View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repository of the University of Namur

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Visualizing SQL Execution Traces for Program Comprehension

Meurice, Loup

Award date:
2013

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Jun. 2020

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2012–2013

Visualizing SQL Execution Traces for Program Comprehension

Loup Meurice

Maître de stage : Anthony Cleve

Promoteur : ______________________ (Signature pour approbation du dépôt - REE art. 40)

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Software maintenance and evolution are important and sensitive activities becoming ubiquitous nowadays. They constitute complex processes due to different factors (lack of money/time/documentation or merely a bad programming work) but those processes are indispensable to keep software systems adapted to ever-changing business needs and technological platforms. Program comprehension is a typical initial phase of software maintenance and evolution. Indeed, understanding existing software systems is a preliminary step that is required before making any changes to them. Furthermore, when understanding data-intensive programs, the communication between those programs and their database constitutes an important aspect that is largely unexplored by the program comprehension research community.

This Master's thesis aims to address this issue by proposing a novel tool-supported approach that consists in analysing the database manipulation behaviour of data-intensive programs. The proposed approach combines the use of dynamic program analysis and software visualization techniques. Thanks to our approach, we bring a new way to automatically support data-intensive program comprehension, with the goal to reduce the costs of this process.

Keywords: dynamic analysis, visualization, SQL, program comprehension.

Contents

1 Introduction **1**
1.1 Software evolution 1
1.2 Program comprehension 2
1.3 Data-intensive systems 4
1.4 SQL statement analysis 4
1.5 Thesis purposes 8
1.6 Overview 8

2 State of the Art **9**
2.1 A Systematic Survey of Program Comprehension through Dynamic Analysis 9
2.1.1 Presentation 9
2.1.2 The proposed tool 13
2.2 Studying of particular cases 13
2.2.1 Dynamic Program Analysis for Database Reverse Engineering 13
2.2.2 WAFA: Fine-grained Dynamic Analysis of Web Applications 14
2.2.3 An approach for mining services in database–oriented applications 14
2.2.4 Improving Dynamic Data Analysis with Aspect-Oriented Programming 15
2.2.5 Visualizing Dynamic Software System Information through High-level Models 15
2.2.6 Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information 16
2.2.7 CodeCity 16
2.2.8 EXTRAVIS 17

3 Database Engineering **19**
3.1 Database schemas 21

3.1.1 Conceptual schema . . . 21
3.1.2 Logical schema . . . 22
3.1.3 Physical schema . . . 24
3.1.4 Database code . . . 24
3.2 ER model subset considered . . . 26
3.3 Schema mapping . . . 30
3.4 SQL . . . 32
3.4.1 SQL-DDL . . . 33
3.4.2 SQL-DML . . . 34
3.5 Database reverse engineering . . . 35

4 DAViS **37**
4.1 Context . . . 37
4.2 Offline method . . . 38
4.3 Online method . . . 41
4.4 Two levels of abstraction . . . 43
4.4.1 Logical approach . . . 44
4.4.2 Conceptual approach . . . 45
4.5 The two visualizations . . . 47
4.5.1 The query visualization . . . 47
4.5.2 The global visualization . . . 48
4.6 The two-phase principle . . . 48

5 Visualization **51**
5.1 Graph-based representation . . . 51
5.2 Global visualization . . . 54
5.2.1 Thickness pattern . . . 55
5.2.2 Advantages of the global visualization . . . 55
5.3 Query visualization . . . 56
5.3.1 Logical approach . . . 57
5.3.2 Conceptual approach . . . 59

6 Implementation **64**
6.1 DB-Main . . . 64
6.2 JIDBM . . . 66
6.3 JDBC . . . 67
6.4 AspectJ . . . 68
6.5 SQL Parser . . . 69
6.5.1 SQL hypotheses . . . 69
6.5.2 Parsed query structures . . . 72
6.6 JUNG . . . 76

7 Evaluation **77**
7.1 Case studies 77
7.1.1 Webcampus 77
7.1.2 Rever 79
7.2 Forces and limitations 80
7.2.1 Forces 80
7.2.2 Limitations 81
7.3 Comparison with the survey 81
7.4 Usage possibilities 83

8 Conclusion **85**
8.1 Summary and contributions 85
8.2 Future directions 86

Appendices **89**

A Additional functionalities **90**

B Offline and online methods **93**

List of Figures

1.1 The software life cycle according to Boehm. 2
1.2 Simple database with two tables 5
1.3 Example of static SQL : extracting address and date of customers placing a particular order. 5
1.4 Common example of dynamic SQL: selecting the name of a particular customer. 5
1.5 Program instrumentation example - bold line represents the code modification 6
1.6 Aspect-based tracing example 7
1.7 API Overloading example - Statement class overloading 7

2.1 Distribution of the attributes in each facet across the summarized articles. The X-axis represents all the attributes classified by facet and the Y-axis represents the number of articles affected to each attribute. This graph is extracted from the report of Delft University. 12
2.2 Logging Aspect 15

3.1 Relational database building - Process 20
3.2 Customer-Order : conceptual schema 22
3.3 Customer-Order : logical schema 24
3.4 DDL Code - Translation of logical schema 25
3.5 Metaschema of ER model 26
3.6 Example of compound attribute - ADDRESS 27
3.7 Example of IS-A relationship - CUSTOMER 27
3.8 Example of non-binary relationship type - A customer orders some products for a particular company. 28
3.9 Compound attribute split into simple attributes - ADDRESS . 28
3.10 Transformation of a supertype into relationship types - CUSTOMER 29

3.11 Transformation of a complex relationship type into binary relationship types - Customer ordering some products for a particular company. 29
3.12 Metaschema of the ER model subset considered in this thesis. 30
3.13 Correspondence between conceptual and logical schema expressed by the relational model - Overview 31
3.14 DDL example - Creating table CUSTOMER 33
3.15 DDL example - Deleting column 33
3.16 DML example - Inserting new product 34
3.17 DML example - Selecting CUSTOMER 34
3.18 DML example - Updating product's stock 34
3.19 DML example - Deleting product 34
3.20 Database reverse engineering process 36

4.1 Program-Database context 38
4.2 Offline method - First phase 39
4.3 Offline method - Second phase 40
4.4 Offline method - Global view 41
4.5 Online method - Overview 42
4.6 Summary of DAViS features. 43

5.1 Example of logical visualization based on graph representation - the arrow represents a foreign key while the dotted line represents a join relationship. A continuous line represents a parental (containment) link between a table and its column(s). 53
5.2 Example of conceptual visualization based on graph representation. 53
5.3 Context applied on a time line. 54
5.4 Thickness pattern 55
5.5 $State_i$ corresponds to the graph state at time α_i. 56
5.6 SQL query selecting the name and price of a particular product. 57
5.7 Query visualization in logical approach - the two phases 58
5.8 Example of SQL query. 58
5.9 Query visualization in logical approach - the two phases 59
5.10 Application domain - entities and relationships 60
5.11 Application domain - clustering of entities of the same nature 60
5.12 Example of SQL query - selecting customers having ordered a particular product. 63

6.1 Example of our conceptual schema written with DB-Main tool 65

6.2 Example of database querying with JDBC - selecting the address of a particular customer 67
6.3 Example of database updating with JDBC - customer deleting 68
6.4 Online method - AspectJ code 69
6.5 Example of managed SQL syntax 71
6.6 Sub-request (IN clause) - Example of managed sub-request: line 3 and line 5 are both managed sub-requests 71
6.7 Sub-request (IN clause) - Example of unmanaged sub-request because element (NAME,STOCK) is not mono-component . . 71
6.8 Sub-request ($EXISTS$ clause) - Example 72
6.9 Parsing process 73
6.10 Java structures of a parsed query - modelling 75

7.1 Logical schema of Webcampus's database 78
7.2 SQL query - WHEN/THEN/ELSE 79

8.1 Example of undocumented code 87
8.2 Example of redocumented code 88

A.1 DAViS interface 92

B.1 Offline method - Components 93
B.2 Online method - Components 94

Chapter 1

Introduction

1.1 Software evolution

Nowadays, we live in a society and an economic world that constantly and rapidly change, which in turn forces the companies to evolve in order to adapt to their ever-changing environment. Moreover, in order to satisfy the market requirements, companies are often likely to evolve and maintain their software systems. Indeed, software systems are omnipresent and they often constitute the heart of business-critical activities. Software maintenance, defined as "*the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.*" [3], is a complex process that is part of *software engineering*. Software engineering is the application of a systematic and disciplined approach allowing the development of a software product. Such an approach is called the software development process, also known as the *software development life-cycle*(SDLC) [5] [18]. This process describes the tasks and activities to follow during software development. We can distinguish three main software development activities (Figure 1.1):

1. System definition: the development process begins with an analysis aiming to define the requirements of the system.
2. Implementation: this activity regroups the system *implementation* (code writing) and *testing* processes.
3. Maintenance: this activity represents the maintenance and evolution of the system over time.

One identifies four categories of software maintenance [1]:

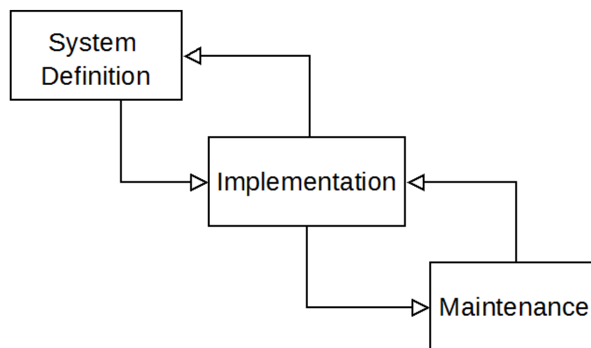

Figure 1.1: The software life cycle according to Boehm.

- *Corrective* maintenance: identifying and fixing errors present in a software product. It consists in correcting discovered problems.

- *Adaptive* maintenance: modifying the software in order to satisfy a changing environment.

- *Perfective* maintenance: modifying the software product to satisfy new user requirements.

- *Preventive* maintenance: modifying the software product to increase maintainability and reliability. This category of maintenance aims to prevent eventual problems in the future.

Software systems maintenance and evolution is a complex, time-consuming, expensive and risky process. One considers that software maintenance represents on average about 60 percent of the entire software development costs and it is therefore the most costly phase of the software life cycle [13].

Furthermore, the community is more and more inclined to use the term *software evolution* to refer to software maintenance. By simplification, we will consider in the remaining of this thesis that software evolution and maintenance denote the same process.

1.2 Program comprehension

One of the most important processes in software evolution is program understanding. Indeed, software evolution and maintenance need a preliminary phase of program comprehension. An in-depth understanding is necessary

before maintenance, but this process remains a sensitive and time(money)-consuming process. Indeed, one considers that more than 50 percent of the total costs of software maintenance are spent on (program) comprehension. This process mainly depends on existing program documentation but unfortunately, this documentation is often partial, outdated or simply missing. Furthermore, a lack of programming competences and important turnover of programmers make program understanding more difficult.
Therefore, re-documentation and program comprehension are part of software evolution and maintenance.

Static analysis A natural way to proceed in the program comprehension process is by means of *static program analysis* techniques. This kind of analysis consists in analyzing the *source code* of the program in order to extract static information and derive program properties of interest. Then, by studying the extracted properties, we can obtain a better understanding of the program. It allows to study the structure, the dependencies and the behaviour of the program but without executing it.

Dynamic analysis However, static analysis techniques can be limited because an important part of information in a program may be generated at runtime, and thus can be impossible to extract statically by means of source code inspection. The solution to this problem is to use *dynamic program analysis.* This second category of program analysis techniques consists in analyzing the properties of the *running* program, i.e., at execution time.
Dynamic analysis allows to mine an exact picture of the program during its execution. On the one hand it can be more precise than static program analysis, but on the other hand, it is obviously restricted to some execution paths of the program.
Dynamic program analysis can be used to extract dynamic information such as:

- program performance figures
- program memory usage
- runtime errors
- execution workflows

In summary, dynamic program analysis typically aims to analyse information built at runtime while static analysis handles information extracted from source code. We can notice that both techniques are complementary for achieving a complete program understanding process.

Visualization The main purpose of program comprehension is to generate usable information allowing a better vision of the program structure and behaviour. Different kinds of information can be extracted from this process. Depending on the kind and the amount of information, it could be interesting to synthetically present the results obtained by means of a *data visualization*. The main goal of data visualization is to communicate information clearly and effectively through graphical means.

1.3 Data-intensive systems

With the explosion of the Internet and the increasing number of websites, data became the key-concept around which our society is founded and, consequently software systems are more and more *data-intensive*. In data-sensitive systems, the database often occupies a central place and, therefore, the communication between the programs and the database is an important aspect of the system behaviour.

Therefore, the comprehension of data handling and exchange allows, as a second stage, to understand the program goals and its actions on the database. Thus, we can consider that understanding the data-manipulation behaviour of a program is an important part of data-intensive program comprehension and more generally, when *reverse engineering* data-intensive systems.

Reverse engineering [6] is the initial phase of the maintenance process that helps you understand the system before you can make appropriate changes.

1.4 SQL statement analysis

The data exchange between programs and the database is generally performed through SQL[1] queries sent by the program. Therefore, by understanding this query exchange, we could be able to better comprehend what the program is doing. SQL is a high-level language that allows programmers to describe in a declarative way the properties of the data they instruct the DBMS[2] to provide them with [7].

The analysis of SQL statements in application programs is a technique allowing to understand interactions between programs and the database. There exist two types of SQL statements : static and dynamic SQL.

[1] SQL: Structured Query Language

[2] Database Management System

In order to illustrate the difference between static and dynamic SQL, let us take an example of database shown in Figure 1.2.

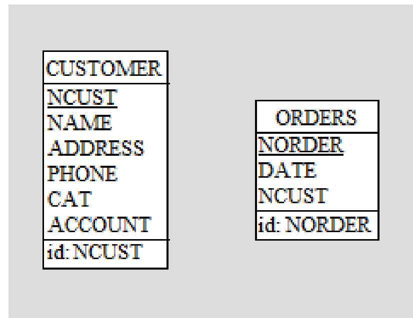

Figure 1.2: Simple database with two tables

Static SQL is when the SQL statements are "*hard-coded*" in the source code of the program. An example is shown at Figure 1.3.

```
select ADDRESS, DATE into :ADDR, :DATE
from CUSTOMER C, ORDERS O
where C.NCUST = O.NCUST and NORDER = :IDORD
```

Figure 1.3: Example of static SQL : extracting address and date of customers placing a particular order.

In this query, there are one input variable and two output variables. The input variable *IDORD* contains the number of an order while the output variables *ADDR* and *DATE* will contain respectively the customer's address and the order's date.

Dynamic SQL is when the SQL statements are built at *runtime*. An example is shown at Figure 1.4.

```
QUERY = "select NAME from CUSTOMER where NCUST = :v1";
exec SQL prepare Q from :QUERY;
exec SQL execute Q into :NAME using :CNUM;
```

Figure 1.4: Common example of dynamic SQL: selecting the name of a particular customer.

This query takes a customer's number as input (variable *v1*) and extracts the customer's name in the variable *CNUM*.

Typically, the programs build each query as a character string, then ask the DBMS to prepare the query and finally execute it. [7]

Static SQL statements can be simply extracted by parsing the source code but more and more systems use dynamic SQL. Thus, code parsing is insufficient and other techniques to extract SQL queries at runtime are required.

Cleve et al. [7] listed different techniques allowing to extract SQL statements built during the program execution :

1. Program instrumentation : modification of the source code to catch each SQL statement. This technique requires program modification and recompilation. Figure 1.5 shows an example of program instrumentation.

```
(1)  Connection conn = DriverManager.getConnection(URL, Login, Password);
(2)  String ID = "D123";
(3)  String query = "select ADDRESS from CUSTOMER where NCUST = ?";
(4)  PreparedStatement state = conn.prepareStatement(query);
(5)  state.setString(1, ID);
(6)  ResultSet result = state.executeQuery();
(7)  Logger.getQueryResult(result);
(8)  result.next();
```

Figure 1.5: Program instrumentation example - bold line represents the code modification

2. Aspect-based tracing : an aspect can be considered as a kind of trigger. This trigger is launched when a SQL query is executed. This technique only requires code recompilation. Figure 1.6 represents an example of aspect allowing to log the executed SQL queries.

```
(1)  pointcut queryExecution(String query):
(2)  call(* java.sql.Statement.executeQuery(String)) && args(query);
(3)  before(String query): queryExecution(query){
(4)     Logger.getQuery(query); //logging the current SQL query
(5)   }
```

Figure 1.6: Aspect-based tracing example

3. API[3] Overloading : Technique consisting in overloading the API executing SQL queries. An example of API overloading is shown at Figure 1.7.

```
(1)  public class Statement{
(2)     java.sql.Statement statement;
(3)      public ResultSet executeQuery(String query) {
(4)          Logger.getQuery(query); //logging the current SQL query
(5)          return statement.executeQuery(query);
(6)   }
(7)  }
```

Figure 1.7: API Overloading example - Statement class overloading

4. API substitution : If the API executing SQL queries is open-source, we can modify the API source code in order to catch queries.

5. DBMS logs : when a SQL query is sent to the database, the former is stored into DBMS logs. One can then simply extract all queries from the logs.

6. Tracing stored procedures : A SQL procedure is a sequence of SQL queries stored into the DBMS. We can modify the program code by replacing each SQL query with an equivalent SQL procedure enriched with tracing facilities.

Each technique has its own merits and drawbacks. Cleve et al. established a list (see Table 1.1). The retained criteria are *the availability of the query results*, the need for code *modification* and for *recompilation*.

[3] API : Application Programming Interface

Techniques	Results availability	Code modification	Code recompilation
Instrumentation	yes	yes	yes
Aspect	yes	no	yes
API Overloading	yes	yes	yes
API Substitution	yes	no	yes
DBMS logs	no	no	no
Stored procedures	yes	yes	yes

Table 1.1: Comparison of SQL trace capturing techniques

1.5 Thesis purposes

We can observe that program comprehension is not a trivial process but the interactions between the program and the database can be considered as a good approximation of what a data-intensive program is doing. These interactions are materialized by SQL queries sent by the program. Nevertheless, more and more programs utilize dynamic SQL statements built only during program execution, i.e., at runtime.
This thesis proposes a new approach allowing to analyse SQL queries captured at runtime in order to make program comprehension easier. The analysis will be performed by visualizing dynamically their impact on the database.

More generally, the thesis purpose is to propose a new tool-based approach allowing to automatically support data-intensive program comprehension in order, as a second stage, to ease software system evolution.

1.6 Overview

Chapter 2 presents the state of the art related to program comprehension and visualization. This chapter attempts to give an overview of the activities from the research community pertaining to this area. Chapter 3 tries to summarize the theoretical concepts of database engineering we reuse in our approach. Chapter 4 and 5 present in detail our tool-supported approach. Chapter 6 presents the technological choices made for the tool's implementation. In Chapter 7, we try to assess the forces and limitations of our tool and we list a few usage possibilities. Finally, Chapter 8 concludes this thesis by discussing the future perspectives.

Chapter 2

State of the Art

Program comprehension is a crucial process in software maintenance/evolution that is necessary in order to sufficiently understand the program before its modification. This activity has received the attention from the research community, particularly over the last decade. Program understanding requires the analysis of such artefacts as source code and documentation but it is not sufficient to get a complete view of the program/system. Indeed, analyzing the program behaviour at runtime can provide additional information pertaining to what the program is doing in specific execution scenarios. As explained above, this kind of analysis is called dynamic program analysis.

2.1 A Systematic Survey of Program Comprehension through Dynamic Analysis

2.1.1 Presentation

Dynamic program analysis is a largery explored research topic in the software reengineering community. This is why a survey [9] was established in order to summarize the most relevant existing literature about dynamic analysis in the context of program comprehension. This survey considered research results that have been published between July 1999 and June 2008. It aimed to extract and select relevant works about this area by means of explicit selection criteria.
The authors selected 4,795 articles published at all relevant venues (journals/conferences). Then they applied their selection criteria on that first selection. This resulted in a final list that comprises 172 articles published in 14 different venues.
After this selection phase, the authors have defined a list of facets in order

to classify all the selected articles. There are four retained facets :

- The **activity** describes what is being performed or contributed
- The **target** reflects the type of programming language(s) or platform(s) to which the approach is shown to be applicable
- The **method** describes the dynamic analysis methods that are used in conducting the activity
- The **evaluation** outlines the manner(s) in which the approach is validated

For each facet, a set of attributes are defined (see Table 2.1).

Facet	**Attribute**	**Description**
Activity	survey	a survey or comparative evaluation of existing approaches that fulfil a common goal.
	design/arch.	the recovery of high-level designs or architectures.
	views	the reconstruction of specific views, e.g., UML sequence diagrams.
	features	the analysis of features, concepts, or concerns, or relating these to source code.
	trace analysis	the understanding or compaction of execution traces.
	behaviour	the analysis of a system's behaviour or communications, e.g., protocol or state machine recovery.
	general	gaining a general, non-specific knowledge of a program.
Target	legacy	legacy software, if classified as such by the author(s).
	procedural	programs written in procedural languages.
	oo	programs written in object-oriented languages, with such features as late binding and polymorphism.
	threads	multithreaded systems.
	web	web applications.
	distributed	distributed systems.
Method	vis. (std.)	standard, widely used visualization techniques, e.g., graphs or UML.
	vis. (adv.)	advanced visualization techniques, e.g. polymetric views or information murals.
	slicing	dynamic slicing techniques.
	filtering	filtering techniques or selective tracing, e.g., utility filtering.
	metrics	the use of metrics.
	static	information obtained through static analysis, e.g., from source code or documentation.
	patt. det.	algorithms for the detection of design patterns or recurrent patterns.
	compr./summ.	compression, summarization, and clustering techniques.
	heuristics	the use of heuristics, e.g., probabilistic ranking or sampling.
	fca	formal concept analysis.
	querying	querying techniques.
	online	online analysis, as opposed to post mortem (trace) analysis.
	mult. traces	the analysis or comparison of multiple traces.
Evaluation	preliminary	evaluations of a preliminary nature, e.g., toy examples.
	regular	evaluations on medium-/large-scale open source systems (10K+LOC) or traces (100K+ events).
	industrial	evaluations on industrial systems.
	comparison	comparisons of the authors' approach with existing solutions.
	human subj.	the involvement of human subjects, i.e., controlled experiments & questionnaires.
	quantitative	assessments of quantitative aspects, e.g., speed, recall, or trace reduction rate.
	unknown/none	no evaluation, or evaluations on systems of unspecified size or complexity.

Table 2.1: Facets and attributes

For the same facet, several attributes can be combined. By applying this analysis grid to each article, the authors of the survey have obtained the results presented in Figure 2.1.

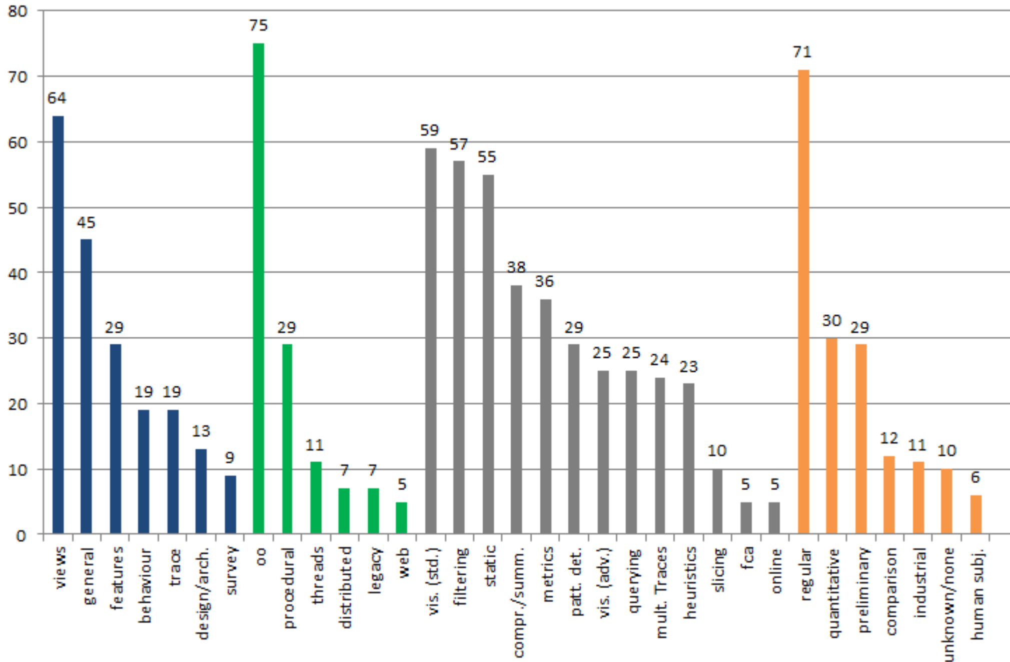

Figure 2.1: Distribution of the attributes in each facet across the summarized articles. The X-axis represents all the attributes classified by facet and the Y-axis represents the number of articles affected to each attribute. This graph is extracted from the report of Delft University.

Activity : We can notice that for the first facet, activity, the view attribute is the most frequent. It is not surprising because program comprehension handles a large amount of information and thus, the reconstruction of specific views seems to be a natural activity.
On the other hand, survey is the least frequent activity (this has motivated the authors to do the survey).

Target : It is quite surprising to observe that web applications is the least frequent target. Indeed, web applications are ubiquitous nowadays. Furthermore, we can also notice that legacy systems are rarely chosen as target while these systems often require a (costly) reverse engineering process in order for

the evolution needs to be satisfied.
In contrast, object-oriented systems are becoming more popular target systems in the domain of dynamic program analysis.

Method : Visualization is the most popular method. Indeed, systems deal with a large amount of information and visualization techniques seem to be natural and intuitive to easily summarize and thus, quickly understand the behaviour of a program at runtime.
Nevertheless, we can also notice that online dynamic analysis is almost never used in the area.

2.1.2 The proposed tool

Through this thesis, a new tool will be introduced to the reader. Indeed, as previously explained, communication between database and the programs is an important aspect when trying to understand data-intensive programs. By analysing the results of this large-scale survey (Figure 2.1), we can notice that there is no article dealing with database/program interactions analysis. Moreover, we can also derive from this survey that the literature about database-program communication analysis is very poor: it appears as a largely unexplored area. Thus, this observation is one reason that motivated us to implement a new dynamic program analysis tool based on SQL queries analysis.

2.2 Studying of particular cases

Before the detailed description of the tool, it is quite important to study some interesting articles (from the survey but also from other venues) about program dynamic analysis. The selected articles have been retained because they target some same goals than the proposed tool or because they use some techniques that have inspired the tool.
For each articles, we have made an in-depth description.

2.2.1 Dynamic Program Analysis for Database Reverse Engineering

This article [8] tackles the issue of database and program evolution. Indeed, such processes should be supported by the database documentation. Nevertheless, this documentation is often inaccurate or even missing. Therefore, database redocumentation is sometimes needed before program and

database evolution. This process typically consists in recovering some implicit database constructs and constraints. This activity requires the analysis of the database code but also other artefacts like the source code of the programs accessing to the target database.
In this context, the authors introduce automated dynamic program analysis techniques facilitating the database redocumentation and more precisely, the recovery of implicit database constructs and constraints. Those techniques are based on the analysis of the dependencies in SQL execution traces.

Dynamic analysis of SQL execution traces and the elicitation of some implicit database constraints (Section 5.2.2) are two concerns tackled by the thesis.

2.2.2 WAFA: Fine-grained Dynamic Analysis of Web Applications

WAFA [4] is an approach aiming to analyse and detect most web application security vulnerabilities (e.g. SQL injections, broken access control, ...) by studying database interactions.
This approach uses both static and dynamic program analysis. In particular, WAFA captures the original SQL statements source but also the SQL execution instances in order to compare both. It obviously needs static and dynamic analysis. Furthermore, WAFA captures other dynamic information like cookies, HTTP variables.

We can easily relate WAFA to the tool proposed in this thesis through the SQL statements capture. Indeed, the tool mainly aims to extract SQL queries created at runtime to analyse and visualize them.

2.2.3 An approach for mining services in database–oriented applications

It proposes an approach using dynamic program analysis that aims to extract SQL queries at runtime and cluster them in order to identify application features in data-intensive programs. We search to export those identified features as services [12].
As the previous approach, it can be related to our proposed tool through SQL queries capture and analysis.

The two first presented approaches consider database interactions understanding as main component to ease reverse-engineering process. As previously explained, SQL statements analysis is pretty rare in the literature.

2.2.4 Improving Dynamic Data Analysis with Aspect-Oriented Programming

As discussed above, program static analysis is generally insufficient to obtain a complete comprehension of a program. So, this approach [14] aims to detail a particular way to dynamically analyse a program : The Aspect-Oriented Programming (AOP). AOP has already been introduced in Section 1.4. We can compare an aspect to a trigger that is activated when a precise phenomenon occurs. In order terms, an aspect is a code executed in addition to the program; an aspect is automatically launched when a defined part of program is executed. For example, if a particular method in the program is called during the execution, the aspect is activated.
A traditional use of AOP is logging action. In Figure 2.2, we defined an aspect launched when the method AddCustomer() is called at runtime. This aspect is a simple log tracing AddCustomer() method.

```
Aspect Logger {
  void AddCustomer(Customer cust) {
  logger.info("Adding customer..."); }
}
```

Figure 2.2: Logging Aspect

The selected article presents an approach based on aspect-oriented programming used for the reverse engineering process. AOP is used to obtain traces of a program's execution and this article shows AOP eases and reduces time necessary to obtain this information.

It is interesting to select this article because AOP techniques were kept by our tool (in particular for SQL query extraction).

2.2.5 Visualizing Dynamic Software System Information through High-level Models

This approach [22] does not handle SQL queries but studies interactions in a (object-oriented) program. It allows to analyse and visualize these inter-

actions: allocation/deallocation of objects, communication between methods (method calls), ...
Two aspects of this project are interesting:

- Visualizing communications in the program: it allows to translate technical information into an intuitive visualization. Moreover, our proposed tool is also based on studying of program communication (through SQL queries exchanges).

- This project allows also to abstract the process by choosing a high-level model: the software engineer can choose a high-level structural view to use as the basis for visualization by stating the names of the abstract entities. An entity may, for example, represent a system's component. Thus, a mapping between concrete and abstract entities is needed to obtain this high-level view.
  We will see later that one of our tool's advantages is the opportunity to abstract the process by choosing a high-level representation allowing to deeper understand the meaning of SQL queries.

2.2.6 Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information

This approach [20] aims to study static and dynamic information in a program in order to extract some high-level views (e.g. state diagram, class diagram, ...). We can find a link between this project and our project : indeed, it allows to extract high-level representations like Entity-Relationship model (ER model). We will detail ER model in Secion 3.1.1.

2.2.7 CodeCity

CodeCIty [23, 24, 25] is a 3D visualization tool supporting the analysis of large object-oriented software systems. This tool introduces a novel approach by using a city metaphor: it depicts classes as buildings and packages as districts of a "software city". This metaphor-based visualization provides a natural environment allowing a user to easily explore the structures of the analysed system. Other visual properties are used in the city metaphor:

- The number of methods represents the building height

- The number of attributes denotes the base size of the building

- ...

Such a representation allows us to analyse and detect possible design disharmonies as, for instance, classes with a high number of methods, a class with few attributes and many methods, ... In summary, CodeCity supports (among others) the identification of sensitive classes and packages that are crucial for comprehension and maintenance.

2.2.8 EXTRAVIS

This article [10] tackles a major issue of dynamic analysis: the scalability. Indeed, execution traces contains huge amounts of data and are not easily understood. The solution proposed by the authors is a novel tool-based approach allowing to visualize execution traces in an intuitive way. This tool, called EXTRAVIS, offers two synchronized views for analysing execution traces:

- A circular bundle view that proposes a detailed visualization of the system's structural entities and their interrelationships. It projects the system's entities (e.g. classes and packages) as well as their dynamic calling on a circle.
- A massive sequence consisting of a sequence diagram that provides an interactive overview of the trace.

The combination of the two views permits an easier comprehension of large execution traces. EXTRAVIS aims at three main goals:

1. **Exploratory program comprehension**: providing an overview of how the system works.
2. **Feature detection**: detecting the features present in the trace.
3. **Feature comprehension**: understanding how the detected features are implemented.

Conclusions of the State of the Art Through this chapter, we have briefly summarized the state of the art in the use of dynamic analysis for program comprehension, based on an existing survey.

We firstly show, with some statistics, that a lot of existing works that deal with dynamic program analysis focus on the analysis of inter-program communication (method calls, class dependencies, etc). But we noticed that

almost none of them address the analysis of program-database communication, especially through SQL execution trace analysis (and visualization). We can therefore conclude that the tool-supported approach proposed in this thesis covers an largely unexplored area and introduces an novel way to understand data-intensive programs.

Chapter 3

Database Engineering

Before introducing in detail our visualization tool, it is important to relate this subject with database engineering. Database engineering is the whole process leading to the design of the database. This process relies on[1] a disciplined approach divided into several phases. In this chapter, we will first describe in detail the database engineering process and then, we will emphasize the different concepts related to our dynamic program analysis approach and tool.

It is important to emphasize that we assume that the database used by the programs we will analyse is a **relational database**. A relational database is a collection of data items organized in tables in order to make data access easier [15].
Figure 3.1 gives an abstract representation of the database engineering process. During this process different database schemas are produced. A database schema represents a model, i.e., an abstract formal representation of a given application domain. Such a model allows to better understand this application domain and to build an operational database allowing to store and manipulate information about it. There exist different types of database schemas, belonging different levels of abstraction. The first schema produced during the process is the most abstract (platform-independent) while the last one is the most concrete (platform-specific) and exactly specifies the actual database structures.
Let us now describe each phase of the database building process.

Requirement analysis The *requirements analysis* phase, also known as *conceptual analysis*, consists in collecting the user requirements about the

[1] or let say, *should* rely on...

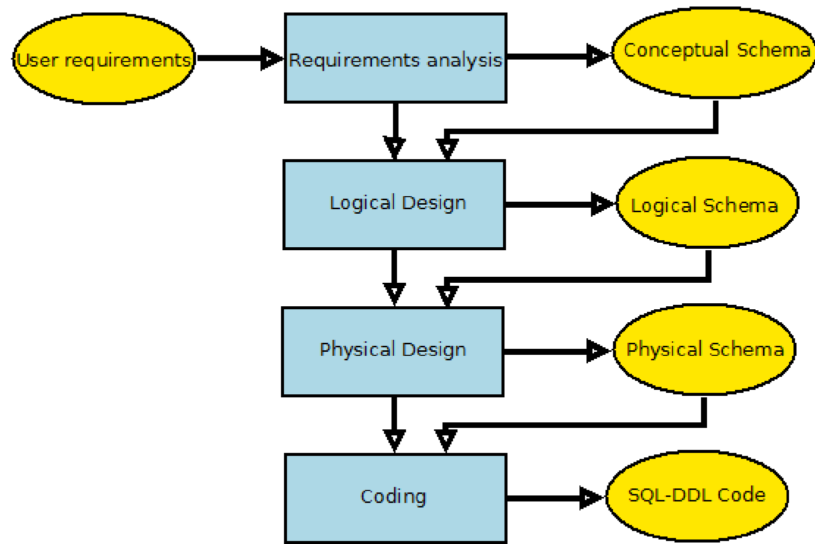

Figure 3.1: Relational database building - Process

application domain under consideration. With this requirements collection, the database engineer will product the **conceptual schema** of the future database. A conceptual schema extracts pertinent concepts of the domain of application from the user requirements, which are typically expressed in natural language.

Logical design The second phase, called *logical design*, consists in extracting the **logical schema** of the future database from the conceptual schema obtained so far. This logical schema describes more concretely the database structures, the relationships between these structures and the integrity constraints. It complies with a given logical model that is compatible with a given database paradigm.

Physical design The *physical design* phase consists in designing a **physical schema**. A physical schema is obtained by enriching the logical schema with technical constructs needed to make the future database efficient and robust (e.g., indexes and physical dbspaces).

Coding phase Finally the *coding* phase consists in translating the physical schema into the executable database code. The generated code will be compatible with the particular DBMS and will allow to define and create the database structures.

In summary, we observe that the database engineering process, similarly to the software engineering process, relies on a disciplined approach to model and design a database, from requirements analysis to coding.

3.1 Database schemas

A database schema is a schema expressed in a formal language that describes the database structure and explains the data organization. A schema defines also the relationships between the different data types. Furthermore, database schemas allow to define some integrity constraints and conditions on data ensuring the database integrity.

During this process, we observed that different database schemas are produced. Each schema has a level of abstraction lower than the schema from which it is derived. Indeed, each new produced schema is more technical and the final purpose is to automatically generate the database code from the most technical schema of the process. Now, we are going to study the database building process with a simple example by describing the schema obtained at each phase.

3.1.1 Conceptual schema

A conceptual schema is usually expressed in a high-level formalism, typically the Entity-Relationship model (ER model). The ER model is a graphical language describing the database in an abstract way. It defines three main types of concepts [15]:

- Entity types : we consider the application domain is composed of concrete and abstract entities. Each entity belongs to a class, called *entity type*.
- Relationship types : There exist some relationships between entity type instances. Each relationship belongs to a class, called *relationship type* in the conceptual schema.
- Attributes : each entity (or relationship) belonging to a same class may have common characteristics. Such a characteristic is called *attribute*. Entity and relationship types may own some attributes.

Let us now assume that we must design a database for a company selling products to their customers. Figure 3.2 represents the conceptual schema obtained after the requirements analysis step.

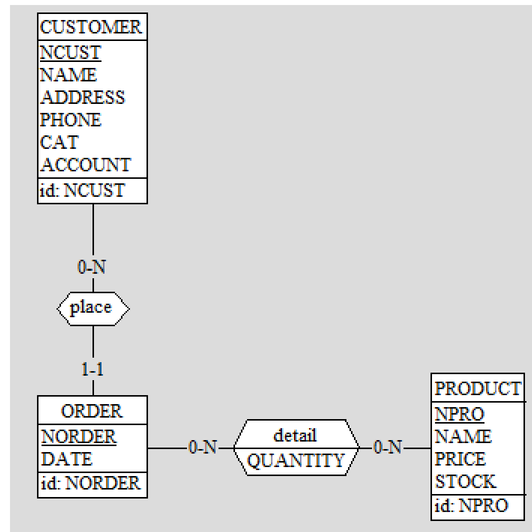

Figure 3.2: Customer-Order : conceptual schema

In this schema, one can see the entity types *customer*, *order* and *product*. We can also observe the relationship type *place* between *customer* and *order*, expressing the fact that a customer may place orders, and the relationship type *detail* defining the fact that an order is detailed by some products.
The entity types may have attributes; for instance, each customer has a customer number, a name, an address, a phone number, a category and has an account. Each ordered product is characterised by a quantity (attribute *quantity* of the relationship type *detail*).

3.1.2 Logical schema

A logical schema is expressed in a particular formalism, called a **logical model**, depending on DBMS. For example, the family of relational DBMS[2] is concerned by a particular logical model that is called the **relational model**. The relational model is a graphical language describing with a better precision the database structures, the relationships between structures and integrity constraints to respect for ensuring the database integrity.
It is important to note that, in this thesis, we consider programs that manipulate a relational database. Therefore, in the remaining of this thesis, we only focus on the relational model.

The relational model defines two main types of concepts [15]

[2] Oracle, Microsoft SQL Server, MySQL, DB2, PostgreSQL, Sybase, ...

- Tables: a database is an organized as a collection of tables. A table contains a collection of rows.

- Columns: each row contains a set of values. A value has a given type and the set of values of that type represents the *column*. Each table has one or several columns.

A logical schema may also define some constraints over the data stored in the tables. The most frequent constraints are the table *identifiers* and the *foreign keys*.
An identifier defines a *uniqueness* constraint. If a column (or a set of columns) is declared as the identifier (a.k.a *primary key*) of a table, then the latter cannot contain two distinct rows having the same value for that column (or set of columns).
A foreign key defines a *referential constraint*. The foreign key identifies a column (or set of columns) in one table (called *referencing* table) that refers to a column (or set of columns) in another table (called *referenced* table). If a set of columns of a table A is declared foreign key to a set of columns of a table B, then for each row of A, it exists one row of B such as the values of the referencing columns refers to the values of the referenced columns.

We can now define a relational schema as follows:
A schema is relational if it respects the following constraints [15]

1. the schema contains entity types, called tables.

2. each table has at least one column.

3. a column is single-valued and atomic; the column is optional or mandatory.

4. The table identifiers and foreign keys are the only data integrity constraints expressed in the schema.

The logical schema obtained by the logical design phase is illustrated by Figure 3.3.

This logical schema includes four tables *customer*, *orders*, *detail* and *product*.
Each table has several columns. For instance, a product has a number, a name, a price and a stock quantity.
Each table has also an identifier, respectively a customer number (*NCUST*), an order number (*NORDER*), a detail identifier (couple of columns *NORDER* and *NPRO*) and a product number (*NPRO*).

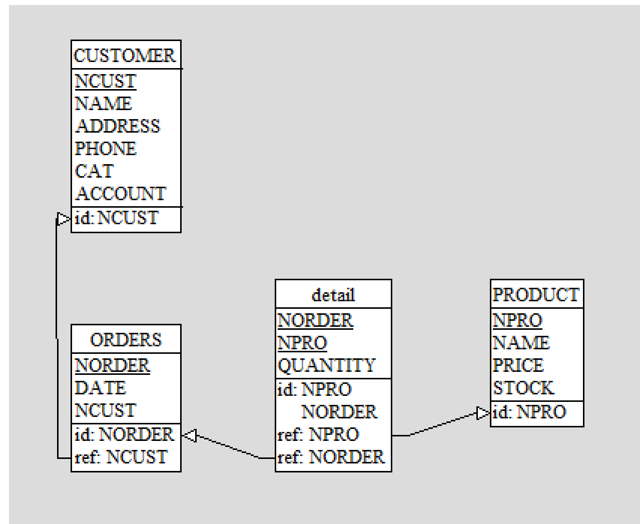

Figure 3.3: Customer-Order : logical schema

Three referential constraints (foreign keys) are declared. Each order refers to a customer, each detail refers to an order and to a product.

3.1.3 Physical schema

As previously explained, a physical schema is very similar to its corresponding logical schema. The only transformation consists in adding some technical schema constructs allowing to improve the efficiency and robustness of the future database. In this thesis, the formalism used for specifying the physical schema is also the relational model.
In our example, the physical phase consists only in declaring an index for each table identifier and for each foreign key. These indexes will allow a faster access to data.

3.1.4 Database code

The coding phase aims to generate the database code that translates the physical schema. The produced code is written in the DBMS-specific data definition language. Since we assume that our future database is relational, we chose SQL-DDL[3] as DBMS language.
The generated code defines the database structures. We can find the SQL-DDL code produced through the coding phase at Figure 3.4.

[3]Structure Query Language - Data Definition Language

```
(1) create table CUSTOMER (
(2)         NCUST numeric(6) not null,
(3)         NAME varchar(30) not null,
(4)         ADDRESS varchar(30) not null,
(5)         PHONE numeric(12) not null,
(6)         CAT varchar(2) not null,
(7)         ACCOUNT numeric(12) not null,
(8)         constraint ID_CUSTOMER_ID primary key (NCUST));
(9)
(10) create table detail (
(11)         NORDER numeric(6) not null,
(12)         NPRO numeric(6) not null,
(13)         QUANTITY numeric(6) not null,
(14)         constraint ID_detail_ID primary key (NPRO, NORDER));
(15)
(16) create table ORDERS (
(17)         NORDER numeric(6) not null,
(18)         DATE date not null,
(19)         NCUST numeric(6) not null,
(20)         constraint ID_ORDERS_ID primary key (NORDER));
(21)
(22) create table PRODUCT (
(23)         NPRO numeric(6) not null,
(24)         NAME varchar(20) not null,
(25)         PRICE numeric(4,2) not null,
(26)         STOCK numeric(5) not null,
(27)         constraint ID_PRODUCT_ID primary key (NPRO));
(28)
(29) alter table detail add constraint REF_detai_PRODU
(30)         foreign key (NPRO)
(31)         references PRODUCT;
(32)
(33) alter table detail add constraint REF_detai_ORDERS_FK
(34)         foreign key (NORDER)
(35)         references ORDERS;
(36)
(37) alter table ORDERS add constraint REF_ORDERS_CUSTO_FK
(38)         foreign key (NCUST)
(39)         references CUSTOMER;
```

Figure 3.4: DDL Code - Translation of logical schema

3.2 ER model subset considered

We previously explained that the conceptual schema is expressed in a particular formalism called the Entity-Relationship model (ER model). We introduced at Figure 3.5 a metaschema of the ER model .

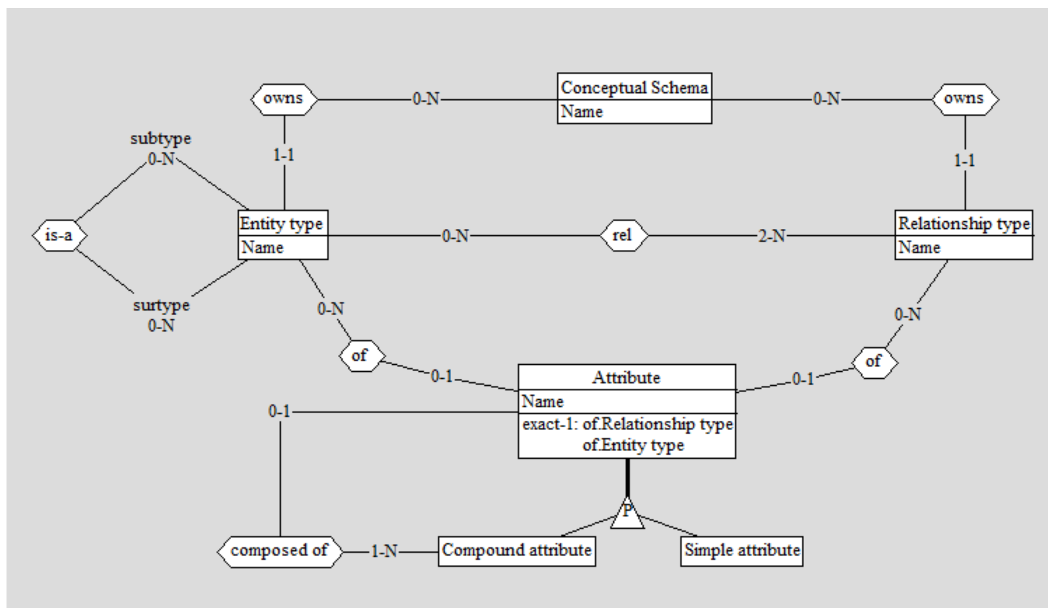

Figure 3.5: Metaschema of ER model

A conceptual schema describes the database application domain. The domain is composed of concrete/abstract entities belonging to a class. These classes are called **entity types**. In the domain, there exist some relationships between entity types. These relationships are called **relationship type**. Besides, an entity/relationship type may own some characteristics; these characteristics are called **attribute**. An attribute is either a **simple** attribute or a **compound** attribute. A simple attribute is an atomic attribute while a compound attribute is composed of other attributes. An example of compound attribute is shown at Figure 3.6.

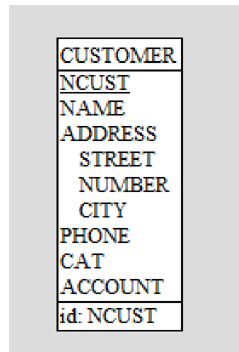

Figure 3.6: Example of compound attribute - ADDRESS

Furthermore, the ER model allows to represent a *hierarchy* of entity types: indeed, in a conceptual schema, several entity types may own a same super-type (entity type parent). The hierarchical relationship between an entity type and its parent is called an **is-a relationship**. In Figure 3.7, we can see an example of is-a relationship defining two different kinds of customer.

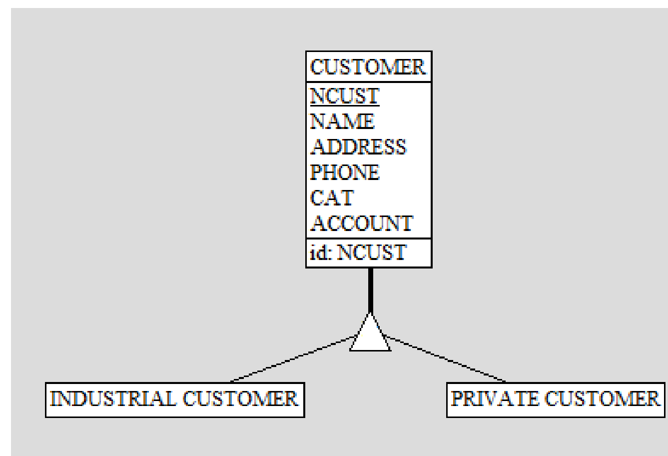

Figure 3.7: Example of IS-A relationship - CUSTOMER

However, given the potential complexity of an arbitrary ER schema, and considerinf the exploratory nature of this thesis work, we decided to make some working assumptions in order to restrict ourselves to a given subset of the ER model. These assumptions are composed of three main rules :

1. Simple attributes: our first conceptual assumption concerns attributes. Indeed, we assume each attribute is **simple**.

2. No is-a relationship: we assume there is no is-a relationship in the conceptual schema.

3. Binary relationship types : we assume each relationship type is binary. It means each relationship type is only between two entity types (not necessarily different because cyclic relationship type may exist). In Figure 3.2, the relationship types *place* and *detail* are binary. An example of non-binary relationship type is shown in Figure 3.8.

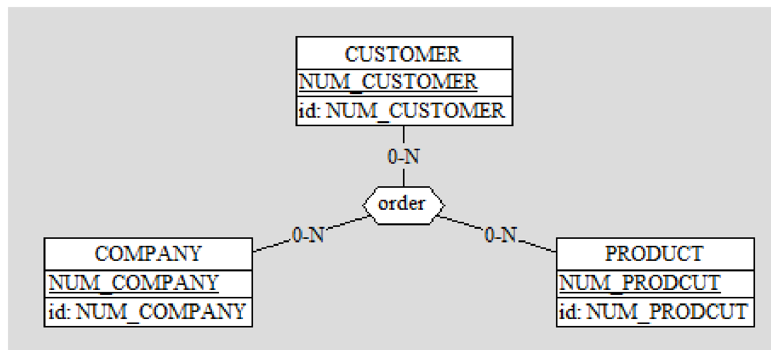

Figure 3.8: Example of non-binary relationship type - A customer orders some products for a particular company.

After making these three assumptions, we have to prove that the expressiveness of ER model subset is not reduced. In other words, we have to show that any ER construct that we exclude can be translated into valid constructs in our ER model subset.

1. Simple attributes: a compound attribute can be easily split into simple attributes (as shown in Figure 3.9).

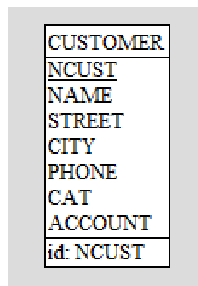

Figure 3.9: Compound attribute split into simple attributes - ADDRESS

2. No is-a relationship: an is-a relationship can be translated into binary relationship type(s). This transformation is illustrated in Figure 3.10.

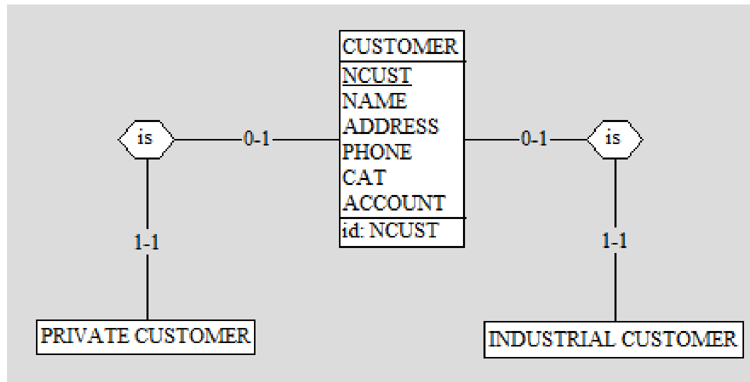

Figure 3.10: Transformation of a supertype into relationship types - CUSTOMER

3. Binary relationship types: a technique to avoid complex (non-binary) relationship types is to transform each complex relationship type R into an entity type E_R and to add a binary relationship type between E_R and each entity type initially attached to R. This transformation rule is illustrated in Figure 3.11.

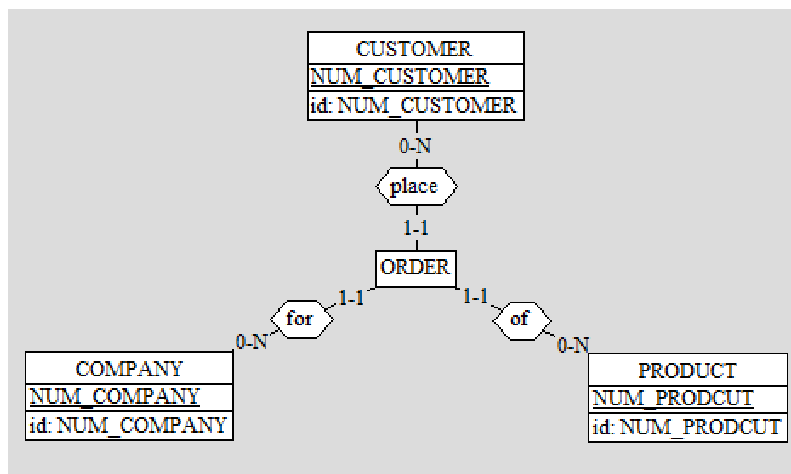

Figure 3.11: Transformation of a complex relationship type into binary relationship types - Customer ordering some products for a particular company.

Thus, we can conclude that our three assumptions does reduce the ER model syntax but not its expressiveness; indeed, one can easily find syntactic alternatives to the schema constructs we decided to exclude. From now on, we will only consider the ER model subset summarized by Figure 3.12. This subset is obtained by making our three assumptions.

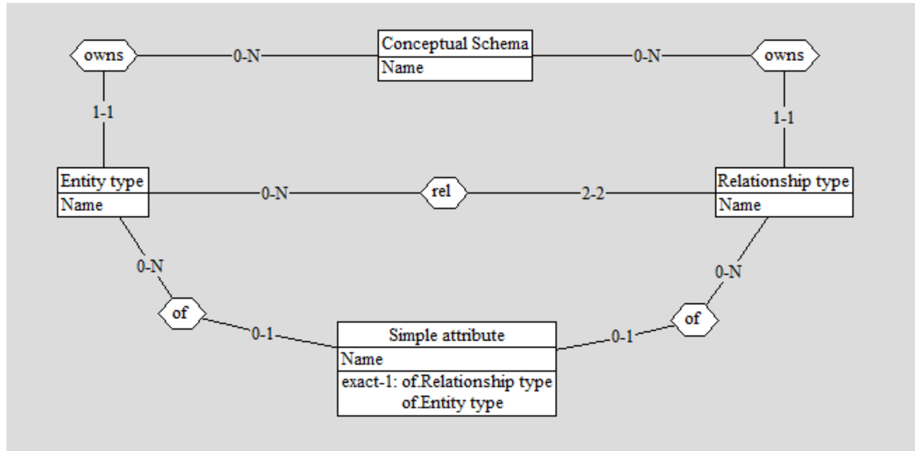

Figure 3.12: Metaschema of the ER model subset considered in this thesis.

3.3 Schema mapping

In the database engineering process, we observe that the logical schema produced during the logical design phase is based on the conceptual schema obtained during the requirements analysis step. Consequently, there is a direct link between the conceptual and logical schema elements. Indeed, in this thesis we can consider the logical design phase as the translation of an ER schema into a relational schema and more precisely, as the translation of conceptual objects into relational objects. Therefore, there exists a so-called *mapping* between the conceptual and logical schemas. This correspondence is summarized and illustrated in Figure 3.13. This Figure establishes the correspondence between the three main conceptual objects (entity types, relationship types and attributes) and the three main logical objects (tables, columns and foreign keys) described in Section 3.1.1 and 3.1.2.

We can actually summarize Figure 3.13 with the following list of propositions.

1. An entity type maps to a table.

2. A relationship type maps to either a table or a foreign key.

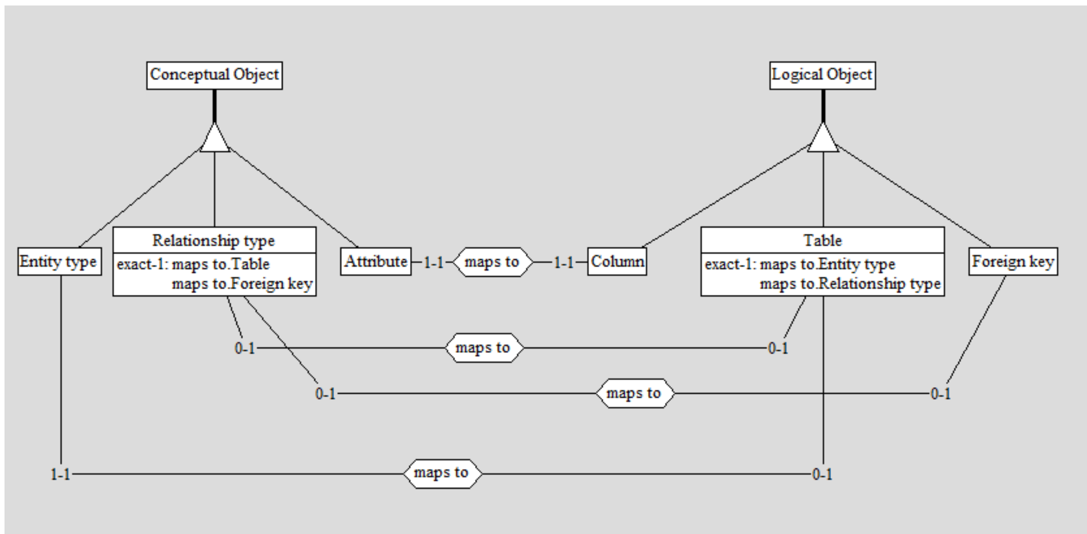

Figure 3.13: Correspondence between conceptual and logical schema expressed by the relational model - Overview

3. An attribute maps to a column.

4. A column maps to an attribute.

5. A table maps to either an entity type or a relationship type.

6. A foreign key maps to either a relationship type or nothing; indeed, a foreign key may have no conceptual mapping.

If we get back to our previous example and analyze the correspondence between the conceptual schema (Figure 3.2) and the logical schema (Figure 3.3), we obtain the following mapping table:

	Conceptual object	**Logical object**
1	**CUSTOMER**	CUSTOMER
1.1	-NCUST	CUSTOMER(NCUST)
	”	ORDERS(NCUST)
1.2	-NAME	CUSTOMER(NAME)
1.3	-ADDRESS	CUSTOMER(ADDRESS)
1.4	-PHONE	CUSTOMER(PHONE)
1.5	-CAT	CUSTOMER(CAT)
1.6	-ACCOUNT	CUSTOMER(ACCOUNT)
2	**place**	ORDERS-ref:NCUST
3	**ORDER**	ORDERS
3.1	-NORDER	ORDERS(NORDER)
	”	detail(NORDER)
3.2	-DATE	ORDERS(DATE)
4	**detail**	detail
4.1	-QUANTITY	detail(QUANTITY)
5	**PRODUCT**	PRODUCT
5.1	-NPRO	detail(NPRO)
	”	PRODUCT(NPRO)
5.2	-NAME	PRODUCT(NAME)
5.3	-PRICE	PRODUCT(PRICE)
5.4	-STOCK	PRODUCT(STOCK)

Table 3.1: Example of correspondence between logical and conceptual schemas

This table is composed of two columns: the first one lists all the conceptual objects (entity type, relationship type and attribute), the second one lists all the logical objects (table, column and foreign key) matching to each conceptual object.

Through this section, we showed that the logical design allows to keep a traceability link between the conceptual and logical schemas, called schema mapping.

3.4 SQL

SQL is a high-level language designed for managing data stored in a relational database. It allows programmers to define the data structures of a

database but also to manipulate data stored in that database. SQL is indeed divided into two specific sub-languages:

1. defining data structure : **SQL-DDL**
2. manipulating data : **SQL-DML**[4]

3.4.1 SQL-DDL

SQL-DDL is a language allowing programmers to define data structures and constraints. SQL-DDL is the database code resulting from the coding phase of the database engineering process. That phase consists of a direct translation of the physical schema into executable DDL code allowing to create database structures.
SQL-DDL proposes two main types of operations :

- Creating operation : aiming to create database objects (table, column, index, ...). Figure 3.14 shows an example of DDL code that creates a table of customers.

```
(1) create table CUSTOMER (
(2)         NCUST numeric(6) not null,
(3)         NAME varchar(30) not null,
(4)         ADDRESS varchar(30) not null,
(5)         PHONE numeric(12) not null,
(6)         CAT varchar(2) not null,
(7)         ACCOUNT numeric(12) not null,
(8)         constraint ID_CUSTOMER_ID primary key (NCUST));
```

Figure 3.14: DDL example - Creating table CUSTOMER

- Updating/Deleting operation : aiming to update or delete database objects (database, table, column, index, ...). Figure 3.15 illustrates an example of DDL code deleting a table.

```
(1) ALTER TABLE CUSTOMER DROP COLUMN CAT;
```

Figure 3.15: DDL example - Deleting column

[4]DML : Data Manipulation Language

3.4.2 SQL-DML

SQL-DML is a language allowing programmers to insert/extract/update/delete data in a database. SQL-DML proposes four main types of operations:

- Inserting operation : used in order to insert data in database. An example of inserting operation is shown at Figure 3.16.

```
(1) INSERT INTO PRODUCT VALUES ('PRO5600', 'Wood plank',
'50.25', '1000' );
```

Figure 3.16: DML example - Inserting new product

- Selecting operation : used in order to extract data. An example is shown at Figure 3.17.

```
(1) SELECT NCUST FROM CUSTOMER WHERE NAME = 'Dupont';
```

Figure 3.17: DML example - Selecting CUSTOMER

- Updating operation : used in order to update data. Figure 3.18 illustrates an example of updating operation.

```
(1)UPDATE PRODUCT SET STOCK = 500 WHERE NPRO =
'PRO5600';
```

Figure 3.18: DML example - Updating product's stock

- Deleting operation : used in order to delete data. An example is shown at Figure 3.19.

```
(1) DELETE FROM PRODUCT WHERE NPRO = 'PRO5600';
```

Figure 3.19: DML example - Deleting product

As previously explained, the thesis purpose is to introduce a tool able to ease the comprehension of data-intensive programs through the analysis of SQL execution traces. However, DDL operations correspond to technical operations (creating/modifying data structures). This is the reason why, in the context of this thesis, we have no particular interest in the analysis of data structure modification operations.
Therefore, we will only focus on SQL-DML operations and particularly on **select**, **insert**, **update** and **delete** queries. All other type of SQL queries will not be considered in this thesis.

3.5 Database reverse engineering

During its life, a database is usually faced with evolution; the developers often have to make changes to their database structures and constraints in order to adapt the database to ever-changing needs. However, such changes are not always trivial and can represent a complex task. Indeed, many existing (legacy) databases have not been designed in a disciplined way and sometimes, no systematic database engineering process was followed during the design phase. Therefore, it is common to encounter databases that are poorly/not documented. Even worse, it is not rare that the DDL code actually constitutes the only available documentation of the database.
The recovery of missing database schemas, and in particular of the conceptual schema, is called **database reverse engineering**. The database reverse engineering is globally considered as the extraction of conceptual schema from the DDL code. This process is illustrated by Figure 3.20.
The database reverse engineering process is composed of three successive steps:

1. Physical extraction: this phase aims to produce the physical schema from the DDL code. The physical extraction produces an *raw* physical schema, that contains all the *explicit* constructs present in the DDL code but does not include potentially *implicit* database constructs like, for example, undeclared foreign keys.

2. Logical reconstruction: this phase aims to produce a *refined* logical schema from the physical schema by analysing additional artifacts such as programs source code. For instance, implicit schema constructs can be recovered by analysing additional SQL code (SQL procedures, triggers, ...), program code using the database, ...

3. Conceptualization: this final step produces the database conceptual schema from the refined logical schema.

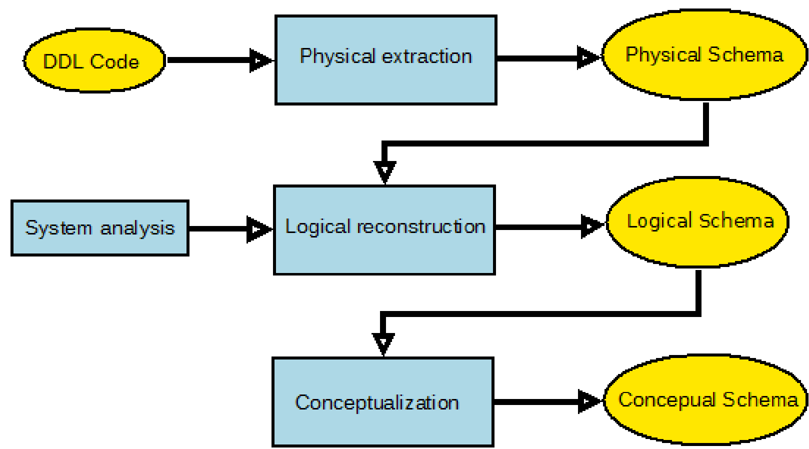

Figure 3.20: Database reverse engineering process

Chapter 4

DAViS

4.1 Context

Nowadays, more and more software systems are data-intensive and therefore depend on data exchange. Data-intensive systems focus on communication between the programs and their database. Let us remind that program evolution/maintenance, in particular program comprehension, is our major issue. Thus, in such systems, program-database interactions represent one important aspect and therefore, understanding the database manipulation behaviour of the programs is central when trying to gain an in-depth understanding of those programs. The communication between database and programs is generally performed through SQL queries sent by the programs.

Therefore, our idea to implement a tool, that would help to program comprehension, appeared; more generally, this tool would allow to analyse and understand the queries sent to the database. Our goal is here to propose a tool enabling (1) to capture SQL traces and (2) to visualize those traces in order to understand their impact on the database. This is why we decided to call this tool *DAViS* because it supports program comprehension through **Dynamic Analysis and Visualization of SQL traces**.

The context targeted by DAViS is a traditional program-database context in which communication between both is performed through SQL query/results exchange. Such a context is illustrated by Figure 4.1.

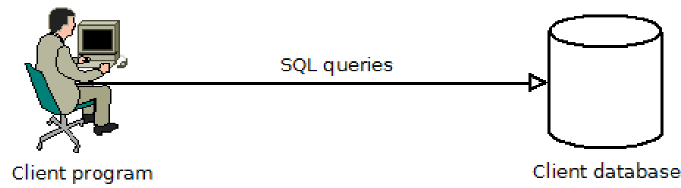

Figure 4.1: Program-Database context

The main objective of DAViS is to analyse the SQL queries sent by a client program and to provide a higher-level language allowing to translate the traces into a dynamic visualization.
However, our tool can be used in two different scenarios :

1. Offline method
2. Online method

4.2 Offline method

The offline method consists in two different phases executed at different times:

1. Trace capture phase : this first step aims to collect SQL queries generated during program execution. Once those queries are collected, they are stored into any storage space (e.g., a file, a database, ...). This step is represented by Figure 4.2.

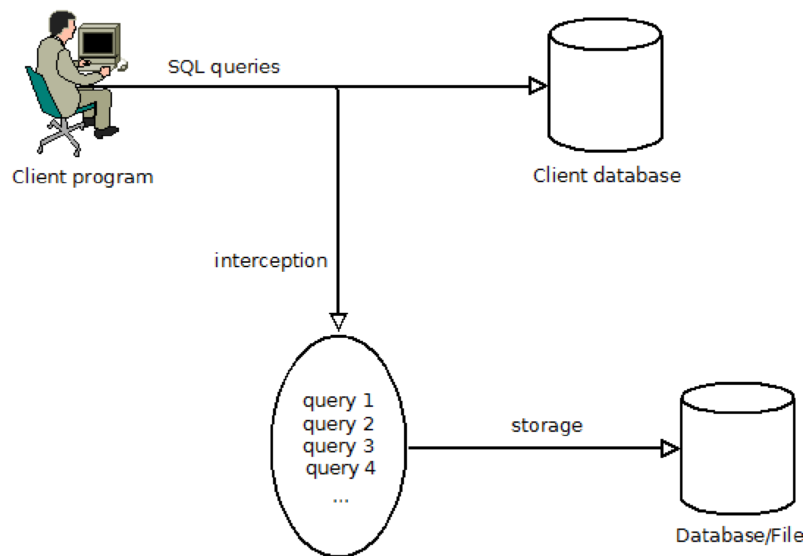

Figure 4.2: Offline method - First phase

We must notice that this step is not managed by the tool. In other words, this phase has to be performed by the engineer himself. Different techniques to intercept a query are explained in detail in Section 1.4. The user is free to choose her favourite technique.

2. Visualization phase : this second step, fully covered by DAViS, aims to get the traces previously stored into a storage space and to visualize their impacts on the database. The visualization consists in extracting pertinent technical information from a SQL trace, translating it into a graphical language and displaying the results in an intuitive way.
But first, we need to parse the query in order to extract relevatn data. We can consider query parsing as the transformation of the query into another format (usable by the tool) that only contains significant information.

Once the query is parsed, the visualization process requires two inputs :

- The parsed SQL query
- The database schema(s) describing the data organization of the target database.

Indeed, a matching between schema and information parsed from SQL query is computed and the process output results in a dynamic visualization of the

query behavior. The database schema is used as a visualization model. This second step is illustrated by Figure 4.3.

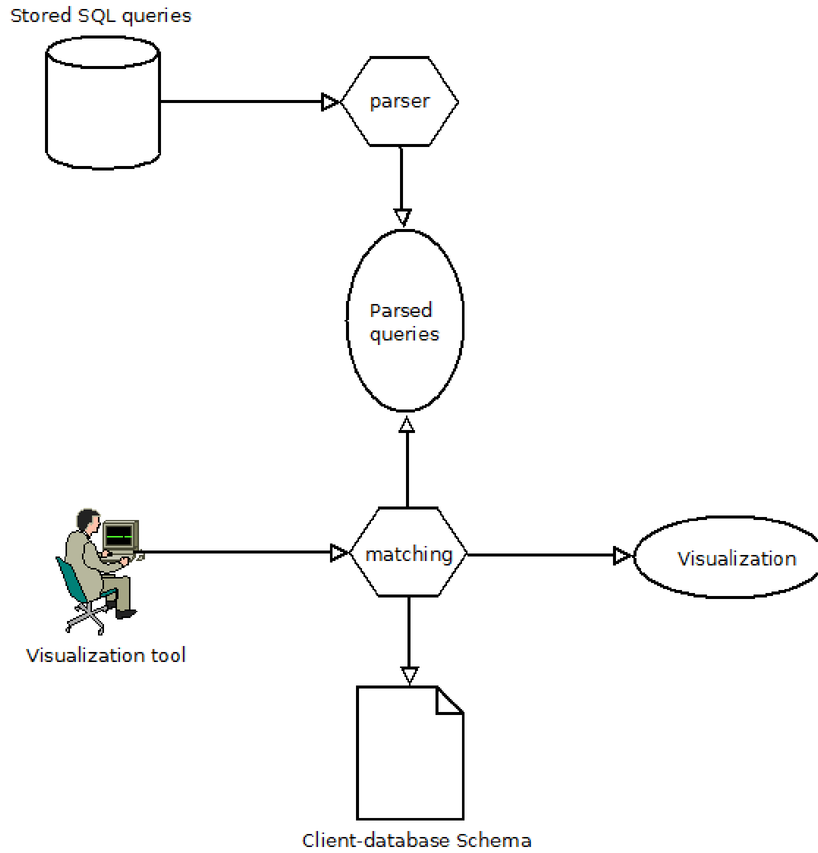

Figure 4.3: Offline method - Second phase

This method is called *offline* because the program execution and the visualization process are two separated processes that are executed at different times. The whole offline method is summarized by Figure 4.4.

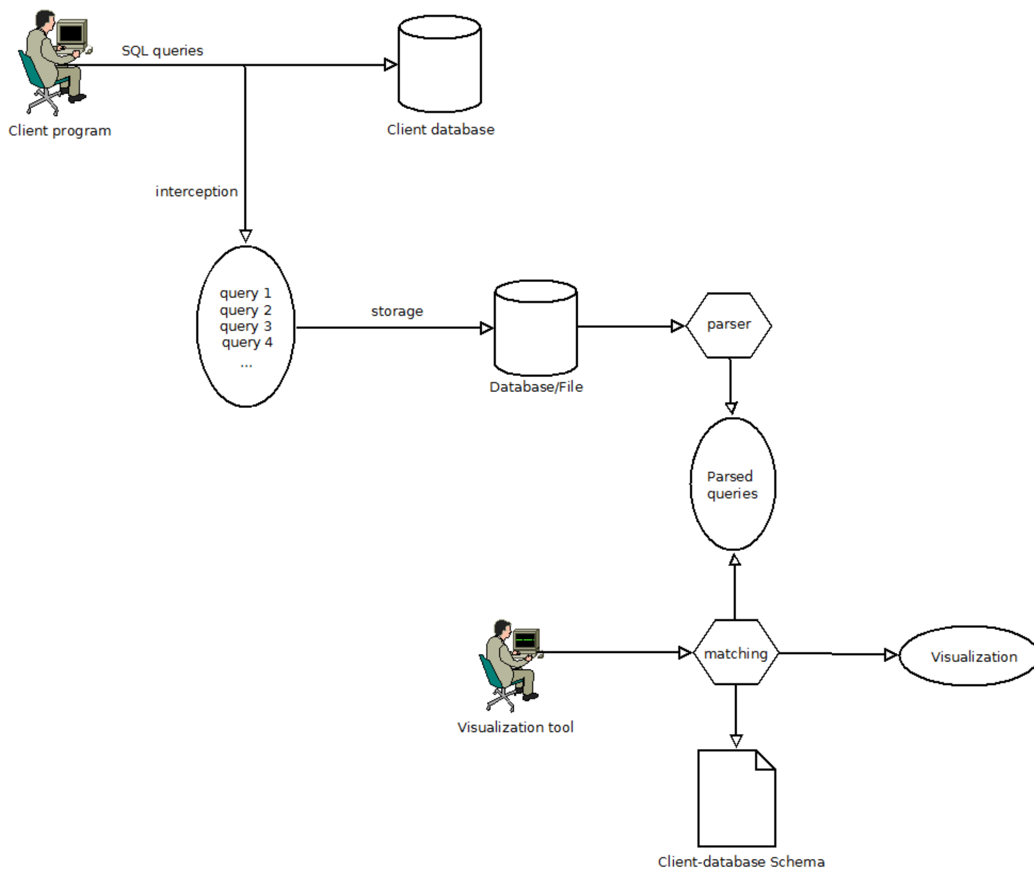

Figure 4.4: Offline method - Global view

4.3 Online method

The online method, also called *on-the-fly* method, consists in two phases executed at the same time: during the execution of the client program. These two phases are similar to the offline method. The online method is illustrated by Figure 4.5.

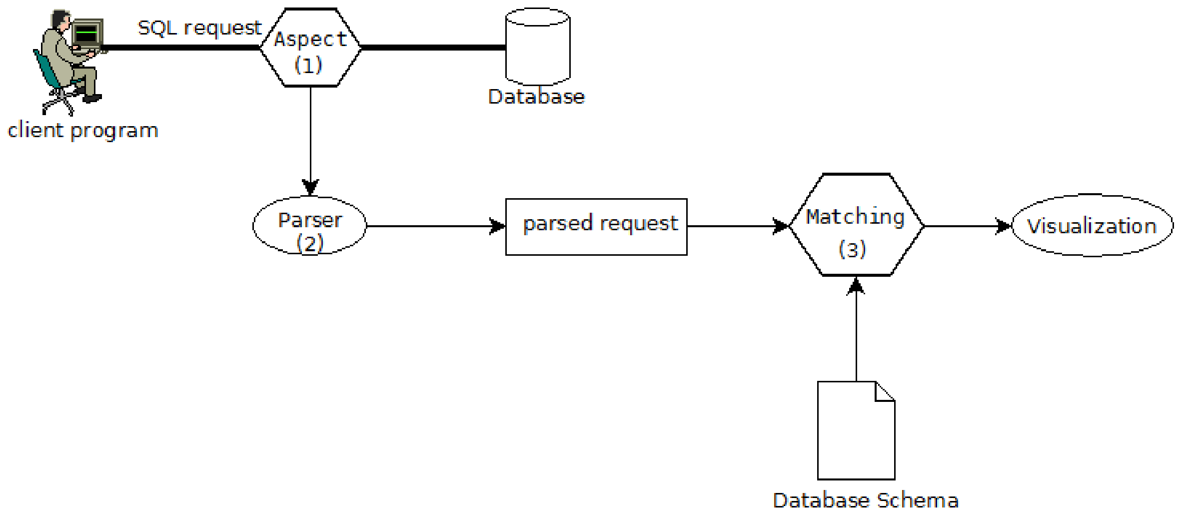

Figure 4.5: Online method - Overview

1. Query interception phase : this first step aims to collect SQL queries generated by the program at runtime. The difference with the offline method is that once intercepted, the query is not stored into a storage space but directly processed in order to be visualized.
   (1) In contrast with the offline method, the interception process is completely managed by the tool by means of *aspects*. This programming technique does not require any modification of the program source code.
   (2) Once intercepted, the query is parsed (the parsing process is the same as in the offline method) and transformed into a format usable by the tool.

2. Visualization phase : this step is exactly the same as in the offline method; it takes the database schema and the parsed query as inputs in order to visualize the matching between both (3).

To conclude, we can see that the online method allows to control and visualize in **real time** the SQL queries generated by the client-program while the offline method processes queries in differed time.

Each method has its own advantages. Indeed, the offline method allows to filter the queries we want to handle while the online method offers a real time control and analysis about actions on the database. Therefore, we can also say that the advantages of one method are the disadvantages of the other one.

We can synthesize DAViS by building a diagram modelling the different proposed features. This diagram, illustrated in Figure 4.6, is based on a part of the feature model's syntax but does not totally respect all its principles; indeed, we can see our diagram is not really a tree because some nodes have more than one parent.

Up to now, we introduced the online and offline methods. We now describes every feature of this diagram. Each feature represents a functionality of DAViS.

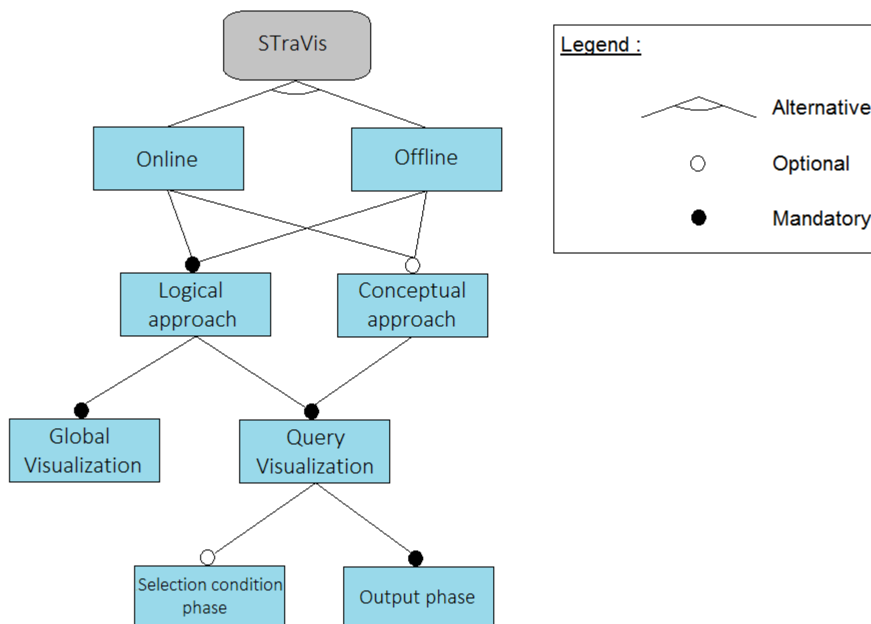

Figure 4.6: Summary of DAViS features.

4.4 Two levels of abstraction

In previous chapters, we saw two different possible modelling ways during the database design process: the logical and conceptual schema.

On the one hand, we observed the conceptual schema allows to define and describe the abstract concepts from the application domain. It permits to detail the domain by extracting the high-level items and by classifying these items by means of concepts (entity/relationship types). Thus, we can

consider a conceptual schema as an abstract representation/translation of the domain (described by the user).
Once the conceptual standards are acquired, the application domain can be understood by reading the conceptual schema of the database.

On the other hand, the logical schema is generated from the conceptual schema. It consists in translating the conceptual objects into logical objects understandable by a DBMS. Nevertheless, a logical schema will always be less intuitive and more technical than the corresponding conceptual schema. Thus, a logical schema is often insufficient to precisely understand the domain.

It appears that conceptual and logical schemas constitute two different levels of abstraction (respectively, an abstract and a technical level) on which an analysis of the communication between a program and a database is possible. By these two levels of abstraction, we have two representation models. We can define a representation model as a model on which we visualize a SQL query in order to understand its impact on the database. That is why we can developed two approaches according to which a SQL query can be visualized: the **logical** and the **conceptual** visualization approaches.

4.4.1 Logical approach

The logical approach aims to extract SQL objects from a SQL query and then to translate these SQL objects into corresponding **logical** objects. Thus, in this approach, the database logical schema is used as representation model.

However, SQL is a complex language with a lot of different versions. Due to this complexity, it is very difficult to fully cover the SQL language with our tool. Thus, we had to make a choice about the SQL limits accepted by DAViS. So, we limited the visualization to SQL-92. As explained in Section 3.4.2, our visualization tool manages only CRUD[1] operations. In SQL-92 queries, the CRUD operations owns three main types of object :

1. table
2. column
3. foreign key

NB: further SQL syntax assumptions are discussed in the Section 6.5.1.

[1]CRUD : Create, Update, Read, Delete

As already explained, the logical schema is a technical model of the database. It contains logical objects understandable by a DBMS. In other words, we can consider the logical schema as a graphical representation of the DDL code. Indeed, each SQL object can be translated into a logical object. Therefore, the rules for matching a SQL object to a logical object rely on one-to-one mappings.
The matching rules for each object type are described in Table 4.1.

	SQL-92 object	**Logical object**
1	Table	Table
2	Column	Column
3	Foreign key	Foreign key

Table 4.1: Matching rules between SQL object and logical object

The first rule expresses that a table present in a SQL query corresponds to a table of the logical schema. The second rule says that a column present in a SQL query corresponds to a column of the logical schema. The third rule means that a foreign key present in a SQL query corresponds to a foreign key of the logical schema.

We illustrate these three rules through a simple example of SQL query. Let us assume we use the logical model depicted at Figure 3.3. Given the following SQL query:

```
select NCUST from CUSTOMER;
```

We can extract two objects : *NCUST* and *CUSTOMER*. The latter is a SQL table and represents a logical table. While the former is SQL column and represents a logical column **but** represents also the foreign key of *CUSTOMER* and thus, *NCUST* represents both a column and a foreign key.

To conclude, we can say that the main goal of the logical approach is to analyse the communication (SQL queries) between a program and its database by using the logical database schema as representation model. Therefore, we see that this approach allows to **logically** understand the program-database communication.

4.4.2 Conceptual approach

The conceptual approach aims to extract SQL objects from a SQL query and then, to translate these SQL objects into corresponding **conceptual**

objects. In this approach, the database conceptual schema is used as representation model. Therefore, we need to establish the different matching rules enabling this conceptual translation.
However, given the previous one-to-one matching rules between SQL objects and logical objects, defining matching rules between SQL objects and conceptual objects can be seen as the definition of matching rules between logical objects and conceptual objects. Thus, the conceptual approach can be considered as the translation of logical objects into corresponding conceptual objects.

The different rules are expressed in Table 4.2 and are directly extracted from Figure 3.13.

	Logical object	**Conceptual object**
1	Table	Entity type
		Relationship type
2	Column	Attribute
3	Foreign key	Relationship type
		nothing

Table 4.2: Matching rules between logical objects and conceptual objects

1. **First rule** : a table can be translated into two possible conceptual objects: an entity type or a relationship type.
   If we consider the matching between the logical schema of Figure 3.3 and the conceptual schema of Figure 3.2), we notice that table *detail* matches to relationship type *detail*, while table *CUSTOMER* matches to entity type *CUSTOMER*.

2. **Second rule** : a column is translated into an attribute.
   Illustration: each column of the logical schema (Figure 3.3) corresponds to an attribute of the conceptual schema (Figure 3.2).

3. **Third rule** : a foreign key does not always correspond to a conceptual object. A foreign key corresponds either to a relationship type or to nothing.
   For instance, in Figure 3.3, the foreign key of table *ORDERS* corresponds to the relationship type *place* of Figure 3.2 while the foreign key (*NPRO*) of the table *detail* has no direct conceptual counterpart.

Matching between logical and conceptual schema Up to now, we observe that the conceptual approach uses the conceptual schema as repre-

sentation model. This approach consists in translating logical objects into conceptual objects. Thus, in addition to the conceptual schema, this approach needs the logical schema as input, but also the mapping between both schemas. This point is discussed in Section 3.3. The translation of logical objects into conceptual objects is allowed by the correspondence between the two schemas.

Database reverse-engineering phase Since the conceptual approach requires the database conceptual schema, a database reverse-engineering process might be necessary if the conceptual schema is missing. It would consist in retrieving the conceptual schema from the logical one (see Section 3.5).

To summarize the conceptual approach, we can say that the main purpose is to analyse the communication (SQL queries) between program and database by using the conceptual schema as representation model. Thus, we see that this approach allows to **conceptually** understand the program-database communication.

In this section, we introduced the logical and conceptual approaches implemented by our visualization tool, DAViS. Each approach consists in visualizing a SQL query by using a particular representation model on which the visualization is based. The logical approach considers the logical schema as representation model while the conceptual approach considers the conceptual schema. Indeed, the logical approach proposes a *technical* visualization based on the logical schema, while the conceptual approach proposes an *abstract* visualization based on conceptual schema. The first one allows a comprehension of the technical impacts on the database whereas the second one proposes an abstract and high-level comprehension.

4.5 The two visualizations

In Figure 4.6, we can also observe that DAViS implements two kinds of visualization: the *global* visualization and the *query* visualizations. We now describe the purposes of each of them.

4.5.1 The query visualization

The query visualization aims to visualize the impact of one SQL query on the database. It consists in analysing a given query in order to comprehend

its behaviour. The query visualization eases program comprehension by separately analysing each query sent to the database. In summary, it aims to represent the meaning of a single query.

4.5.2 The global visualization

While the query visualization focuses on the comprehension of a given query, the global visualization targets the understanding of the global impact of a *set* of queries, typically corresponding to an entire program execution scenario. Indeed, since each query has its own impact on the database, we visualize the *sum* of these impacts.

As shown by Figure 4.6, the logical approach proposes the two visualizations, whilst the conceptual approach does not support global visualization. Indeed, since the main purpose of the logical approach is to observe the technical impacts of a SQL query on the database, the logical approach proposes the two visualizations: analysing the **global impact** of an execution scenario with the global visualization and separately analysing the impact of each SQL query of this execution scenario. However, since the high-level comprehension of SQL queries is the main aim of the conceptual approach, we decided the conceptual approach will only propose the query visualization. However, it would be technically possible to support global visulization using a conceptual approach too.

4.6 The two-phase principle

As explained in Section 3.4.2, DAViS focuses only on DML queries and thus CRUD operations. A SQL query can be divided into two parts:

1. The first part contains all the selection conditions of the query.

2. The second part contains the query result.

Here we present a simple example of query aiming to select all the products ordered by the customer '1245' :

```
(1)   select NPRO
(2)   from detail D, ORDERS O
(3)   where D.NORDER = O.NORDER
(4)   and O.NCUST = '1245';
```

We notice that there are two selection conditions: a condition joining the details of table 'detail' with the orders of table 'ORDERS' (3) and a condition on the customer (4), while the query result represents the selection of product numbers (1).

However, it is important to notice that an insert query does not have any selection condition: such a query only consists of new inserted values. Only select, update and delete queries may have selection conditions.
Regarding the result, each query type has its own category:

- Insert query: the result is represented by the inserted values.
- Update query: the result is represented by the updated values.
- Select query: the result is represented by the selected values.
- Delete query: the result is represented by deleted lines of database.

Therefore, we can split the query comprehension process into two phases: firstly, understanding the selection conditions of the query (**selection condition** phase) and secondly, understanding the results of the query (**output** phase). Furthermore, we can apply the two-phase principle on the query visualization (Section 4.5.1); indeed, the query visualization proposed by DAViS is divided in two steps:

1. First, visualizing the selection condition phase. This phase is optional; indeed, some queries do not have selection conditions.
2. Second, visualizing the output phase.

To conclude the two-phase principle, we observe that a SQL query is divided in two parts: the selection conditions and the query result. Thus, we noticed that the decomposition of the query visualization allows a better comprehension: first, we visualize the selection conditions and second, we visualize the query result.

We can summarise this chapter by the following table:
The first column lists all the features of the Figure 4.6, while the second one describes the main purpose of each feature.

Feature	**Purpose**
DAViS	Understanding the communication between the programs and the database
Offline method	Understanding the SQL queries stored in a storage space
Online method	Understanding the SQL queries sent by the program at runtime
Logical approach	Understanding the SQL queries by using the logical schema as representation model, leading to a technical comprehension of the queries.
Conceptual approach	Understanding the SQL queries by using conceptual schema as representation model, leading to a high-level comprehension of the queries.
Query visualization	Visualizing the impact of a given SQL query on the database.
Global visualization	Visualizing the global impact of a set of SQL queries on the database.
Selection condition phase	Visualizing the selection conditions of a given query.
Output phase	Visualizing the result of a given query.

Table 4.3: Purpose of each feature

Chapter 5

Visualization

In the previous chapter, we have detailed the theoretical principles of DAViS. In this chapter, we describe the properties of the visualizations implemented by our tool and we illustrate them by means of a few examples.

5.1 Graph-based representation

In the graph theory, a graph is a set of points such as some points are connected by one or several links. These links may be oriented and, in this case, the graph is called *oriented graph*. A point is called *node* (or *vertex*) while a link between nodes is called *edge*.

We previously explained the logical and conceptual approaches use database schemas (logical/conceptual schemas) as representation model in order to visualize the SQL queries. Moreover, the (logical or conceptual) representation model consists of a graph-based representation: a node represents an object and an edge represents a relationship between two objects (represented by the nodes it connects). The two visualization approaches define their own category of nodes and edges.

Logical approach The logical approach has two kinds of nodes and four kinds of edges.

- Nodes: tables and columns
- Edges:
    1. Relationship between a table and its column: this relationship merely represents a parental link (containment).

2. Foreign key: it represents a foreign key between two tables.
3. Conjunction relationship: it represents a relationship between two tables. Two tables are *joint* if they are present in the same query.
4. Join: it represents a relationship between several columns. A join is used to couple two or more tables, based on relationships between several columns of these tables. We can see an example of join in the Section 4.6.

Conceptual approach The conceptual approach has three kinds of nodes and two kinds of edges.

- Nodes:
  1. Entity type
  2. Relationship type: despite its name, a relationship type is considered as a node.
  3. Attribute
- Edges:
  1. Relationship between an entity/relationship type and its attribute: this relationship merely represents a parental (containment) link.
  2. Relationship between an entity type and a relationship type: this relationship means that an entity type plays a role in a relationship type.

Through Figures 5.1 and 5.2, one can observe an example of logical and conceptual visualizations.

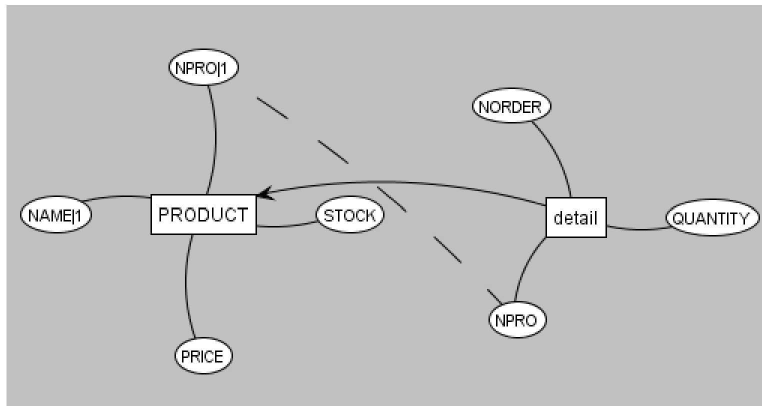

Figure 5.1: Example of logical visualization based on graph representation - the arrow represents a foreign key while the dotted line represents a join relationship. A continuous line represents a parental (containment) link between a table and its column(s).

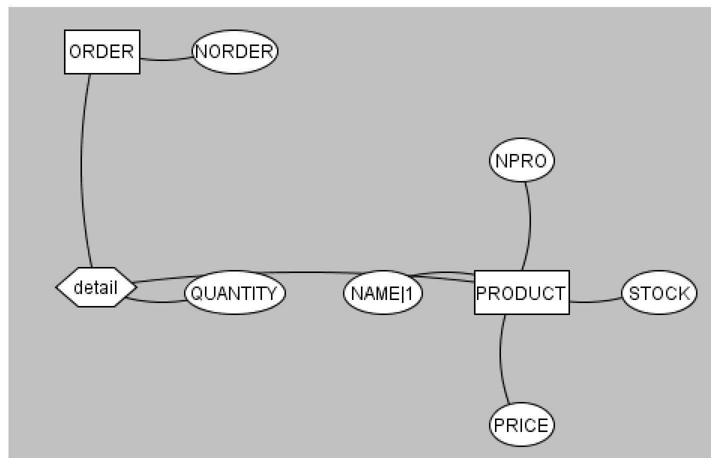

Figure 5.2: Example of conceptual visualization based on graph representation.

NB: In the logical and conceptual approaches, all parental edges (table-column and entity/relationship type-attribute) are based on a circular shape: the parent is at the center and its children form a circle around it. This pattern is an implementation choice but it may be replaced by the pattern "*container*" in which all children would be placed inside the parent.

5.2 Global visualization

The global visualization consists in visualizing the global impacts of a set of SQL queries on the database. Let us consider the following context:

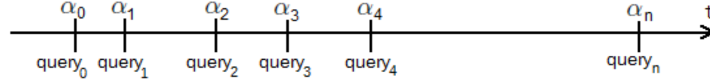

Figure 5.3: Context applied on a time line.

This context represents a scenario where a program sends successively *n+1* queries to a database. The first query is sent at time α_0, the second at time α_1 and so on. We want to analyse the global impact of this scenario by using the global visualization.

Usage frequency A SQL query contains some logical objects, such as columns and tables, but also creates some relationships between objects such as join and conjunction relationships. Thus, one can observe that a query contains some logical nodes and edges defined by the logical approach, as explained in Section 5.1.
We decided to attach a property to each object and relationship: a **usage frequency**. The usage frequency represents the number of queries in which the object/relationship appears. Therefore, we can define a function returning the usage frequency of an element at a given time, as follows:

Given O, the set of logical objects, and R, the set of relationships,
$\forall x \in \{O \cup R\}, \forall i \in \{0, ..., n\}$,
the function $freq_i(x)$ denotes the frequency of x at time α_i such as:
$freq_i(x) = freq_{i-1}(x) + \delta_x(query_i)$, with $\delta_A(B)$ =1 if $A \in B$, otherwise 0.
And thus, recursively, we obtain: $0 \leq freq_i(x) = \sum_{j=0}^{i} \delta_x(query_j) \leq (n+1)$

This recursive formula shows the frequency of an element may evolve in a given time interval.
This is the reason why DAViS gives the possibility to visualize the frequency of each element at time $\alpha_{i \in \{0,...,n\}}$. Thanks to this feature, DAViS allows to precisely analyse the frequency evolution.

5.2.1 Thickness pattern

The frequency visualization is enabled by the **thickness pattern**. The thickness pattern is based on the following principle: the higher the frequency, the thicker the graph component. Therefore, it constitutes a graphical principle allowing to detect the most *"requested"* logical objects/relationships. An example of global visualization is presented in Figure 5.4. The figure shows the global visualization of a set of queries.

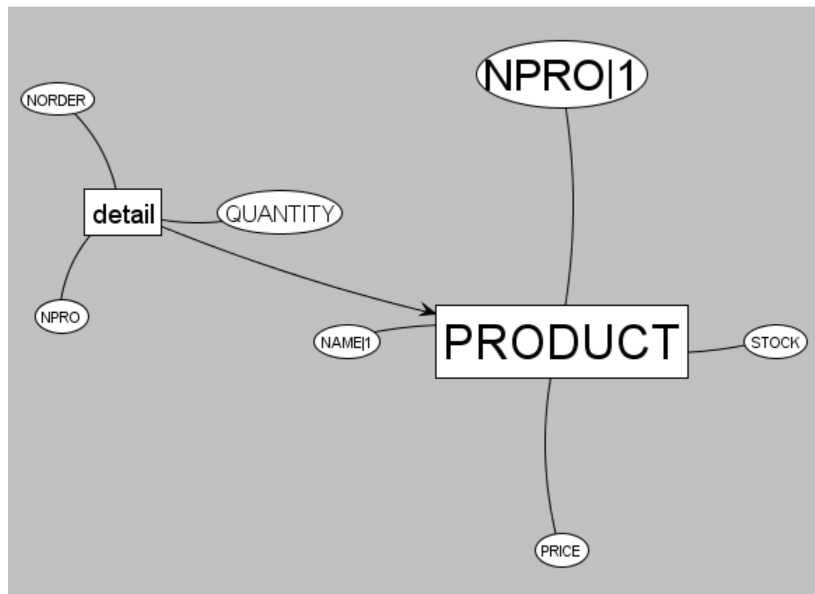

Figure 5.4: Thickness pattern

In this example, we immediatly notice the most requested logical nodes. Indeed, the tables *product*, *detail* and the column *product[npro]* have the highest frequencies.

5.2.2 Advantages of the global visualization

Detection of major graph components The global visualization allows to easily detect the main/sensitive logical nodes involved in a given set of queries. The detection of (over)sensitive nodes can indicate a system problem. For instance, if after the analysis of the queries sent by a program to a database, we notice that a table is excessively requested, this *could* be an indication of a database design problem. Let us imagine, for instance, a *God*

table, that contains hundreds of (unrelated) columns, and used by almost all programs.

Detection of implicit relationships In most legacy databases, some integrity constraints can be *implicit*: they do exist, yet they have not been explicitly declared in the DDL code. For instance, some legacy database management systems do not allow the explicit declaration of foreign keys and thus, during the database reverse-engineering process, the recovery of such constraints is required, and it is a complex problem. Nevertheless, DAViS proposes a solution to this issue. Indeed, thanks to the presence of join/conjunction relationships, it permits to facilitate the recovery of implicit foreign keys. Let us assume there is no declared foreign key between two tables A and B but, after visualizing a scenario, we detect a significant (high-frequency) join relationship between both tables, then it might indicate there exists an implicit foreign key between A and B.

Frequency evolution As we described below, DAViS implements a functionality permitting to analyse the frequency evolution of each graphical component. Since the global visualization consists of a graph-based representation, the purpose of this functionality is to study the graph evolution; indeed, according to α_i, the displayed graph owns a specific state $state_i$ and thus, one can summarize the graph evolution by Figure 5.5.

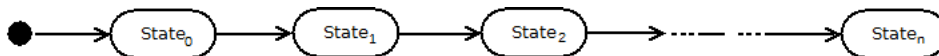

Figure 5.5: $State_i$ corresponds to the graph state at time α_i.

The global visualization allows to observe the final state but also each intermediate state in order to analyse the evolution.

5.3 Query visualization

In the previous section, we described the global visualization proposed by DAViS. It provides a support for understanding the global impacts of a program execution scenario that sends a set of SQL queries to the database. Nevertheless, we observe that the global visualization allows neither to understand the impact of a given query nor to comprehend its meaning. Therefore, DAViS proposes a second visualization method, the *query visualization*.

5.3.1 Logical approach

The purpose of the query visualization, by using the logical schema as representation model, is to visualize a given query in order to *logically* understand what this query means.
To achieve this, DAViS uses a two-phase principle (see Section 4.6) for visualizing a given query. It consists of two steps:

1. First, it visualizes **selection condition phase** of the query.

2. Second, it visualizes the query result (**output phase**).

We now show a simple example of query visualization, using the *logical* approach.
Let consider the SQL query shown at Figure 5.6.

```
(1)   select NAME, PRICE from PRODUCT
(2)   where NPRO = '1234';
```

Figure 5.6: SQL query selecting the name and price of a particular product.

This query aims to select the name and price of a particular product:

- Line 2 selects the product '1234' and thus, constitutes the selection condition phase.

- Line 1 returns the name and price of the selected product and thus, constitutes the output phase.

The *logical* query visualization of this query is illustrated in Figure 5.7, showing the two consecutive phases. It results in an animated and dynamic visualization.

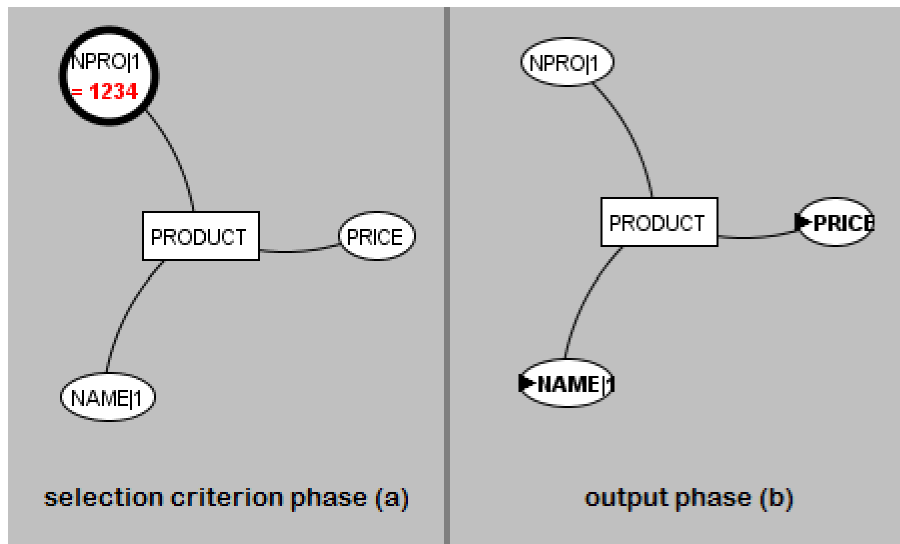

Figure 5.7: Query visualization in logical approach - the two phases

To summarize the logical approach, one can notice that the logical approach can serve as a means to obtain statistical information (*global visualization*) but also as a means to understand the meaning of a given SQL query (*the query visualization*). However, the query visualization only provides a **logical** understanding. Indeed, it uses the logical schema as representation model with its logical objects. Such a representation model is technical and low-level and does not allow the user to deeply and concretely understand what a SQL query really means in terms of application domain objects. One can conclude that the logical approach does not fully support an in-depth understanding of the data manipulation behaviour of a running program.
We can show an example illustrating the weakness of the logical approach. Let us consider the SQL query shown at Figure 5.8. This query aims to select all the customers who have ordered the product '1234'.

```
(1)  select O.NCUST
(2)  from ORDERS O, detail D
(3)  where O.NORDER = D.NORDER
(4)  and D.NPRO = '1234';
```

Figure 5.8: Example of SQL query.

Figure 5.9 represents the resulting logical query visualization. The selection condition phase shows a join (line 3) between the two columns *NORDER* (illustrated by the dotted edge) and the selection of the product '1234' (line 4). The result phase represents the selection of column *NCUST*.

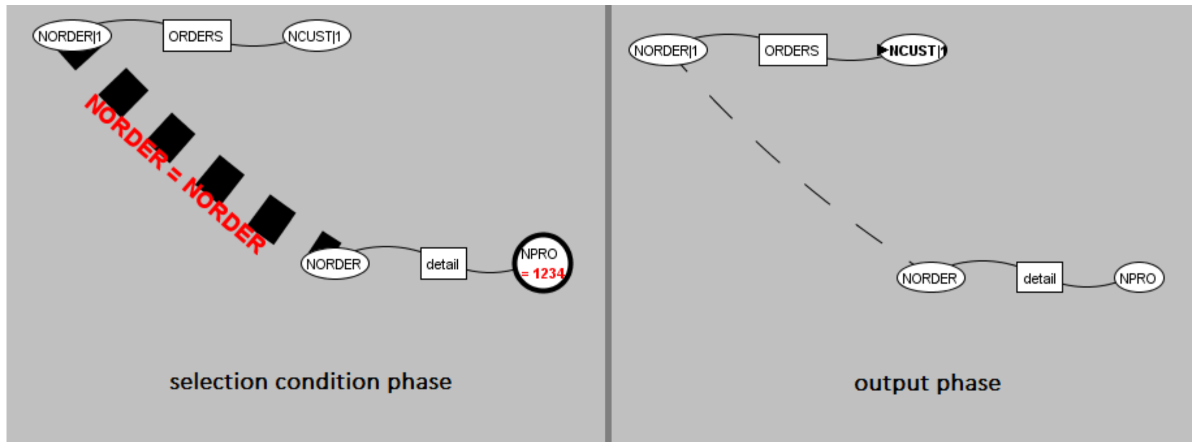

Figure 5.9: Query visualization in logical approach - the two phases

We notice that this visualization is far from intuitive: it does not offer a natural and easy way to understand the exact meaning of this query.
This is exactly why we decided to implement the conceptual approach: in order to reach a *complete* and *high-level* comprehension.

5.3.2 Conceptual approach

Since the purpose of the conceptual approach is to separately study each query of an execution scenario, it only proposes the *query visualization*, as shown in Figure 4.6. The conceptual approach uses the conceptual schema as representation model. Indeed, a conceptual schema is more abstract and it offers a representation of the database domain that is more intuitive than the logical schema.

When a database engineer wants to design a database, he or she must describe the application domain.
The latter is made of concrete and abstract entities. There exist some relationships between those entities. Figure 5.10 gives a graphical representation of an application domain.

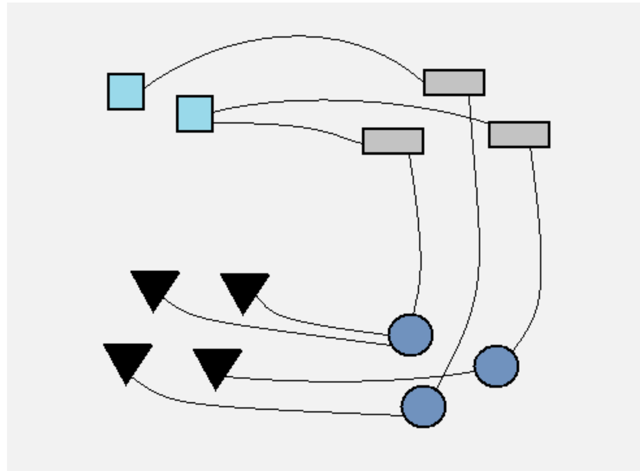

Figure 5.10: Application domain - entities and relationships

Then, the database engineer produces a conceptual schema in order to classify the entities. Each entity belongs to a class called entity type and thus, all the entities being of the same nature belongs to the same entity type.
Figure 5.11 shows the clustering of entities according to our conceptual schema described in Section 3.2.

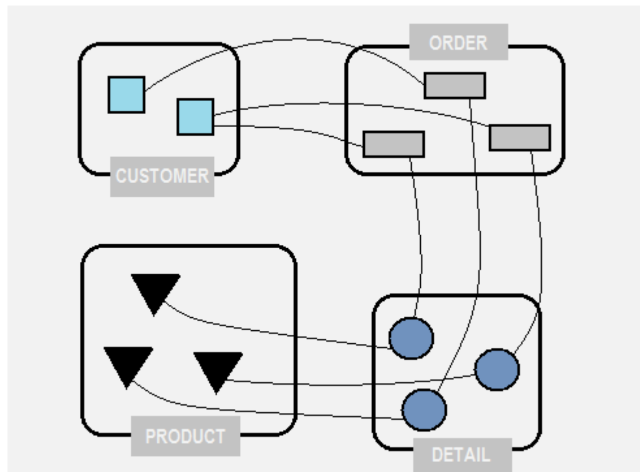

Figure 5.11: Application domain - clustering of entities of the same nature

Furthermore, a DML-SQL query can be seen as an extraction or modification of some entities of the domain. Therefore, we can consider a SQL

query as an extraction or modification of **some instances of one or several entity type(s)**.
Let us illustrate this observation by reusing our example of Figure 5.8: one observes that this query targets only one family of entities, namely the customers. However, the query does not target the whole set of customers but only the subset of customers having ordered the product '1234'.

As previously explained, the query visualization uses the two-phase principle by displaying (1) the selection condition phase and (2) the output phase. In the conceptual approach, the goal of the first phase is to graphically show the subset of entity type instances involved in the query, while the second phase allows to understand the query result.
To achieve the goal of the first phase, DAViS defines two graphical patterns.

The target pattern The *target* pattern consists in detecting the entity type(s) targeted by the query and displaying the corresponding conceptual component(s) larger than the other ones. This allows to graphically highlight the main entity type(s) involved in the query.

The subset pattern Once the entity types involved in the query are defined, the *subset* pattern consists in detecting the subset of entity type instances that is targeted by the query and in highlighting this subset by making it bold.

Let us show an example of these two patterns by reusing our example query of Figure 5.8. We illustrate the two phases of the conceptual query visualization in Figure 5.12. The entities targeted by the query are the customers and, as one can see, the entity type *CUSTOMER* is graphically larger than the other components (**target pattern**).
Moreover, the bold components (*ORDER, detail, NPRO*) denote the subset of customers involved in the query (**subset pattern**).
Without having read the query, and even without any knowledge of SQL, the user is now able to easily understand its meaning. Indeed, the selection condition phase could be translated as follows: "the visualized query targets the **customers** having **placed orders detailing** the **product** '1234'".
Concerning the output phase, it simply represents the selection of the customer's number (*NCUST*).

In conclusion, thanks to the graphical language defined by the conceptual approach, DAViS provides a support for understanding the precise meaning

of a given query in terms of application domain (conceptual) objects.

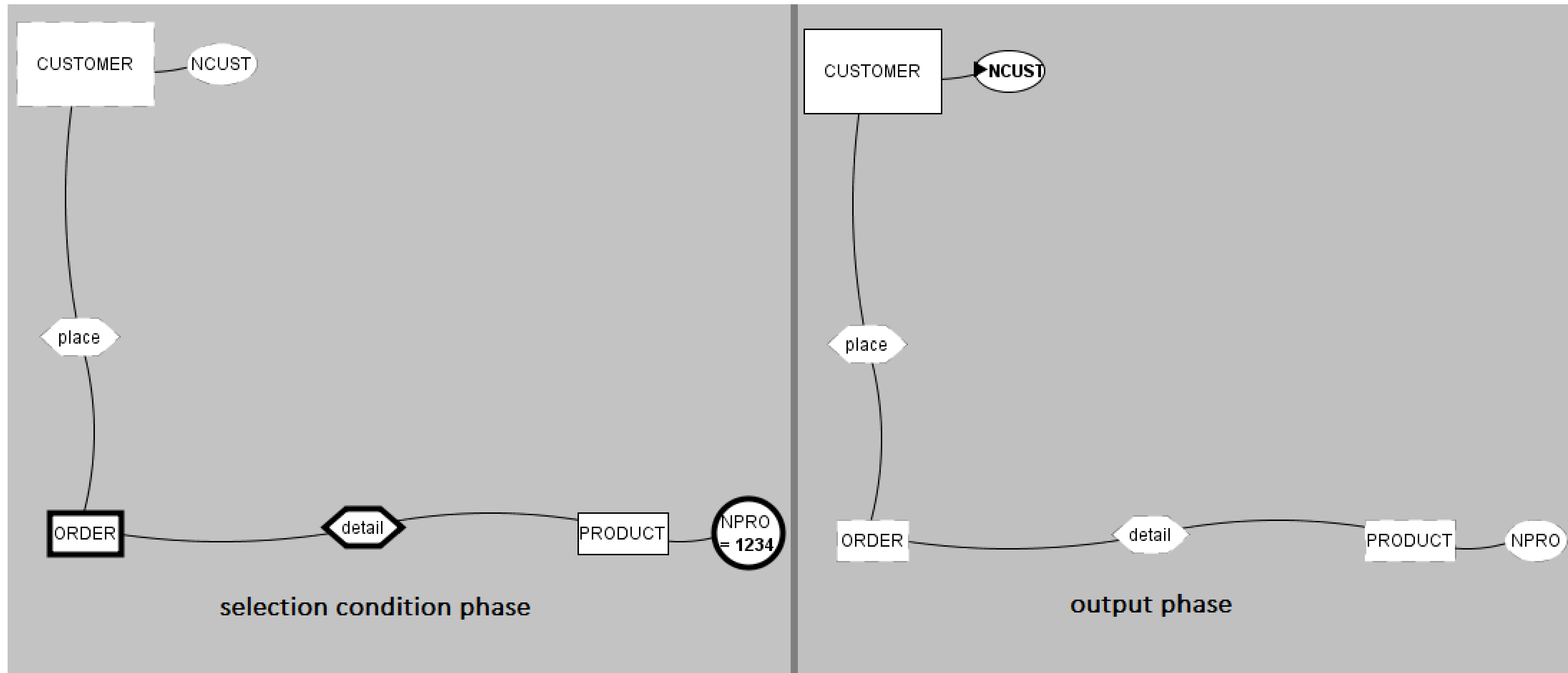

Figure 5.12: Example of SQL query - selecting customers having ordered a particular product.

Chapter 6

Implementation

In this chapter, we discuss the technological choices we made when implementing DAViS.

6.1 DB-Main

The DB-Main [11] CASE tool[1] is a tool dedicated to the database engineering domain. It constitutes a tool support for executing all the steps of the database design process (Figure 3.1: requirement analysis, logical design, physical design and coding). Moreover, DB-Main provides support for reverse engineering and program understanding.

In particular, DB-Main can be used as database modelling tool. For instance, a database can be modelled by a logical/conceptual schema written with DB-Main. Figure 6.1 shows an example of usage of DB-Main.

Schema matching As previously explained, the conceptual approach can be considered as a translation of logical objects into conceptual objects. Therefore, this process requires the database logical schema, the conceptual schema but also the mapping between both. This schema mapping aspect is discussed at Section 3.3.
DB-Main proposes an answer to this mapping problem. Indeed, it defines a *mapping meta-property* according to which each (logical or conceptual) objects has its own mapping identifier. The logical and conceptual objects having the same mapping identifier map to each other. And thus, thanks to this mapping meta-property, DB-Main permits to establish a mapping between the logical and conceptual schemas.

[1]CASE tool: Computer-Aided Software Engineering tool

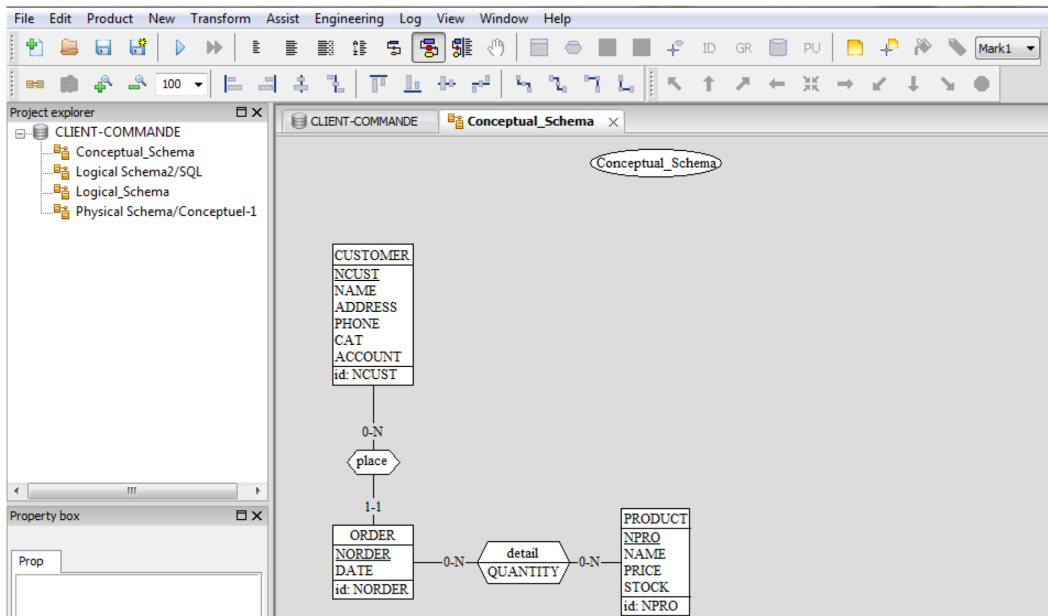

Figure 6.1: Example of our conceptual schema written with DB-Main tool

At Figure 3.1, we have established the correspondence between our logical and conceptual schemas. From now on, we are able to implement this correspondence with the mapping meta-property defined by DB-Main. Table 6.1 shows the detailed schema mapping.

	Conceptual object	Logical object	Mapping identifier
1	**CUSTOMER**	CUSTOMER	982
1.1	-NCUST	CUSTOMER(NCUST)	968
	"	ORDERS(NCUST)	968
1.2	-NAME	CUSTOMER(NAME)	964
1.3	-ADDRESS	CUSTOMER(ADDRESS)	988
1.4	-PHONE	CUSTOMER(PHONE)	948
1.5	-CAT	CUSTOMER(CAT)	986
1.6	-ACCOUNT	CUSTOMER(ACCOUNT)	976
2	**place**	ORDERS-ref:NCUST	960
3	**ORDER**	ORDERS	978
3.1	-NORDER	ORDERS(NORDER)	966
	"	detail(NORDER)	966
3.2	-DATE	ORDERS(DATE)	974
4	**detail**	detail	972
4.1	-QUANTITY	detail(QUANTITY)	952
5	**PRODUCT**	PRODUCT	954
5.1	-NPRO	detail(NPRO)	962
	"	PRODUCT(NPRO)	962
5.2	-NAME	PRODUCT(NAME)	970
5.3	-PRICE	PRODUCT(PRICE)	958
5.4	-STOCK	PRODUCT(STOCK)	950

Table 6.1: Example of mapping between the logical and conceptual schemas

This table is composed of 3 columns: the first one lists all the conceptual objects (entity type, relationship type and attribute); the second one lists all the logical objects (table, column and foreign key) that map to each conceptual object; the last one indicates the corresponding mapping identifier.

6.2 JIDBM

JIDBM[2] is an interface offered by DB-Main. It consists of a Java API allowing to access the DB-Main schemas in order to read and write them. It is composed of a set of classes written in the Java programming language. Therefore, JIDBM allows the development of Java applications (plug-ins) manipulating DB-Main schemas.

[2]Java Interface for DB-Main

We decided to use DB-Main as database schema modelling tool. DB-Main schemas will be used as inputs for our tool (see Figures 4.4 and 4.5). Thanks to this opportunity to write plugin applications, we decided to implement DAViS as a DB-Main plug-in by using the JIDBM API for accessing the schemas.
JIDBM is one of the reasons why Java was chosen for implementing the tool.

Java Java is an object-oriented programming language. The main advantage of Java is that any Java application can run on any Java Virtual Machine (JVM). JVM is an abstract layer that proposes an interface between the program and the operating system. With this particularity, a Java program is portable and can work on different operating systems (Windows, Linux, MacOS, ...).

6.3 JDBC

JDBC[3] is an API for the Java programming language that defines how to access a database. It provides methods for querying and updating data in a database. JDBC is mainly implemented for relational databases.
The class *Statement* provides two main methods allowing to query and update data :

- Querying : *Statement.executeQuery(query)*
- Updating : *Statement.executeUpdate(query)*

Figures 6.2 and 6.3 present an example of querying and updating with JDBC.

```
(1) Connection conn = DriverManager.getConnection(URL, Login, Password);
(2) String ID = "D123";
(3) String query = "select ADDRESS from CUSTOMER where NCUST = ?";
(4) PreparedStatement state = conn.prepareStatement(query);
(5) state.setString(1, ID);
(6) ResultSet result = state.executeQuery();
(7) result.next();
(8) System.out.println(result.getInt(1));
```

Figure 6.2: Example of database querying with JDBC - selecting the address of a particular customer

[3] JDBC : Java Database Connectivity

```
(1)   Connection conn = DriverManager.getConnection(URL, Login, Password);
(2)   String ID = "D123";
(3)   String query = "delete from CUSTOMER where NCUST = ?";
(4)   PreparedStatement state = conn.prepareStatement(query);
(5)   state.setString(1, ID);
(6)   state.executeUpdate();
```

Figure 6.3: Example of database updating with JDBC - customer deleting

6.4 AspectJ

As detailed in Section 4.3, the online method aims to capture each SQL query sent by the program to the database and visualize them in real time. A first technique would be to modify the program source code so that each query is captured at runtime before being sent to the database. But such a code modification may be considered as a complex and unrealistic process in most cases. In order to face this problem, we introduced in Section 1.4 the use of *aspects*. This technique allows to trace/extract each SQL query at runtime by adding some behaviour to the program without any source code modification.
This solution seems to be the most convenient technique to satisfy our needs. That is why we chose the aspect-based tracing for implementing the DAViS online method.

AspectJ [26] is an aspect-oriented extension created for the Java programming language. For implementing the online method, we decided to retain AspectJ in order to extract SQL queries produced at runtime and handle them in real time.
Furthermore, as the aspect-oriented technique does not necessitate any code modification but only a code recompilation, all you need to do is to add a piece of code to the program.
Figure 6.4 shows an example of aspect allowing to capture the SQL queries sent by the program.

```
(1)  pointcut queryExecution(String query):
(2)  call(* java.sql.Statement.execute*(String)) && args(query);
(3)  before(String query): queryExecution(query){
(4)      MyGraph graph = getGraph(); //retrieving the current visualization
graph
(5)      Request req = getRequest(query);//parsing the SQL query
(6)      graph.visRequest(req); //visualizing the parsed query
(7)    }
```

Figure 6.4: Online method - AspectJ code

At line 2, we define a pointcut every time a method *Statement.execute*(String str)* is called. This grammar covers the methods *executeQuery* and *executeUpdate* (based on JDBC syntax). At line 5, we retrieve the query to send and we parse it. At line 6, we visualize the parsed query.

6.5 SQL Parser

Both offline and online methods (illustrated by Figures 4.4 and 4.5) visualize a program execution scenario and more precisely the SQL queries involved in this scenario. Nevertheless, the captured queries have to be *parsed* before they can be visualized. In our context, *a parsed SQL query is a SQL query translated into a Java structure containing relevant information and exploitable by our visualization tool*. Thus, this chapter aims to introduce the parser integrated into DAViS. But before describing the Java structures of a parsed query, we have to list our working hypotheses regarding the SQL syntax.

6.5.1 SQL hypotheses

SQL is a high-level language designed for managing data in a relational database management systems. SQL consists of a data definition language (DDL) and a data manipulation language (DML). SQL became a standard of the International Organization for Standards (ISO) in 1987 [2]. Since then, the standard has evolved several times with added features. Up to now, there exist different revisions of SQL standard. One can directly notice SQL has a wide range and it would be very complicated to implement a visualization tool able to manage all SQL versions. This is why we had to delimit the SQL grammar subset covered by DAViS.

SQL-92 One of the main SQL versions is SQL-92 (also called *SQL2*), the third revision. As already explained in Section 4.4.1, we decided to restrict ourselves to this version. Indeed, the following revision is SQL:1999 (also called SQL3) and this version introduces a significant number of extensions, in comparison to SQL2, such as triggers, SQL procedures and some new object extensions [15]. For instance, some new complex data types appeared in SQL3 (not allowed by SQL2) such as *row* and *array* types. Moreover, SQL3 allows users to define their own data types. One can note major changes compared with SQL2 that would be manageable with difficulty by our parser. In brief, our parser only manages a subset of the SQL2 grammar.

CRUD operations In summary, SQL is a language consisting of two sub-languages: DDL allowing to define database structures and DML allowing to manipulate data. Since the DAViS purpose is to understand the impact of SQL queries, we limited SQL to the DML part and thus, to the data manipulation operators. Therefore, DAViS is limited to four types of queries: select, update, delete and insert, in other words the CRUD operations. Any other type of query will not be parsed (and thus visualized) and will be considered as an *unparsed* query.

Managed and unmanaged SQL operators It is important to notice that we present DAViS as a *prototype* visualization tool. This means that this tool does not deal with the whole SQL2 syntax. The *managed* syntax denotes what DAViS is able to parse and visualize, while the *unmanaged* syntax denotes what DAViS is able to parse BUT does not visualize.
Let us distinguish the *managed* from the *unmanaged* syntax by establishing a *non-exhaustive* list containing only the important points:

1. Where clause: select, delete and update queries may have a *where* clause defining the query selection conditions.

    a. Unary and binary operators: a *where* clause may contains several unary and binary operators. Such operators are managed by the parser. For instance, $+, -, \times, :, <, >, \leq, \geq, =, <>$, *like*, *is null*, *is not null* are managed.

    b. Composition of expressions: a *where* clause may be composed of several expressions. The compositions of expressions with the operators OR and AND are managed by the parser.
    Figure 6.5 shows an example of compositions of expressions and operators managed by the parser.

```
(1) select NPRO
(2) from PRODUCT
(3) where (NORDER = 2
(4) and QUANTITY < 3)
(5) or NAME like '%WOOD%'
```

Figure 6.5: Example of managed SQL syntax

c. Sub-request: a sub-request is an embedded request present in a *where* clause. A sub-request may, in turn, contain a sub-request. There exist two ways to build a sub-request:

  - *(NOT-)IN* clause: this kind of sub-request is managed by the parser. However, the sub-request has to be **mono-component**. Figure 6.6 presents an example of managed sub-requests, while Figure 6.7 shows an unmanaged case.

```
(1) select *
(2) from PRODUCT
(3) where NPRO IN (select D.NPRO
(4)                from detail D)
(5) and NAME NOT IN ("PRO1","PRO2")
```

Figure 6.6: Sub-request (*IN* clause) - Example of managed sub-request: line 3 and line 5 are both managed sub-requests

```
(1) select *
(2) from PRODUCT
(3) where (NAME,STOCK) IN ( ("PRO1",20), ("PRO2",15))
```

Figure 6.7: Sub-request (*IN* clause) - Example of unmanaged sub-request because element (NAME,STOCK) is not mono-component

  - *(NOT-)EXISTS* clause: this kind of sub-request is not managed by the parser. We present an example of unmanaged *EXISTS* clause at Figure 6.8.

```
(1) select *
(2) from PRODUCT
(3) where EXISTS (select * from detail D
(4)                        where D.NPRO = NPRO)
```

Figure 6.8: Sub-request ($EXISTS$ clause) - Example

2. Set operators: the set operators (*except, intersect union, union all, any, for all*, ...) are not managed by the parser.
3. Aggregative functions: no aggregative function is managed by the parser. For instance: *count(), sum(), max(), min()*, ...
4. Other pre-existing SQL functions: some pre-existing SQL functions are not managed by the parser. Example: *substring(), trim()*, ...
5. Explicit join operators: join operators having the form

   join [tables] **on** [join_conidition]

   are managed by the parser. The set of managed join operators includes [*right/left*][*inner/outer*] joins.

```
(1) select *
(2) from PRODUCT P right outer join detail D
(3) on (P.NPRO = D.NPRO)
```

Example of explicit join operator

6. Other operators: some other operators are not managed by the parser like *distinct, order by, group by, having*, ...

6.5.2 Parsed query structures

Given all our hypotheses about SQL parsing, a customized parser is required. That is why we decided to build our own SQL parser using a parsing Java API. The chosen Java API is **JSqlParser** [16]. JSqlParser parses a SQL query and translates it into a hierarchy of Java classes. The generated hierarchy can then be navigated. Thus, our customized parser can be seen as a tool extracting pertinent data from the hierarchy obtained by JSqlParser. Then, once extracted, data are stored into predefined Java structures. This

process is summarized by Figure 6.9. However, the DAViS users can freely bring their own SQL parser and replace the current parser with another one. The only constraint is to respect the predefined Java structures into which parsed data will be stored.

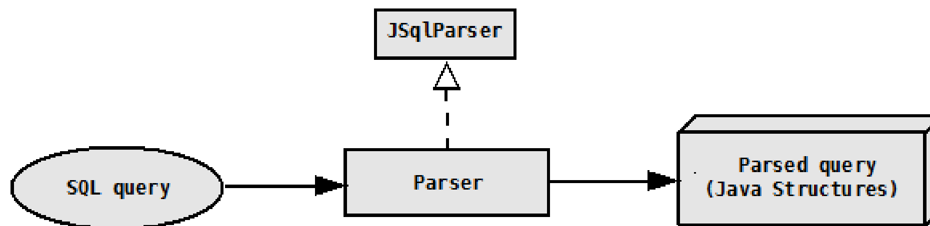

Figure 6.9: Parsing process

In this section, we introduce an overview of the Java structures into which the parsed query is stored. This object represents a recursive structure and is implemented as a *Java Arraylist*. To avoid a technical description of the Java structures, we decided to design a model describing the Java object (see Figure 6.10).
Our customized parser and its Java structures have mainly been thought and designed by Nesrine Noughi, teaching assistant and PhD student at University of Namur[4].

Parsed_query It represents the parsed query. There exist three main types of query: an *Invalid_query*, an *Insert_query* and a *Where_clause_query*.

Invalid_query It represents an invalid query. An invalid query can appear in two cases:

1. Unparsed query: the parser cannot parse the query either if there is a syntax error or if the query is not a CRUD operation.

2. No relevant information: an invalid query may be a query without relevant information. For instance, if the query contains only columns and tables that are not present in the logical schema.

Insert_query It represents an inserting query. An inserting query consists in putting new values (*Inserted_values*) in a table (*table_name*). It inserts a new value (*new_value*) for each column (*column*).

[4]http://www.unamur.be/universite/personnes/page_view/01008459/

Where_clause_query Deleting query (*Delete_query*), updating query (*Update_query*) and selecting query (*Selecting_query*) belong to this category, namely every query type that **might** have a *where* clause (*Where_clause*).

Delete_query It represents a deleting query. It consists in removing some lines of a table (*table_name*).

Update_query It represents an updating query. It consists in updating some columns (*Updated_values*) of a table (*table_name*). Each updated column (*column*) receives a new value (*new_value*).

Select_query It represents a selecting query. This query type consists in extracting some columns (*Selected_column*). A selecting query also has a *from* clause with tables (*From_table*). Moreover, this query type may have an explicit join operator (*Join*). A join between two tables (*table1* and *table2*) may have a special type (right/left inner/outer) and owns a join condition (*join_condition*) on two columns.

Where_clause It represents a where clause. A where clause may contain several expressions with unary/binary operators (*Expression*) and may have several sub-request.

Subrequest It represents a mono-component sub-request. A sub-request has a type (*IN* or *NOT IN* type) and is based on the matching between a source column (*source_column*) and a set of values. This set of values is extracted from a selecting query (*Select_query*) containing only one selected column (in order to satisfy the condition of **mono-component** sub-request). We notice that a sub-request has a recursive definition.

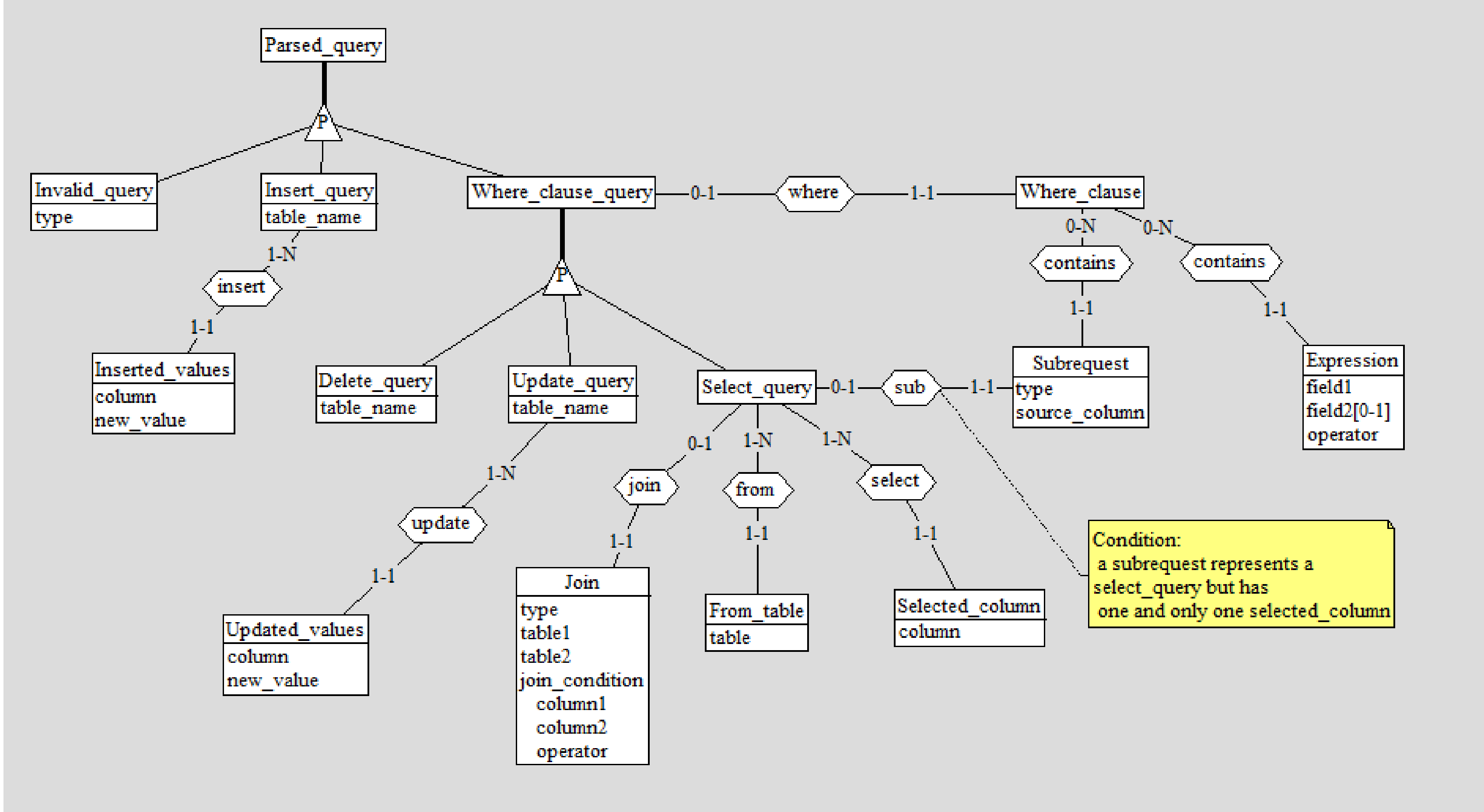

Figure 6.10: Java structures of a parsed query - modelling

6.6 JUNG

JUNG[5] is a Java API that provides a common and extensible language for the visualization of data that can be represented as a graph or network [17]. This API is designed to support a variety of representations of entities and their relationships, such as directed and undirected graphs. JUNG facilitates the creation of tools allowing to visualize complex data sets and easily explore data networks .
Furthermore, users can easily use one of the predefined layout algorithms, or use their own custom layouts.

We decided to choose JUNG as visualization API. Thanks to it, we are able to implement a graph-based representation and dynamically visualize complex data sets.

[5] Java Universal Network/Graph

Chapter 7

Evaluation

In this chapter, we present the global evaluation of our DAViS visualization tool, as well as its usage possibilities.

7.1 Case studies

During the implementation phase of DAViS, we had the opportunity to evaluate our tool by studying two concrete cases.

7.1.1 Webcampus

In order to evaluate our approach, we experimented the case of Webcampus [21], an e-learning application used at the University of Namur. Webcampus is a platform allowing teachers to offer learning stuff on the web. Webcampus is written in PHP and uses a MySQL database. The database contains 33 tables, 198 columns and 37 foreign keys. Thanks to our collaboration with Jean-Roch Meurisse, the administrator of Webcampus, the database logical schema as well as the SQL traces generated by 14 execution scenarios were available to us. These 14 scenarios denotes the most typical operations performed by Webcampus users. For each scenario, the generated SQL traces are stored into a file. We had a total of about 10,000 SQL queries to visualize with DAViS.
By studying this case study and visualizing SQL traces generated by real execution scenarios, we corrected and improved our visualization tool.
Figure 7.1 shows the overview of the Webcampus logical schema.

Figure 7.1: Logical schema of Webcampus's database

Table 7.1 presents some statistics pertaining to the collected queries. The 14 scenarios contains 10,192 queries. However, 5,026 queries are *out-of-schema* (no logical object is in the logical schema) and 157 cannot be parsed. Therefore there are only 5,009 queries the user can visualize:

- 4,828 selecting queries
- 29 deleting queries
- 3 updating queries
- 149 inserting queries

We observe DAViS is able to parse 98 percent of the queries. The majority of the unparsed queries is due to either the presence of SQL functions our tool cannot parse (e.g. *ISNULL()*, *IFNULL()*, *FOUND-ROWS()*, ...) or a non-CRUD operation (e.g. a *creating* query).

total of queries	**10192**
unparsed queries	**157**
out-of-schema queries	5026
selecting queries	4828
deleting queries	29
updating queries	3
inserting queries	149

Table 7.1: Webcampus - Overview

7.1.2 Rever

Rever [19] is spin-off company of the database engineering lab of the University of Namur. Rever brings model driven data engineering solutions to companies. In particular, the DB-Main CASE tool has been developed and is still currently maintained by Rever. We had the opportunity to collaborate with Rever. Rever is interested in integrating DAViS in a future project. A senior consultant of the company has tested the tool on a logical schema of about hundred tables, and with a SQL execution trace 1,584 queries. Thanks to this collaboration, we detected some technical problems and added a few features recommended by Rever.

Table 7.2 presents statistics about the collected queries. We observe 76 percent of the queries are successfully parsed. The majority of unparsed queries is due to a SQL syntax not considered by our tool. Figure 7.2 shows an example of SQL query not covered by the parser.

```
WHEN T.TYPE='X' THEN NULL WHEN T.TYPE='N'
THEN 'SYNONYM' ELSE TT.TABLE_TYPE END
IN ('TABLE') ORDER BY 4,1,2,3
```

Figure 7.2: SQL query - WHEN/THEN/ELSE

total of queries	1584
errors	376
out-of-schema queries	1042
selecting queries	151
deleting queries	1
updating queries	12
inserting queries	2

Table 7.2: Rever - Overview

7.2 Forces and limitations

Through this thesis, DAViS has been presented to the reader. In this section, we proposes a synthetic view of the forces and limitations present in the current version of this tool.

7.2.1 Forces

Through the different chapters, we have seen that DAViS supports two main approaches: the logical approach allowing to obtain a **logical** understanding of an execution scenario and the conceptual approach providing a **conceptual** comprehension.
The former uses the database logical schema as representation model for the visualization and aims to extract technical information pertaining to the visualized scenario:

1. Detection of major logical objects: DAViS is a support for detecting the most requested logical nodes (tables and columns), each of them with some statistical information such as their usage frequency in the scenario.

2. Detection of implicit relationships: as explained, most legacy databases cannot explicitly declare some integrity constraints; indeed, it is not rare to encounter a database without explicit foreign key but implicitly existing. The recovery of implicit foreign keys is a complex process. DAViS allows to detect the **join** and **conjunction** relationships between tables and thus, offers a support for the detection of implicit foreign keys.

3. Evolution analysis: our visualization tool has a functionality that proposes users to analyse the evolution of the displayed graph. It allows to

study the evolution of the usage frequency of nodes and relationships during a scenario.

The conceptual approach uses the database conceptual schema as representation model for the visualization and offers a support for **conceptually** understanding the meaning of a given query. The conceptual approach proposes a graphical language for visualizing and comprehending queries.

7.2.2 Limitations

We are aware that DAViS is only a first prototype. This is why we present here a list of the limitations of the current version as well as possible improvements for future versions:

- SQL limitations: since there exist different SQL versions, we made the choice to limit our parser to SQL-92. Moreover, we use a customized parser that does not cover all the SQL-92 syntax. In Section 6.5, we define the subset of SQL-92 managed by our parser. A future version would consist in covering the whole SQL-92 and thus, adding new visualization patterns.

- Conceptual schema: in Section 3.2, we made some assumptions pertaining to the subset of the ER model considered by our tool. We showed that the conceptual expressiveness was not impacted by those assumptions but however, it might be interesting to permit other conceptual constructions in the conceptual approach, such as an *is-a* relationships, ternary relationship types and compound attributes.

- Memory: as explained in Section 5.2.2, the global visualization allows to observe each intermediate graph state. Therefore, we store each intermediate state in memory. Nevertheless, when the graph has very high number of nodes and edges, DAViS is likely to be memory-consuming.

- Slowness: DAViS may encounter problems of slowness when the displayed graph is composed of a very high number of nodes. Indeed, the displaying and the positioning of each node is calculated whenever the graph is updated. Thus, the graph updating may be a slow process depending on the number of nodes.

7.3 Comparison with the survey

Let us remind that we introduced in Section 2.1 a survey established by Delft University summarizing the existing literature about the use of dynamic

analysis in the context of program comprehension. This study defines a list of facets in order to classify all the articles: the activity, the target, the method and the evaluation. Since we have now presented DAViS in detail, we can now apply the same evaluation grid (the four facets) on our tool. Table 7.3 shows the grid applied on DAViS.

	Facet	**Attribute**
1	Activity	Views Trace analysis Behaviour
2	Target	All systems
3	Method	Vis. (std.) Online

Table 7.3: Comparison with survey

Let us analyse the comparison:

1. Activity:
    - Views: we saw that DAViS proposes the reconstruction of specific views about the database (logical and conceptual schemas as representation model).
    - Behaviour: DAViS aims at analysing the communication between a program and its database, i.e., the data manipulation behaviour of the program.
    - Trace analysis: the tool mainly cares about the trace analysis by understanding the SQL traces generated at runtime by the program.
2. Target: we observe DAViS is able to target all kinds of data-intensive systems generating SQL traces.
3. Method:
    - Visualization: we propose a dynamic visualization of the interactions between a program and its database through a graph-based representation.
    - Online: in Section 4, we define two contexts targeted by DAViS, namely the offline and online methods. We saw that the latter allows to visualize the SQL queries in real time.

In Section 4, the graph (Figure 2.1) synthesizes results obtained by the survey.
The graph shows the **trace analysis** is poorly exploited by the research community. But DAViS tries to cover this point by analysing the communication between the program and the database (SQL queries analysis) in order to ease program understanding.
About the targeted systems, this tool is able to deal with any system that communicates with a database by producing SQL queries. Thus, DAViS is suitable for all data-intensive systems and in particular, for **legacy systems** and **web applications**, rarely targeted.
Concerning the used methods, our tool provides both an *offline* and *online* analysis of the communication between the program and the database. The survey shows the online analysis is the least used method.
To conclude this comparison, we introduce in this thesis, a novel tool exploring some yet almost unexplored areas like trace analysis applied to legacy systems and web applications (more generally on data-intensive systems) and allowing to analyse SQL traces generated at runtime. Thus, this tool covers a few unexplored areas. This is why we can consider that this thesis proposes an original way to ease program maintenance/evolution by supporting data-intensive program comprehension.

7.4 Usage possibilities

We now discuss the usage possibilities of DAViS:

1. Program understanding: in the introduction, we highlight the high costs related to software systems evolution. Indeed, our main purpose is to reduce those costs by providing support to data-intensive program comprehension. The first usage possibility offered by DAViS is to support the understanding of the database manipulation behaviour of a program.

2. Program debugging: DAViS can also serve as a program debugger. It would indeed allow to check the program correctness by analysing its execution, and in particular the behaviour of the SQL queries sent by the program.

3. Database monitoring: DAViS might allow a real-time monitoring of a database. For instance, it could control online the impact of the queries received by the database, the frequency of access to each table, etc. Such a tool could be useful, for instance, for database administrators.

4. Logical reconstruction: in Section 3.5, we introduced database reverse engineering. A particular phase consists in reconstructing the logical schema from the database source code and other artefacts. Such an activity is far from being straightforward. Thanks to its logical approach, DAViS also constitutes a support for recovering some implicit integrity constraints and thus, for refining the logical schema (e.g., the recovery of implicit foreign keys).

Chapter 8

Conclusion

8.1 Summary and contributions

In Chapter 1, we introduced the purpose of this thesis; we saw that software maintenance and evolution constitute a time-consuming and expensive processes. In particular, we observed that program comprehension represented more than 50 percent of the total costs of maintenance. Program comprehension is a complex activity, but we noticed that the analysis of the communication between the program and the database may be a good angle for understanding what a data-intensive program is doing. In the case of a relational database, this communication is materialized by the SQL queries sent by the program. However, we saw that more and more data-intensive programs use dynamic SQL statements, i.e., queries that are build at run-time.

Therefore, the aim of this thesis was to propose a novel tool-supported approach consisting in *dynamically* analysing the SQL queries sent by the program in order to ease program understanding, and transitively, to ease software system evolution.

In Chapter 2, we summarized the state of the art related to program comprehension through dynamic analysis. We showed that almost none of the existing works focus on the analysis of the program-database communication.

Chapter 3 gave an overview of database engineering, that constitutes the conceptual background of our work. We described all the phases for designing a database. During the database engineering process, we saw that different database schemas are produced: in particular, the logical and conceptual schemas.

Chapter 4 introduced our tool, DAViS, to the reader by presenting its theoretical principles:

- Dual context: DAViS can be used both in *offline* or *online* usage scenarios. The former allows an offline visualization of an execution scenario: firstly, the user collects SQL queries generated during program execution and secondly, visualize them with DAViS. The latter consists in collecting and visualizing SQL queries in real time, during program execution.
- Logical and conceptual approaches: DAViS offers a graph-based visualization and proposes two possible representation models: either the logical schema or the conceptual schema. The former allows to understand the technical impact of the SQL queries on the database while the latter permits a more abstract comprehension.
- The global and query visualizations: DAViS provides the user with the opportunity to visualize the global impact of a set of queries on the database or to analyse a particular query. The global visualization allows to obtain statistical information whereas the query visualization supports the comprehension of one given query. The combination of both visualizations is required to obtain an in-depth understanding of the data manipulation behaviour of a given program in a given execution scenario.

Chapters 5 and 6 gave a technical description of our visualization tool as well as an illustrating example of its usage.
Finally, Chapter 7 presented the global evaluation of DAViS; we used the tool in two practical cases (Webcampus and Rever) and obtained promising results. However, it is important to notice that our visualization tool still has to be formally evaluated through a more systematic experiment involving real users, in order to measure to what extent it can actually ease data-intensive program comprehension.

In conclusion, we presented, in this Master's thesis, a novel tool-supported approach supporting program comprehension in general, and in particular, the comprehension of the communication between application programs and their database, with the general objective to reduce the costs of data-intensive software evolution.

8.2 Future directions

Obviously, this Master's thesis only constitutes as an humble starting point for further research developments. We anticipate two possible future directions in this domain:

- Paraphrasing and automatic redocumentation: with the conceptual approach, DAViS proposes a support for *conceptually* understanding the precise meaning of a given query. Therefore, we could easily imagine to add a functionality making query paraphrasing possible; the query paraphrasing would consist in generating a sentence expressing the meaning of the query in natural language. Thus, we could imagine an automated program redocumentation process. Figure 8.1 represents an example of a Java method whose the documentation is missing and Figure 8.2 shows the method after automatic redocumentation. Alternatively, such a paraphrasing could also be used on-the-fly, to help developers and users to easily understand what a program is doing on the database at runtime in certain execution conditions. The application of such techniques in the context of debugging or customer acceptance tests look promising.

```
public void doIt(int Num){

Connection conn = DriverManager.getConnection(url,login,pwd);
String query = "select O.NCUST"+
               "from ORDERS O, detail D"+
               "where O.NORDER = D.NORDER"+
               "and D.NPRO = ?;";

PreparedStatement state = conn.prepareStatement(query);
state.setString(1, Num);
ResultSet result = state.executeQuery();

               ....

               }
```

Figure 8.1: Example of undocumented code

```
public void doIt(int Num){

Connection conn = DriverManager.getConnection(url,login,pwd);
String query = "select O.NCUST"+
               "from ORDERS O, detail D"+
               "where O.NORDER = D.NORDER"+
               "and D.NPRO = ?;";

PreparedStatement state = conn.prepareStatement(query);
state.setString(1, Num);
/**
* Selecting the customers having ordered a particular product.
*/
ResultSet result = state.executeQuery();

               ....

               }
```

Figure 8.2: Example of redocumented code

- Program workflow extraction: an enhancement of our dynamic analysis approach would consist of a new feature allowing to extract and visualize the behaviour of a data-intensive system from *several* SQL execution traces that all correspond to the same *business process*. The idea, currently under investigation, would be to extract a *workflow* describing the successive data manipulation events, possibly with sequential, iterative and choice operators. Such a feature would be a good support for understanding and redocumenting the complete data-manipulation behaviour of a data-intensive program, not only of one of its execution scenario as we do in this thesis.

Appendices

Appendix A

Additional functionalities

DAViS proposes some additional functionalities. Figure A.1 shows the graphical user interface of DAViS.

1. **Logical schema**: allowing to reach the logical approach. The logical schema is used as representation model.

2. **Conceptual schema**: allowing to reach the conceptual approach. The conceptual schema is used as representation model. This feature is optional and depends on the availability of the conceptual schema.

3. **Main view**: representing the graph-based visualization.

4. **Main view panel**: this panel permits to filter information the user does not wish to visualize in the main view such as the foreign keys, the attributes/columns, the conjunction and join relationships.

5. **Satellite view**: representing a simplified view allowing the user to obtain an overview of the visualization.

6. **Satellite view panel**: this panel permits to filter information the user does not wish to visualize in the satellite view. The main and satellite views can be complementary: the users may display the most important information on the main view while the satellite view displays secondary information.

7. **List of queries**: this panel contains all the SQL queries to visualize. The user can easily navigate through the queries.

8. **Query description**: panel describing the current query.

9. **Graph evolution**: feature allowing to analyse the *frequency evolution* by successively displaying each intermediate graph (see Section 5.2.2).

10. **Visualizing one query**: feature permitting to visualize the current query in order to understand its impacts on the database (see Section 5.3).

11. **Previous query**: selecting the previous query.

12. **Next query**: selecting the next query.

13. **Showing unparsed query**: the user can extract the details of each query the tool cannot parse.

14. **Exporting graph PNG**: allowing to export the current displayed graph as an image.

15. **Injection into DB-Main**: the user can inject the results of the visualization into the source database schema.

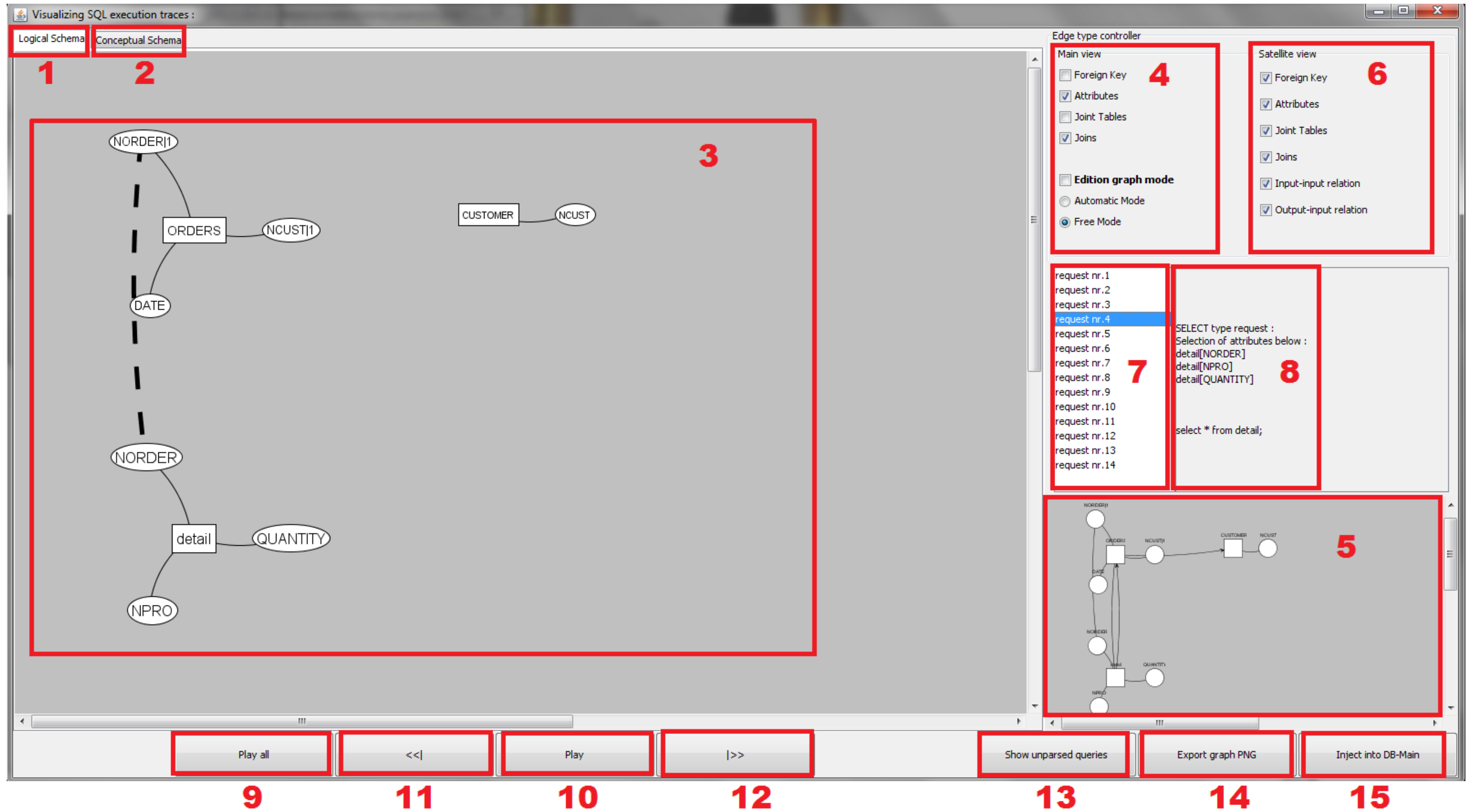

Figure A.1: DAViS interface

Appendix B

Offline and online methods

Figure B.1 represents the execution of the offline method.

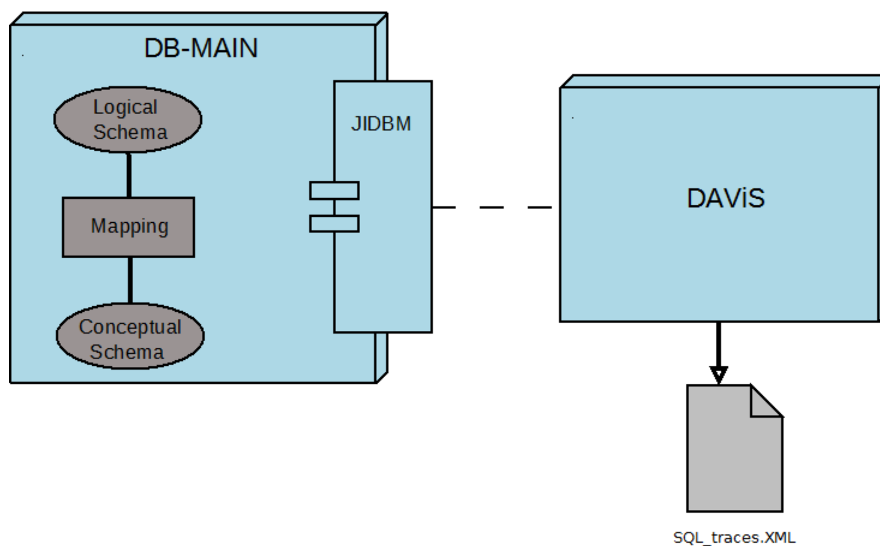

Figure B.1: Offline method - Components

DAViS takes an XML file as input. This file contains the SQL queries to visualize. DAViS accesses to the database logical (mandatory) and conceptual (optional) schemas thanks to the JIDBM interface (see Section 6.2).

Figure B.2 represents the execution of the online method.

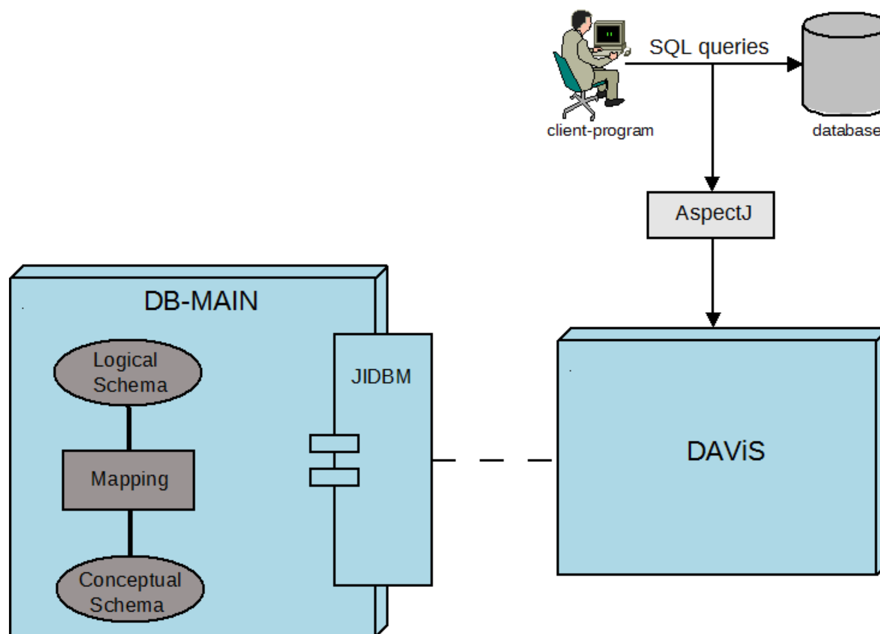

Figure B.2: Online method - Components

DAViS does not take an XML file anymore as input but the queries generated at runtime. These queries are captured thanks to AspectJ (Section 6.4). DAViS accesses to the database logical (mandatory) and conceptual (optional) schemas thanks to the JIDBM interface.

Bibliography

[1] Software Engineering - Software Life Cycle Processes. ISO/IEC 14764:2006.

[2] Information Technology - Database Languages - SQL. ISO/IEC 9075-1, 1987.

[3] Ieee standard for software maintenance. *IEEE Std 1219-1993*, pages 1–45, 1993.

[4] M. Alalfi, J.R. Cordy, , and T.R. Dean. WAFA: Fine-grained dynamic analysis of web applications. In *Proc. of the 11th International Symposium on Web Systems Evolution (WSE'2009)*, pages 41–50. IEEE Computer Society, 2009.

[5] Barry William Boehm. A spiral model of software development and enhancement. In *ACM SIGSOFT Software Engineering Notes*, volume 11, pages 14–24, 1986.

[6] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[7] Anthony Cleve, Jean-Roch Meurisse, and Jean-Luc Hainaut. Database semantics recovery through analysis of dynamic SQL statements. *Journal on Data Semantics*, 15:130–157, 2011.

[8] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. Dynamic program analysis for database reverse engineering. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 7680 of *Lecture Notes in Computer Science*, pages 297–321. Springer, 2013.

[9] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering (TSE)*, 35(5):684–702, 2009.

[10] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 49–58, Washington, DC, USA, 2007. IEEE Computer Society.

[11] DB-MAIN. The DB-MAIN official website. http://www.db-main.be, 2006.

[12] Concettina Del Grosso, Massimiliano Di Penta, and Ignacio García Rodríguez de Guzmán. An approach for mining services in database oriented applications. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 287–296. IEEE Computer Society, 2007.

[13] Robert L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, May 2001.

[14] Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, CSMR '03, pages 259–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] J-L H. Hainaut. *Bases de données - Concepts, utilisation et développement*. Dunod, 1st edition, 2009.

[16] JSqlParser. http://jsqlparser.sourceforge.net/.

[17] JUNG. http://jung.sourceforge.net/, 2003.

[18] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[19] Rever. http://www.rever-sa.com/.

[20] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings ICSM'99 (International Conference on Software Maintenance*, pages 13–22. IEEE, 1999.

[21] University of Namur. http://webcampus.fundp.ac.be/, 2001.

[22] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 271–283, New York, NY, USA, 1998. ACM.

[23] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In Jonathan I. Maletic, Alexandru Telea, and Andrian Marcus, editors, *VISSOFT*, pages 92–99. IEEE Computer Society, 2007.

[24] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 921–922, New York, NY, USA, 2008. ACM.

[25] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM.

[26] Xerox PARC. AspectJ. http://www.eclipse.org/aspectj/, 2001.