shows much better results. They are obtained by indexing table PATH on column (Node1). This index speeds up the evaluation of expressions **neighbors** and **newCost**.

Third column **Time3** shows the execution time of the shortest-path engine coded in Python (program **Shortest_Path.py**). Its times are the best for small problems but become, by far, the worst for large values if **N**.

Figure 31.16 represents graphically the execution time of these three techniques, repectively named Plain (**Time1**), Index (**Time2**) and Python (**Time3**). The fourth curve translates the theoretical law $\mathbf{N} \log \mathbf{N}$. It allows to observe that the result of the indexed technique is quite close to the theoretical performance of Dijkstra's algorithm, the complexity of which is $O(\mathbf{N} \log \mathbf{N})$.[4]

To test other grid sizes from N = 4 to 64, use predefined script **Large-grid-maps_Solver.sql**.

4. The *complexity* of an algorithm is a measure of the limit of the behaviour of its execution time (or any other resource it needs, such as memory space) when the size of the problem (here **N**) increases. Writing that the execution time of Dijkstra's algorithms varies as $\mathbf{N} \log \mathbf{N}$ means that if the execution time is **T** for a certain value of **N**, then, for a grid of $2\mathbf{N}$, the execution time will be a bit higher than $2\mathbf{T}$ (linear) but much better than $4\mathbf{T}$ (quadratic).

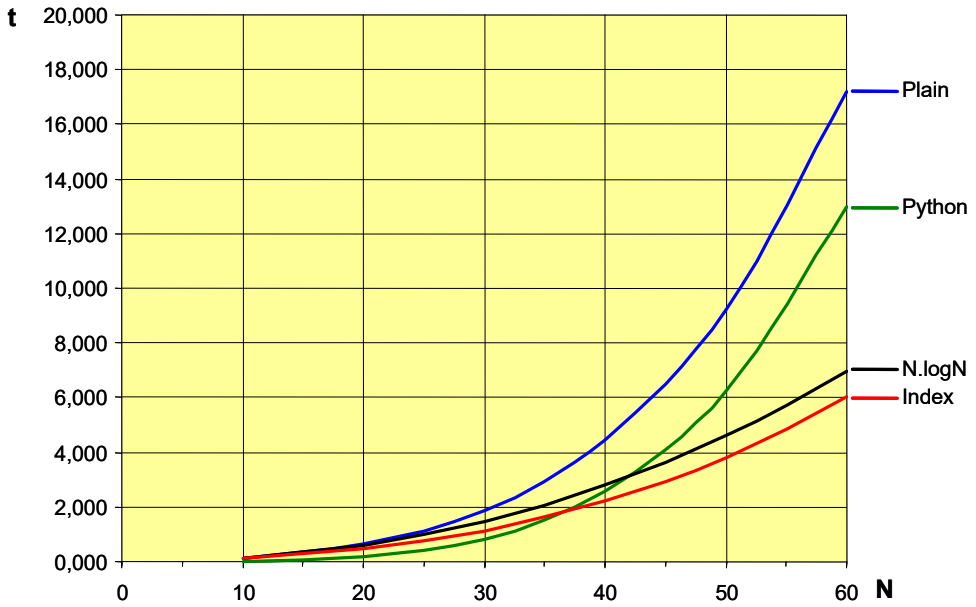


Figure 31.16 - Comparison of the performance of various algorithms

31.9 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory **SQLfast/Scripts/Case-Studies/Case_Shortest_Path**. They can be run from main script **Shortest-MAIN.sql**, that displays the selection box of Figure 31.17.

These scripts are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.

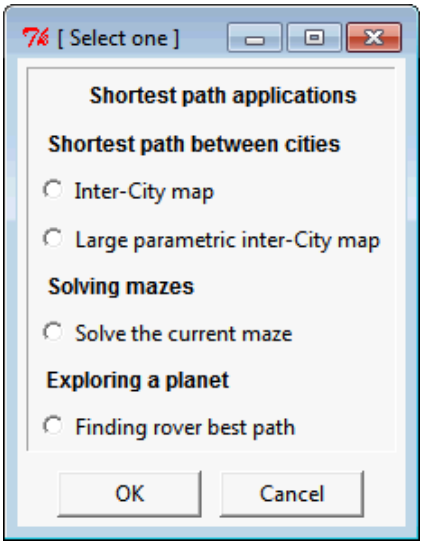


Figure 31.17 - Selecting and solving a best path problem

Case study **34**

Blockchains

In this study, we examine some fundamental aspects of blockchains, particularly the security of data and the way(s) it is achieved through cryptographic transformations. Basically, a blockchain is a historical database in which the description of operations, generally called transactions, are stored in chronological order. Once recorded, the data of a transaction can never be deleted nor modified.

The document first introduces the elements of cryptography necessary to build a blockchain, notably secure hashing, and symmetric and asymmetric key encryption. Then, it describes the distinctive aspects of blockchains independently of its application domain and applies them to cryptocurrencies. Finally an experimental toolbox, comprising a collection of functions designed to manage and explore blockchains, is built step by step.

Keywords. blockchain, blockchain explorer, proof of work, distributed database, cryptocurrency, trust, security, cryptography, RSA, AES, secure hashing

34.1 Introduction

Blockchain is certainly one of the most popular techniques currently discussed, both in the general public and in scientific circles. Initially known as a cryptocurrency implementation platform, its applications have quickly spread beyond the financial world. Now applied (or suggested to apply) to the most diverse fields of human

activity, blockchain is considered by some futurologists as **the** disruptive technology that can lead to major societal changes.

Far from these speculations, whose validity has yet to be demonstrated, this study has a particularly modest objective: to illustrate in practice the technical bases of blockchain technology.

In short, a blockchain will be seen a a *massively replicated, secure, append-only, historical database* supported by a peer-to-peer network. The complexity of its data structures and management rules makes blockchain a remarkable application of database technology. That alone justifies a full chapter in this series of case studies

Since blockchains make intensive use of cryptographic operations, we give in Section 34.2 a very short introduction of those used in this technology. In Section 34.3 we recall the principles of blockchains, that we apply to cryptocurrencies in Section 34.4. The remaining of this study (from Section 34.5) is devoted to the analysis and implementation of some of the main functions of a didactic blockchain management system.

34.2 Elements of cryptography

Cryptography is a set of techniques devoted to the transformation of explicit information into a meaningless string of bits. Their main objective is to **securely** store or communicate information. The concept of *security* encompasses several issues, such as preventing unauthorized third parties to read the information, checking its integrity or preventing parties to contest its authenticity. We will briefly describe the principles of three important cryptographic techniques used by blockchains. In this presentation, we will call *message* any kind of source user data that we intend to process through cryptographic techniques: text, file, picture, program, etc.

Secure hashing

Hashing is the derivation of a (seemingly) random fixed-length bit string from a source message. This derivation in performed by a hash function (let us call it **h**) that implements complex mathematical operations. The resulting bit string is called the **hash** or the **digest** of the message. A good hash function must satisfy the following properties:

1. **h** is deterministic: applying **h** to source message **M** always provides the same result.
2. **h** is not invertible: there is no way to compute message **M** from its hash. In other words there is no function **g** such that **M** = **g**(**h**(**M**)).
3. **h** is injective: applying **h** to two different messages yields two distinct hashes: if **M1** ≠ **M2**, then **h**(**M1**) ≠ **h**(**M2**). This property is also called *collision resistance*.

One of the most popular hash function is SHA256, that produces hashes of 256 bits. It is available in the SQLfast library under the name hash(). The fact that (good) hash functions have no inverse is of course important but it does not prevent attackers to try to *crack* a hash, that is to guess the (unknown) source message of a (known) hash through a brute-force attack such as that of Script 34.1, where messages(N) is an iterator that yields successive bit strings of length **N**, from 'ob0000...0000' to 'ob1111...1111', and Hash is the hash of which one tries to guess the source.

```
def guessSource(Hash):
    # Try messages of length 1 to 1000
    for N in range(1,1001):
        # Create an iterable object from class 'messages'
        Nmessages = messages(N)
        for source in Nmessages:
            if hash(source) == Hash:
                return source
    return ''
```

Script 34.1 - Guessing the source message of a known hash through a brute-force method

Three comments on this approach:

– The number of potential messages to check is extremely large: 2^N for each value of **N** (generally, we do not know the length of the actual message), so that the exploration is not feasable with current computing technology. A machine capable to compute *one billion hashes per second* would take about 300 years to examine all the combinations of 64 bits, i.e., about 10 printable ASCII characters.

– Property 3 defines *perfect* hash functions. Actually, there is no way to ensure that, therefore, there is no guarantee that the source message found with this method is the true one, intended by the sender. Good functions make collisions (the fact that two distinct messages are assigned the same hash) extremely rare, in such a way that they are considered *practically* injective. Such good functions are described as *secure*, hence the name *secure hash algorithm* or SHA.

– More refined cracking techniques could be based on the knowledge of parts of the original message, such as its length or a constant signature or introduction. However, a property of secure hash functions makes such hints useless: two almost identical source messages produce very different hashes.

  This is illustrated by the following experiment: computing the hash of these two messages, where only the case of the first letter is different,

```
compute hash1 = hash('what is the best database language?');
compute hash2 = hash('What is the best database language?');
write $hash1$;
write $hash2$;
```

produces these very different hashes (displayed in hexadecimal):

```
e6b67be466c63231af78a4bece714def58f33cbc48911b7319579cee242b0415
d9879a0017d31dd03144ce0f4be8e0f1b46d503c27c1ee4c87c49d4ee49548c1
```

It is important to understand that hashing does not preserve the contents of the source message, contrary to the next two techniques.

Symmetric key cryptography

Encryption is a reversible transformation of a source message that produces a meaningless byte (or character) string. This transformation is performed by a complex algorithm based on an *encryption key*. The encrypted message can be securely transmitted or stored. To recover the source message, the encrypted message must in turn be transformed by a decryption algorithm using an *decryption key*.

In *symmetric key cryptography* the encryption and decryption keys are the same. To protect the encrypted messages, this key must be kept secret, only known by the sender and the recipient(s) of the messages.

The SQLfastUDFlib library includes two primitives to encrypt and decrypt messages. They are particularly fit to process plain character strings. The following script illustrates their usage:

```
set message0  = What is the best database language?;
set secretkey = my secret key;
compute crypto   = encrypt('§message0§','§secretkey§');
compute message1 = decrypt('§crypto§','§secretkey§');
write $message0$;
write $crypto$;
write $message1$;
```

With the following result, showing that the original message has been preserved:

```
What is the best database language?
xOGB54XM5YXoiNCF29LslJPJxObG1oHeypnZ2o7a2sTZyrM=
What is the best database language?
```

Formally, if **M** is the source message and **K** the symmetric key:

```
decrypt(encrypt(M,K),K) = M
```

To process pure binary messages, functions generateSYMkey, encryptSYM and decryptSYM of LStr.py library are more appropriate. Function generateSYMkey generate a pseudo-random byte secret key of given length (expressed in base64). The usage of these functions is illustrated here below. The length of the key is 12 bytes (i.e., exactly 16 in base64).

```
set message0  = What is the best database language?;
 function secretkey = LStr:generateSYMkey 12;
 function crypto    = LStr:encryptSYM  {$message0$},$secretkey$;
 function message1 = LStr:decryptSYM  {$crypto$},$secretkey$;
 write $message0$;
 write $secretkey$;
 write $crypto$;
 write $message1$;
```

With the expected result:

```
What is the best database language?
ue9ihdMB7hnUC/sn
zM2a3YjNwGKr0NN1pZTm4pXJmt3Jxq61nIjatrGW6M_cyng=
What is the best database language?
```

The SQLfast functions are based on the *Vigenère* symmetric key algorithm. This technique is particularly simple and the encryped messages are reasonably hard to crack. More secure algorithms are available such as the *Advanced Encryption Standard* (AES). The main weakness of symmetric cryptography is that all the senders and recipients must keep the key secret. Once the secret key has been generated, it must be distributed over a secure channel, for example through asymmetric encryption.

Asymmetric public key cryptography

Asymmetric encryption uses two keys. The *public key* is used by all the senders who intend to send messages to a recipient. This key is specific to the recipient. It is public and can be found in some sort of public directory. When the recipient receives a message that has been encrypted by her public key, she can decrypt it with a second key, her *private key*. Only the recipient possesses this private key, which is then easier to protect.

To perform asymmetric encryption, SQLfast offers three functions similar to those of symmetric cryptography: generateRSAkey (that yields both public and private keys), encryptRSA and decryptRSA of LStr.py library. They use RSA, among the most popular asymmetric cryptographic algorithms. The usage of these functions is illustrated in the following snippet. The length of the public key is 420 bits (converted in base64).

```
set message0  = What is the best database language?;
function secretkey,publickey = LStr:generateRSAkeys 420;
function crypto    = LStr:encryptRSA {$message0$},$secretkey$;
function message1 = LStr:decryptRSA {$crypto$},$publickey$;
```

Formally, if **M** is the source message, **P** the public key and **V** the private key:

```
decryptRSA(encryptRSA(M,P),V) = M
```

A major drawback of the asymmetric technique is that the public key must be at least as long as the longest message to encrypt. To overcome this problem, one can encrypt long messages by slices the length of the key or to use symmetric encryption and to encrypt the symmetric key through an asymmetric technique.

Blockchain management makes intensive use of asymmetric cryptography to verify the authenticity of transactions. However, the roles of *sender* and *recipient* are inverted. Only the sender is allowed to encrypt a message but all the members of the blockchain, the recipients, may decrypt it. So, when a pair of (*public, private*) keys are generated by the RSA algorithm, the originally *public key* is used as a *secret key*, which must be carefully protected by the sender, while the originally *private key* is made *public*, allowing everybody to decrypt messages. To summarize:

standard	blockchain
public key	secret key
private key	public key

34.3 The blockckain paradigm

Basically, considered as a service, a blockchain is a *journal*, or a *ledger*, in which a community of members securely record information on operations of interest to them. Quite often, these operations concern the creation, modification and exchange of critical resources such as money, energy, goods, services, stock market securities or real estates.

In the (so far *usual*) real world, such operations are controlled by trusted intermediaries such as banks, public administrations, stock exchange authorities, lawyers or notaries public.

The most significant characteristic of blockchains is that there is no such intermediaries in charge of controlling the legitimacy of the operations. Each member is responsible for the validity of her operations, but also of the validity of the operations carried out by all the other members of the community.

This rises the issue of **trust**. As a member of the community, how can I be certain of the **validity of an operation**? When a house is being sold, is its seller really the owner this house? When a customer pays me the product I sent her, is the balance of her account sufficient?

In addition to the validity of the operations themselves, I must be confident that the information recorded in the ledger will never be falsified; in other words, that the content of the blockchain never can be altered once it has been recorded (it must be **immutable**).

The trust in the information of a blockchain is ensured by a set of rules that define the way operations must be recorded. These rules form the protocol of this blockchain. Simply stated, the blockchain technology must meet the following conditions:

- *Reliability*: the operations described in the blockchain are definitively effective as soon as they have undergone a validation process.
- *Non falsifiability*: data can be appended to the blockchain, but, once recorded, they can no longer be deleted or modified.
- *Transparency*: the recorded data are public; all members of the community can review them and check their validity.
- *Robustness*: the blockchain must resist all kinds of attack, be they accidental or intentional.
- *Anonymity*: the identity of the members involved in an operation is protected, except in some special situations, such as transactions involving a bank (to prevent from money laundering), this identity is mentioned via a meaningless pseudonym.

The absence of a trusted third party has an important consequence: there is no central service responsible for storing the blockchain. It is simply copied on the computer of each member of the community, thus contributing to its robustness.

Members

A member who joins the community receives three specific information items:

- an unique *identifier*, which is a random character string (principle of *anonymity*) that will be used to identify her in any transaction she will be a partner of,
- a *secret key* with which she will *encrypt* her data,
- a *public key* with which all the members of the community can *decrypt* these data.

These keys allow data to be securely recorded in the blockchain through asymmetric cryptography. The couples (*identifier*, *public key*) are public and can be known by the community, or at least by a part of it.

Transactions

Each operation related to the resource managed by the blockchain is executed as a *transaction*. The properties of the transaction are recorded in the blockchain as soon as it has been executed, but it will be effective only when it is validated.

Let us suppose that I sell my house to a buyer for 150,000€, the data of this transaction are recorded in the blockchain: my id, the id of the buyer, the id of the house, the price we agree upon and the date. The sale will be effective when these data have been validated: I am the owner of the house, the house is free from any debt, the price is plausible, the buyer exists and this amount is available on her account.

When the data of the transaction have been recorded in the blockchain, they are vulnerable, notably prone to all kinds of fraud, such as a member discretely modi-

fying some data. For example, my buyer could be tempted to lower the value of the house to, say, 125,000€. To prevent this fraud, I modify the transaction in this way:

- When all the data of the transaction have been collected (let us call them **T**), I encrypt them with my secret key (**S**); the result is the *signature* of the transaction:

  signature = encrypt(**T**,**S**)

- I add this signature to the transaction data.
- I make this augmented transaction public.

Therefore, every member of the community can examine this transaction, decrypt its signature, compare the result with the source data and conclude that the source data are authentic or corrupt. Denoting by **P** the public key associated to **S**, **T** is considered correct if:

decrypt(**signature**,**P**) = **T**

Since only the creator of the transaction can encrypt the data, there is no way for a fraudster to alter the source data.

To summarize,

- a transaction describes an operation,
- it is complemented by its signature
- it is distributed to the members of the blockchain,
- who can validate it,

then it becomes effective (or rejected). However, as we will see, recording a transaction in the blockchain is a bit more sophisticated.

Blocks and blockchains

For reasons that will be discussed later, when a sufficient number of transactions have been created, they are grouped together to form a **block**. This block is assigned an identifier, that becomes a part of it, as well as the identifier of the last block that has been created. So, each block (but the first one) references the previous block, thus forming a chain of blocks, hence the term **blockchain**.

The identifier of a block is computed as the *hash* of its contents, that is:

$$\begin{aligned} Bbody &= trans_1 + trans_2 + \ldots + trans_n + rectime + previous \\ Bhash &= hash(Bbody) \\ block &= Bhash + Bbody \end{aligned}$$

where $trans_i$ is the ith transaction collected since the creation of the preceeding block,
n is the standard capacity of a block,

`rectime` is the timestamp of the block,
`previous` is the hash of the previous block,
`Bhash` is the identifier of the block being built,

The quality of the hash function, based on SHA256, ensures that there will be no collision between two blocks, at least in a forseeable future!

The block structure provides a second level of security against data corruption, whether accidental or intentional.

Suppose that the creator of a transaction wants to fraudulently modify it, for example to increase the price of the house he sold. Since he possesses the secrete key of the transaction, he can modify the transaction data, recalculate the signature then replace the transaction in its block, hoping that no one will notice.

Unfortunately, the block structure makes the task a bit more complex. The hash of the block now is invalid and must be recalculated then replaced in its block. Therefore, the next block becomes invalid, since it references the fraudulent block. It must be updated as well, and so on until the last block of the chain has been modified. Worse, all the members of the community have a copy of these blocks. The fraudster therefore has no choice but to ask them to replace these blocks with the new ones. Hoping that nobody will notice!

Validation of a block

The last step of the protocol consists in validating the block we have built and publishing it to make it accessible to all the members of the community. This task is fairly tricky and will be best explained in the specific application domain of cryptocurrencies.

34.4 Application: cryptocurrencies

One of the most popular applications of blockchains is *cryptocurrencies*, and more specifically *bitcoins*, managed through the *Bitcoin* protocol. When I want to buy a book that cost 20 €, I usually transfer this amount of money from my account to that of the bookshop. For this, I make use of the services provided by a bank, which both (me and the bookshop) consider a *trusted intermediary.*

The objective of the Bitcoin protocol is to allow us to execute such operations without relying on the services of a bank or of any other intermediary such as PayPal. If the bookseller and myself are members of the same cryptocurrency community, we just have to record a *money transfer transaction* in the blockchain of our community.

Following the description given in the preceeding section, such a transaction will normally include *my id*, the *id of the bookseller*, the *price* of the book and the *date* of the transaction. And of course the *signature* of these data.

Note. In the following, we describe a protocol similar to that of Bitcoin but strongly simplified, in order to comply with the didactic objective of this series of case studies. Motivated readers are invited to refine this protocol to include some specific aspects of Bitcoin, such as *multi-output transactions* and *miner conflicts*.

The member id

When joining the blockchain system, each member receives a *public key* through which all the members can decrypt the signature of his transactions. This key is unique and therefore could be used to identify the member. However, the public key is a fairly large character string, so that a new, shorter, identifier is derived from it by hashing this key. In the Bitcoin protocol, the member id, called *Bitcoin address*, is 160 bit long.

The transactions

First of all, some terminological clarification: in addition to *member* and *member id*, we will use the terms *account* and *account id*. A member is the agent who creates and manages an account. There is a one-to-one relationship between members and accounts: each member manages one account and each account is managed by one member. However, the blockchain only knows accounts. We will keep the concept of member when necessary, notably to give operations an intuitive interpretation.

To make things concrete, we consider three types of transactions.

- **registering** a new account. *Input*: recording date. *Output*: an account id, a secret key and a public key.
- **depositing** an amount of money on an account. *Input*: recording date, id of the source account and the amount deposited.
- **transferring** an amount of money from a source account to another account. *Input*: recording date, id of the source account, id of the recipient account and the amount transferred.

When recording a transaction, we add to its base data a *transaction id* and its *signature*, computed with the secret key of the member who initiated the operation.

Block building

When a transaction has been recorded, it is sent to the members of the blockchain who are in charge of validating them. These members are called **miners**, for a reason we will explain below.

First, miners verify that the data of each transaction are not corrupt by comparing them with the data of the decrypted signature. Then, they analyze the components of the transactions to verify that they are semantically correct: for example, the accounts ids reference accounts that really exist and the amount of money is available, therefore preventing a member to spend more than the balance of his account.

When the transactions a miner has received are proved to be valid, they are grouped to form a block. This block is signed with its hash and sent to the other members of the blockchain to (tentatively) become its new last block. This step is quite fast and does not require powerful machines.

Finally, we note that the hash of a block is guaranteed to be unique and therefore can be used as the **identifier** of the block in its blockchain.

Block mining

A first problem that arises is that all the miners carry out the same task, so that many candidate blocks are broadcasted to the community almost at the same time. Hence the dilemma: which one to choose? In a context in which there is no central authority, the idea is simple, the first one is the winner. To make this rule effective, the process of block building must be slowed down, in such a way that the candidate blocks are received in a (seemingly) random way.

Another problem is that, due to the ease with which this task can be carried out, quite many members can pretend to the status of miner, therefore opening the door to the risk of inappropriate behaviour and fraud. It is important to limit the number of miners to those who really are interested in, and therefore willing to contribute to, the proper functioning of the blockchain. Therefore, the miners are required to prove that they enjoy a certain level of resource that gives them an outstanding status among the community that, in turn, grant them its trust. The most popular of these resources are their *computing power* (the miner must show his *Proof of work*) and their *notoriety*[1] (the miner must show his *Proof of stake*). We will describe the first of them.

To solve these problems, the blockchain protocol imposes a special constraint on the hash with which the block is signed: the block must be accompanied by a short bit string, called the **nonce** of the block, that, if appended to the block, produced a hash the **k** first bits of which are zeroes.[2] Satisfying this constraint consists in finding a nonce. There is no clever way to find it: one just has to try all the combinations of **k** bits until one of them produces a good hash. That is:

```
body = transactions + rectime + previous
nonce is valid if
   hash(body + nonce)[0:k] = '0b000...000'
```

This nonce and the hash it yields are added to the block, which is then sent to all the members as the suggested last block of the chain. Figure 34.1 illustrates the structure of a blockchain by showing the contents of two successive blocks. In each block, Bhash denotes the valid hash, used as the block identifier.

Checking that blocks are valid is quite fast, so that every member, whether miner ou not, can verify that a block is not corrupt. On the contrary, finding a valid nonce

1. The notion of *notoriety* can be quite diverse: the current balance of your account, the average balance in the last year, your activity in the blockchain, your social position, etc.
2. In 2019, k = 72

is very expensive in computing time. In particular, it requires a large computing infrastructure. So, broadcasting a block with a valid nonce is a proof that one has worked *very hard*. In other words, a valid nonce is a **proof of work** that any other member can verify.

To make things even more challenging, only the first miner who finds a valid nonce wins the competition. We could wander what may motivate miners to perform such a hard and expensive work for such an uncertain result. The answer is both simple and universal: *money*! The winner receives a certain amount of crypto-money as compensation for his work. Hence the name of *miner*: working hard to hopefully find, from time to time, some small golden nugget!

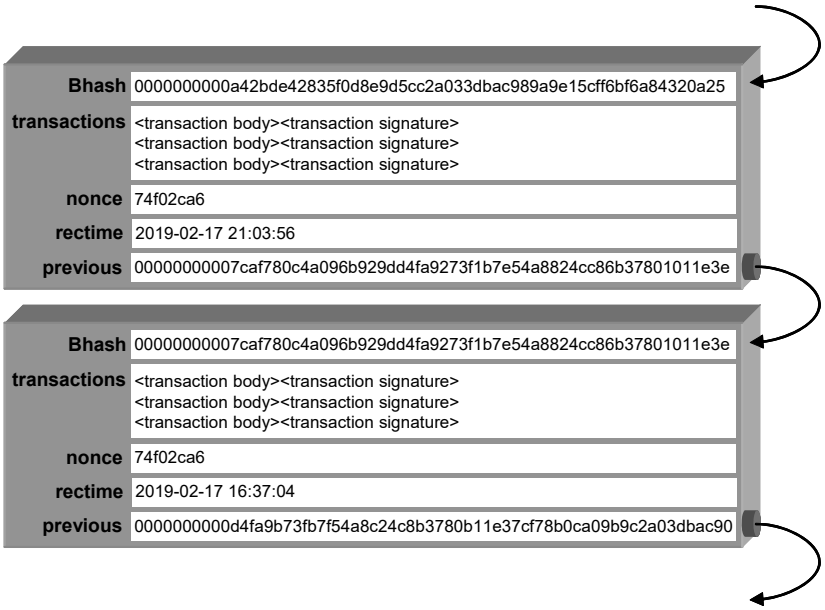


Figure 34.1 - Fragment of a blockchain

This description of the blockchain mechanisms is just an introduction to the main principles. It leaves out many problems and their solution. More details can be found in countless publications both in libraries and on the web. As far as this study is concerned, we now know enough to start the implementation of some components of a simplified prototype blockchain.

34.5 The BLOCKCHAIN toolbox

The programs that implement a blockchain system *in the real world* form a complex distributed system. In the context of this series of case studies, the objective of which basically is didactic, it would not be realistic to develop a full-fledged block-

chain software. We will rather develop a set of independent functions sharing a database representing an elementary but representative blockchain.

The application chosen is that of a cryptocurrency based on the presentation of Section 34.4. The function set includes the three basic transactions of **registering an account**, **depositing an amount** of money and **transferring an amount** of money from a source account to a recipient account. These transactions will then be processed by a **validation** function and gathered into **blocks**, including their **mining**. To these functions, that form the basis of a blockchain application, we add a set of **visualization** functions that will allow us to examine the data with different degrees of granularity.

The basic mining function will be discussed and implemented, but we will leave aside the aspects of *distribution* (data replication in each member machine in particular), *reward* (remuneration of miners), *competition* (the first miner "*takes it all*") or *conflict resolution*. Similarly, transfer transactions between accounts are limited to a single recipient account (no *multiple ouputs*). These extensions can be developped without problem once a solid basic architecture has been defined.

Following the approach of the case studies series, a blockchain will be designed and implemented as a relational database application, naturally developed in the SQLfast language and environment.

34.6 The database

The main data structures include three public tables, containing information on the *transactions*, the *blocks* and the *public keys* of the members.

34.6.1 The transactions (table BTRANSACTION)

There are three types of transactions: registering a new account (Operation = REGISTER), depositing an amount of money on an account (Operation = DEPOSIT) and transferring an amount of money from a source account to a recipient account (Operation = TRANSFER) .

Let us consider the example of a transfer transaction of an amount of 100 units of money from **Mary**'s account to **Luke**'s account. Figure 34.2 shows how we will code this information. It indicates the *type of operation*, (Mary's) *source account id*, (Luke's) *recipient account id*, the *amount* of money transferred, the *recording date* of the transaction and finally the *signature* of the transaction (only a part of the signature is shown).

To this basic data, we add the *reference of the block* (its Bhash) in which the transaction will be inserted. Then we assign to each transaction an *id* (unique in the blockchain) formed by the id of the source account to which we add the recording date. This pattern is valid provided that the transactions initiated by an account are created on distinct dates, which is not a particularly hard constraint since this date is detailed to the second.

Operation: TRANSFER
Source: FT7UzFgSpMmNsnlet_ooAKV4wWTyJAaLrnYH6TRtIBw=
Recip: QWA7QEjrvrsg1lpALJi34f8Ksrzb0ANfa5tqsylm7Xw=
Amount:100
RecTime: 2019-05-23 15:58:3
Signature: C75d4HLpm9STKHj[..]h-blt2uCmMM_mfK

Figure 34.2 - Data describing a TRANSFER transaction

Finally, in order to *facilitate experimentation*, transactions will be provided with a short numerical identifier, called a *transaction key*, which will be assigned automatically. This identifier is obviously not part of the blockchain model.

These components are translated into the BTRANSACTION table[3], the structure of which is shown in Script 34.2. Column TransKey will contain the transaction key, TransID the transaction id, Bhash the reference of the block. The other columns have an obvious meaning.

```
create table BTRANSACTION(
   TransKey   integer      not null primary key autoincrement,
   TransID    varchar(72)  not null unique,
   Bhash      varchar(48)  references BLOCK(Bhash),
   Operation  char(12)     not null,
   Source     varchar(48),
   Recip      varchar(48),
   Amount     decimal(16,10),
   RecTime    datetime     not null,
   Signature  varchar(512) not null);
```

Script 34.2 - Schema of the table describing the transactions

34.6.2The blocks (table BLOCK)

A *miner* creates a block as soon as a sufficient number of candidate transactions have been received. Basically, a block consists of a *block identifier*, a sequence of *transactions*, its *creation date* and the identifier of the *previous block*.

The identifier of the block is obtained by hashing its content to which a **nonce** is added. This nonce will be represented in the BLOCK table as a long integer represented by a decimal character string. The hash is translated into base64 for convenience. Finding this nonce, which is particularly time-consuming, constitutes the *proof of work* of the block validation. In the standard version of the toolbox, k = 20, which requires a mining time from 5 to 30 seconds for blocks of 5 transactions.

3. 'TRANSACTION' would have been a better name. Unfortunately, it is an SQL reserved word!

The maximum number of hash computations is 5,000,000. These parameters can be changed if needed.

Transactions are not integrated into the block as usually described in the literature (see Figure 34.1) but are associated with it via foreign key Bhash in table BTRANSACTION. We will simply indicate the number of transactions associated with the block.

A block that has just been created is submitted to the community of miners. At this stage, it is only a project that has yet to be validated and selected as the official block. Blocks are characterized by their *status*, which indicates their validation status: 0 for a validated block and 1 for a selected block.

Just as we have done for transactions, once again to facilitate experimentation, we will associate a numerical identifier, called a *block key*, to the blocks. The translation into a table structure is illustrated in Figure 34.3. Column BlockKey is the block key, Bhash the hash of the block, and therefore its id, Transact is the number of transactions.

```
create table BLOCK(
   BlockKey   integer      not null primary key autoincrement,
   Bhash      varchar(48)  not null unique,
   Transact   integer      not null,
   Nonce      varchar(32),
   RecTime    datetime     not null,
   Status     integer      not null default 0,
   Bprev      varchar(48) references BLOCK(Bhash));
```

Script 34.3 - Schema of the table the blocs

34.6.3The public key directory (table DIRECTORY)

To transfer an amount of money to a recipient account, we are supposed to know its id, just as, to make a bank transfer, we need to know the number of the recipient account. To validate a transaction we must decrypt its signature to verify that it corresponds to the values of the different fields of this transaction. This decryption is performed by the public key of the creator of the transaction, which is the member who owns the source account.

We must therefore create a *directory* that, for each account id, provides its public key. This directory is stored in the DIRECTORY table whose schema is shown in Script 34.4. A column has been added indicating whether (1) or not (0) the account owner is a miner.

```
create table DIRECTORY(
   AccountID  varchar(64)   not null primary key,
   PublicKey  varchar(1400) not null,
   Miner      integer       not null default 0);
```

Script 34.4 - Schema of the table providing public keys

34.7 Experimental data

The data in the BTRANSACTION, BLOCK and DIRECTORY tables are public, and therefore accessible to all members of the blockchain.

To these so-called *functional data* (i.e., necessary for the operation of the blockchain), we must add those that will allow us to experiment with the concepts of blockchain. These data, which do not exist in real implementations (we represent them in red in the table schemas), allow us to experimentally perform, in a simple and intuitive way, the different operations on our block chain. We will call them *experimental data*.

First, it should be noted that tables BTRANSACTION and BLOCK already contain experimental data in the form of the transaction keys (TransKey column) and the block keys (BlockKey column).

The data of the accounts we will play with will be stored in ACCOUNT table. An account has an id generated by hashing its public key and displayed in base64 character code. Memorizing and manipulating these long and meaningless character strings will not be particularly comfortable. For this reason, we associate with each account a nickname easy to remember, for example a person name, which we call *private name* (PrivName column). So, we will describe a transfer operation in this way:

Mary *transfers an amount of 100 to* Luke*'s account*

obviously more natural than its technical equivalent:

an amount of 100 is transferred
from account `FT7UzFgSpMmNsnlet_ooAKV4wWTyJAaLrnYH6TRtIBw=`
to account `QWA7QEjrvrsg1lpALJi34f8Ksrzb0ANfa5tqsylm7Xw=`

When an account is created, a pair of keys are generated to compute (by the secret key) and then decrypt (by the public key) the signature of transactions. In a real implementation, only the owner of the account knows its secret key, which he must store and protect with the greatest care. In this experimental prototype, we will play all the roles: account creator, source, recipient and miner. We will therefore need to know the secret keys of all the members of the blockchain!

These experimental data are stored in table ACCOUNT. It contains the *account id*, the *private name*, the *secret key* and the *time* the data of the account were recorded

(Script 34.5). There is no need to store the public keys since they already are available in table DIRECTORY.

```
create table ACCOUNT(
   AccountID  varchar(64)   not null primary key,
   PrivName   varchar(64)   not null,
   SecretKey  varchar(360)  not null unique,
   RecTime    datetime      not null);
```

Script 34.5 - Schema of the table describing the accounts

34.8 Account registration

Creating an account leads, on the one hand, to the creation of an RSA key pair (secret key and public key) allowing its owner to execute transactions on this account and, on the other hand, to the creation of a transaction that formalizes the existence of this new account.

The whole operation comprises five steps, shown in Script 34.6:

```
ask name = Private name:;
function skey,pkey = LStr:generateRSAkeys 1700;
function aid = LStr:hash {$pkey$},2;
set      dat = $date$ $clock$;
set      tid  = $add$-$date$_$clock$;
set      op  = REGISTER;
function sig = LStr:encryptRSA {$tid$;$op$;$aid$;$dat$},{$skey$};
insert into ACCOUNT (AccountID,PrivName,SecretKey,RecTime)
       values ('$aid$','§name§','$skey$','$dat$');
insert into DIRECTORY (AccountID,PublicKey)
       values ('$aid$','$pkey$');
insert into BTRANSACTION
      (TransID,Operation,Source,RecTime,Signature)
       values ('$tid$','$op$','$aid$','$dat$','$sig$');
```

Script 34.6 - Script creating a new account (excerpts)

1. Computing the *secret* and *public keys*. We use function generateRSAkeys of LStr library. The size of the secret key must be greater than or equal to the size of the strings it must encrypt. A 1700-bit key is sufficient to encrypt a transfer transaction, the largest of the three types. The length of the secret key thus generated, expressed in base64, is 280 characters and that of the public key is about 1330 characters. These keys are stored in variables skey and pkey.
2. Computing the *account id*. It is obtained by hashing the public key using the hash function of LStr library. The second parameter of the function specifies

the format of the result: hexadecimal (1) or base64 (2). We choose the shorter base64 format. This value is computed in variable aid.

3. Inserting a row describing the account in table ACCOUNT. We record the *account id*, the *private name* the experimenter will associate with it, the *secret key* and the *date* the account was created. This date consists of the date itself followed by the current time of the day, information obtained by the date and clock functions.
4. Inserting a row publishing the *public key* associated with the *account id* in table DIRECTORY. By default, a new member is not a miner.
5. Inserting a row describing the registration transaction in table BTRANSACTION. The *transaction id* (column TransID, the value of which comes from variable tid) is made up of the account id concatenated with the creation date. The *signature* of the transaction is obtained by encrypting with the secret key the string formed by the transaction id, the transaction, the account address and the date the account was created. An example of contents of the new BTRANSACTION row is shown in Figure 34.3.

TransKey	259
TransID	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=-2019-06-11_09:32:17
Operation	REGISTER
Source	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=
Recip	--
Amount	--
RecTime	2019-06-11 09:32:17
Signature	Bp8tBqgkTgxdyig[..]PmWAK8q9ngBMKWf

Figure 34.3 - Composition of a row of table BTRANSACTION describing the creation of an account for member *Mary*

The operation of account creation is coded in script _BC_Create_Account.sql.

34.9 Amount deposit

To simplify, we postulate that the amount a member deposits on his account comes from a source that we agree to ignore. So, this account is the recipient of an amount from an unknown source.

The function requires the identification of the recipient account and the amount to deposit. In this prototype, the recipient account is identified by the private name of the member.

The logic of the function is similar to that of creating an account, except that the value of the secret key is not calculated but is extracted from table ACCOUNT.

The main part of the deposit function is shown in Script 34.7. Figure 34.4 represents an example of the contents of a deposit transaction.

```
ask aid,amnt = Your name:[!select PrivName,AccountID
                           from   ACCOUNT order by PrivName]
               |Amount:;
extract  skey = select SecretKey from ACCOUNT
                 where  AccountID = '$aid$';
set      dat = $date$ $clock$;
set      tid = $aid$-$date$_$clock$;
set      op  = DEPOSIT;
function sig = LStr:encryptRSA
               {$tid$;$op$;$aid$;$amnt$;$dat$},{$skey$};
insert into BTRANSACTION
       (TransID,Operation,Recip,Amount,RecTime,Signature)
values ('$tid$','$op$','$aid$',$amnt$,'$dat$','$sig$');
```

Script 34.7 - Script executing a money deposit (excerpts)

TransKey	274
TransID	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=-2019-07-09_16:50:32
Operation	DEPOSIT
Source	--
Recip	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=
Amount	150
RecTime	2019-07-09 16:50:32
Signature	7h7Y025kG8tBqd[..]Pq9ngBMKWfmWAK8

Figure 34.4 - Composition of a row of table BTRANSACTION describing a DEPOSIT operation: *an amount of 150 is deposited on Mary's account*

The operation of amount deposit is coded in script _BC_Execute_Deposit.sql.

34.10 Money transfer

The user specifies the *source* account, the *recipient* account and the *amount* to be transferred between them. Accounts are designated by the user via their private names, which the ask function converts into account ids, then stored respectively in the variables **sou** (the *source* account) and **rec** (the *recipient* account) (Script 34.8).

The most interesting part of the function is the validation of the amount to be transferred, which cannot exceed the balance of the source account. This balance is obtained by a general procedure combining three subtotals representing the inputs and outputs of the source account:

– *Input*: the sum of the deposits on this account. This value is stored in variable **plusD**.

– *Input*: the sum of the transfers for which this account is the recipient is added. This value is stored in variable **plusT**.

– *Output*: the sum of the transfers for which this account is the source. This value is stored in variable **minus**.

We note the use of SQL function coalesce, that replaces *null* values with numeric *zero*. A *null* value appears when no rows have been found, which, in this particular case, must be interpreted as 0.

```
ask sou,rec,amnt =
          Your name:[!select PrivName,AccountID from ACCOUNT
                     order by PrivName]
         |Recipient:[!select PrivName,AccountID from ACCOUNT
                     order by PrivName]
         |Amount:;
extract skey = select SecretKey from ACCOUNT
               where  AccountID = '$sou$';
set     tid  = $sou$-$date$_$clock$;
set     op   = TRANSFER;
set     dat  = $date$ $clock$;
function sig = LStr:encryptRSA
              {$id$;$op$;$sou$;$rec$;$amnt$;$dat$},{$skey$};
extract plusD = select coalesce(sum(Amount),0)
                from   BTRANSACTION
                where  operation = 'DEPOSIT'
                and    Recip = '$sou$';
extract plusT = select coalesce(sum(Amount),0)
                from   BTRANSACTION
                where  operation = 'TRANSFER'
                and    Recip = '$sou$';
extract minus = select coalesce(sum(Amount),0)
                from   BTRANSACTION
                where  operation = 'TRANSFER'
                and    Source = '$sou$';
compute balance = $plusD$ + $plusT$ - $minus$;
if ($balance$ < $amnt$);
   showMessage Insufficient balance.@nOperation cancelled.;
   goto END;
endif;
insert into BTRANSACTION
  (TransID,Operation,Source,Recip,Amount,RecTime,Signature)
values ('$tid$','$op$','$acc$','$rec$',$amnt$,'$dat$','$sig$');
```

Script 34.8 - Script executing a transfer between two accounts (excerpts)

This computation could be performed more elegantly with a single SQL query. However, the detailed procedure of Script 34.8 is more intuitive and therefore has

been preferred. It differs from the Bitcoin protocol in that it processes the entire history of the source account since its creation and not only the last transactions the ouptput of which have not yet been consumed. This approach is particularly simple. Though it may seem ineffective, it will be appropriate insofar as:

– the source account history contains a reasonable number of transactions (a few thousands for example),

– indexes can been created on the Source and Recip columns respectively,

– a transfer operation execution time of a one or two seconds will certainly be considered acceptable by the user.

Figure 34.5 represents an example of the contents of a transfer transaction.

TransKey	322
TransID	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=-2019-07-22_13:18:45
Operation	TRANSFER
Source	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=
Recip	g20TQfu12uhOxztWBJ0tIUrNC5L0bhJKMLIUDKTt-Sk=
Amount	18.5
RecDate	2019-07-22 13:18:45
Signature	C75d4HLpm9STHj[..]h-blt2uCmMM_mfK

Figure 34.5 - Composition of a row of table BTRANSACTION describing a TRANSFER operation : *Mary transfers an amount of 18.5 to Luke's account*

The operation of amount transfer is coded in script _BC_Execute_Transfer.sql.

34.11 Validating a transaction

This function checks that the accounts involved in the transaction are recorded in the DIRECTORY table, that the amount of the transfer transaction is legitimate (less than or equal to the balance available on the transaction creation date) and that the values of the components are identical to the content of the signature, decrypted via the public key.

The operation of transaction validation is coded in script _BC_Validate_Transaction.sql.

34.12 Creating and mining a block

This function creates a block containing all the transactions still pending and calculates the nonce that produces a hash value starting with **k** binary zeroes. This hash thus obtained is the block identifier (column Bhash). The identifier of the last block created so far is also added to the new block (column Bprev), therefore increasing the blockchain by one unit.

The procedure of block creation and mining is translated in Script 34.10. It comprises three steps:

1. Building the data of the block

The body of the block is built by concatening the list of pending transactions (those not yet included in a block), the recording time and the hash of the last block of the chain.[4]

The hash of the last block, that will become the previous block, is extracted from the row of the block with the highest recording time[5]:

```
extract previous = select BChash from BLOCK
                   order by RecTime desc;
```

The list of pending transactions is computed in variable transList by a single SQL query that assembles the contents of each transaction according to its type (column Operation) then concatenates them though group_concat aggregate function. For instance, a REGISTER transaction is assembled as follows (as an SQLfast list[6]):

```
TransID||';REGISTER;'||Source||';'||RecTime||';'
       ||Signature
```

In addition, this query counts the transactions to include in the new block (variable blockSize). The body of the block can then be computed:

```
set body = $transList$#$recTime$#$previous$;
```

It is interesting to note that the list of transactions itself is an SQLfast list the separator of which is the **#** character. So, this *list of lists* can be easily disassembled to retrieve the properties of each transaction of the list thanks to the SQLfast list functions. This means that the value of variable body is a valid serialization of the contents of the block that can be broadcasted to the other members of the blockchain through *email*, *p2p* or *ftp* protocols. This distribution aspect is not taken in charge by this prototype.

2. Mining the block

To compute a valid hash and its nonce, we use **miningHash**, a variant of SQLfast function **hash**. This function uses four arguments:

4. Which, actually, is the *building* time, before mining.
5. The `extract` statement processes the first row of the result set. Clause `limit 1` is therefore useless.
6. An SQLfast list is a character string formed by a series of elements separated by a special character (here a semi-colon).

– The character string to hash (in variable body).
– The constraint on the hash, as a **k**-length string of zero characters (in variable prefix); **k** is the *difficulty* of the mining, set to 20 in the standard SQLfast distribution.
– The maximum number of trials (in variable trials); set to 5,000,000 in the standard distribution.
– The format of the resulting hash; 1 for hexadecimal and 2 for base64.

It returns three results:

– hash, the value of a valid hash, if one has been found.
– nonce, the corresponding nonce.
– nHash, the number of hash computations that have been needed to get this valid hash.

If no valid hash has been found, hash = nonce = ''.

```
set k = 20;
compute prefix = repeat('0',$k$);
set trials = 5000000;
function hash,nonce,nHash
         = LStr:miningHash $body$,$prefix$,$trials$,2;
```

The core of this function is shown in Script 34.9. Variable Bdata denotes the string to hash (as a *byte string*, not a *unicode string*) augmented with the current candidate nonce (str(N)). Variable Bhash denotes the sha256 hash of this value, as a binary string. Chash is the prefix of this hash, converted in ASCII characters (one character per bit). Bhash is a valid hash when Chash = prefix.

This procedure certainly is not the fasted we can write, but it is quite sufficient considering the didactic objective of this prototype. A more realistic version would be written in C with true bit manipulation primitives and executed by a cluster of (let us keep it modest) 4,096 processors!

3. Adding the block in the blockchain

A new row is inserted in table BLOCK, comprising the valid hash, the number of transactions, the nonce, the recording time and the hash of the previous block, if any. The rows of the transactions of the block are updated to make them reference the new block.

Figure 34.6 below shows the contents of the rows of the BLOCK table representing blocks with BlockKey 113 and 114 of the chain[7]. We will see how to visualize the transactions of each block in the next section.

7. Prefix 'AAAK' translates '000000 000000 000000 001010' in base64, which complies with *difficulty* k = 20.

```
def miningHash(data,prefix,hashNbr,form):
    lenPrefix = len(prefix)
    nonce = ''
    for N in range(trials):
        Bdata = data.encode('latin-1') + str(N)
        Bhash = hashlib.sha256(Bdata)
        Chash = ''.join('{0:08b}'.format(ord(x), 'b')\
                for x in Bhash.digest()\
                        [:lenPrefix//8+1])[:lenPrefix]
        if Chash == prefix:
            nonce = str(N)
            break
    if form == 2:
        hash = base64.b64encode(Bhash.digest(),'_-')
    return hash,nonce,N
```

Script 34.9 - Main section of Python function LStr:**miningHash** used to mine a block (excerpts)

```
Block n° 114
    Block hash:     AAAKqRd-Zq2PtROoXNLidrG_lYAVT5XnfkKsToEGy1w=
    Nb transact:    5
    Nonce:          1526497
    Creation date:  2019-09-02 12:25:56
    Status:         0
    Previous hash:  AAAKayF8dzDLswTa5rb_V9nP_Gp536vaZDPuInYCIcM=

Block n° 113
    Block hash:     AAAKayF8dzDLswTa5rb_V9nP_Gp536vaZDPuInYCIcM=
    Nb transact:    5
    Nonce:          967544
    Creation date:  2019-09-02 12:15:25
    Status:         0
    Previous hash:  AAAKQ7I-HfRKPSCbFK4OKoJ3vkUdLrC-UXC2GVAtECU=
```

Figure 34.6 - Contents of two successive blocks of 5 transactions

The operation of block creation and mining is coded in script _BC_Create_Block.sql.

34.13 TExamining the blockchain data

Blockchain management systems include an application devoted to the examination of a blockchain and commonly called *Blockchain Explorer*. In this section, we present three exploration functions that can help experimenters understand the result of the blockchain manipulation operations.

```
extract previous = select BChash from BLOCK
                   order by RecTime desc;
extract transList, blockSize
        = select group_concat2(
                  case Operation
                    when 'REGISTER'
                    then TransID||';REGISTER;'||Source
                         ||';'||RecTime||';'||Signature
                    when 'DEPOSIT'  then ...
                    when 'TRANSFER' then ...
                    else 'Transaction type unknown'
                  end,0,TransKey,1,'#'), count(*)
          from   BTRANSACTION
          where  BChash is null;

set recTime = $date$ $clock$;
set data = $transList$#$recTime$#$previous$;

set k = 20;
compute prefix = repeat('0',$k$);
set trials = 5000000;

function hash,nonce,nHash
     = LStr:miningHash $data$,$prefix$,$trials$,2;

insert into BLOCK(BChash,Transact,Nonce,RecTime,BCprev)
       values('$hash$',$blockSize$,'$nonce$','$recTime$',
              case when '$previous$' = '' then null
                   else '$previous$'
              end);

update BTRANSACTION
set    BChash = '$hash$'
where  BChash is null;
```

Script 34.10 - Script creating and mining a block (excerpts)

To illustrate the result produced by these fonctions we suppose that the following operations have been executed, in this order:

1. Mary creates an account
2. Mary deposits an amount of 100 into her account
3. Luke creates an account
4. Mary transfers an amount of 60 to Lucke's account
5. Luke deposits an amount of 20 into his account
6. Ann creates an account
7. Luke transfers an amount of 35 to Ann's account
8. Luke transfers an amount of 15 to Mary's account
9. Mary transfers an amount of 18 to Ann's account
10. Ann deposits an amount of 30 into her account

11. Ann transfers an amount of 40 to Luke's account
12. Ann transfers an amount of 12 to Mary's account

In addition, a block is created as soon as 5 transactions have been recorded. The final chain will therefore consist of two blocks of 5 transactions and 2 pending transactions.

34.13.1Examining an account

The user selects an account to display the successive transactions of which it is the source or the recipient, as well as its current balance. For reasons of readability, in this function as well as in the other two, accounts are designated by the private name of their owner. The report below is related to Mary's account.

Transactions of your account

Source	Operation	Amount	Recip	TimeCreated	InBlock
Mary	REGISTER	--	--	2018-12-30 19:25:47	yes
--	DEPOSIT	100	Mary	2018-12-30 19:26:09	yes
Mary	TRANSFER	60	Luke	2018-12-30 19:26:48	yes
Luke	TRANSFER	15	Mary	2018-12-30 19:31:44	yes
Mary	TRANSFER	18	Ann	2018-12-30 19:31:59	yes
Ann	TRANSFER	12	Mary	2018-12-30 19:36:34	--

Your account balance

Account	Deposit	Received	Transfered	Balance
Mary	100	27	-78	49

This function is coded in script _BC_Examine_Account.sql.

34.13.2Examining the transactions

The function displays the chronological list of transactions and the balance of each account.

Transactions

Source	Operation	Amount	Recipient	TimeCreated	Block
Mary	REGISTER	--	--	2018-12-30 19:25:47	yes
--	DEPOSIT	100	Mary	2018-12-30 19:26:09	yes
Luke	REGISTER	--	--	2018-12-30 19:26:25	yes
Mary	TRANSFER	60	Luke	2018-12-30 19:26:48	yes
--	DEPOSIT	20	Luke	2018-12-30 19:27:02	yes
Ann	REGISTER	--	--	2018-12-30 19:31:00	yes
Luke	TRANSFER	35	Ann	2018-12-30 19:31:26	yes
Luke	TRANSFER	15	Mary	2018-12-30 19:31:44	yes
Mary	TRANSFER	18	Ann	2018-12-30 19:31:59	yes
--	DEPOSIT	30	Ann	2018-12-30 19:32:12	yes
Ann	TRANSFER	40	Luke	2018-12-30 19:36:19	--

```
| Ann    | TRANSFER  | 12     | Mary      | 2018-12-30 19:36:34 | --    |
+--------+-----------+--------+-----------+---------------------+-------+
```

Account balance

Account	Deposit	Received	Transfered	Balance
Ann	30	53	-52	31
Luke	20	100	-50	70
Mary	100	27	-78	49

This function is coded in script _BC_Examine_Transactions.sql.

34.13.3Examining the blockchain

The function displays the content of each block in the chain, its properties and those of each of its transactions.

Block n° 2

Block hash: AAALlzLVgFBSa_E5Wzqf1RQ6VlK8M1mVkrlvHha26r4=
Nbr transact: 5
Nonce: 334045
Record Time: 2018-12-30 19:35:05
Status: 0
Hash previous: AAAD8ewt9-jHhBJts048MamR-vbeCts4BXLCpZh8QsY=

Source	Operation	Amount	Recipient	TimeCreated
Ann	REGISTER	--	--	2018-12-30 19:31:00
Luke	TRANSFER	35	Ann	2018-12-30 19:31:26
Luke	TRANSFER	15	Mary	2018-12-30 19:31:44
Mary	TRANSFER	18	Ann	2018-12-30 19:31:59
--	DEPOSIT	30	Ann	2018-12-30 19:32:12

Block n° 1

Block hash: AAAD8ewt9-jHhBJts048MamR-vbeCts4BXLCpZh8QsY=
Nbr transact: 5
Nonce: 1815939
Record Time: 2018-12-30 19:27:10
Status: 0
Hash previous: --

Source	Operation	Amount	Recipient	TimeCreated
Mary	REGISTER	--	--	2018-12-30 19:25:47
--	DEPOSIT	100	Mary	2018-12-30 19:26:09
Luke	REGISTER	--	--	2018-12-30 19:26:25
Mary	TRANSFER	60	Luke	2018-12-30 19:26:48
--	DEPOSIT	20	Luke	2018-12-30 19:27:02

Pending transactions (not yet in block)

Source	Operation	Amount	Recipient	TimeCreated
Ann	TRANSFER	40	Luke	2018-12-30 19:36:19
Ann	TRANSFER	12	Mary	2018-12-30 19:36:34

This function is coded in script _BC_Examine_Blockchain.sql.

34.14 Wrapping it all: the BLOCKCHAIN toolbox

The functions of the toolbox are independent, but, for convenience, they are available via the control screen in Figure 34.7, displayed by script **BLOCKCHAIN.sql**. The execution of the three operations generating a transaction can be accompanied by the display of the contents of the ACCOUNT, DIRECTORY and BTRANSACTION tables. To do this, check the box entitled "Display table contents after each transaction".

The table contents can also be viewed via the "Database > Show DB Data" menu function in the main SQLfast window.

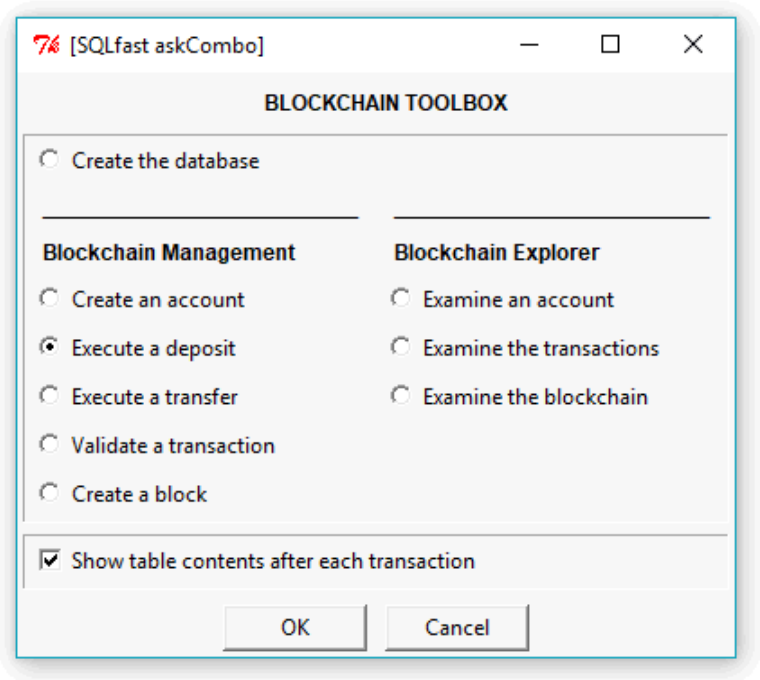


Figure 34.7 - Control panel of the toolbox

The scripts of the prototype Blockchain toolbox are available in the SQLfast/Scripts/Case-Studies/Case_BlockChain directory.