THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Innovative Techniques and Tools for Database Reverse Engineering in Large Data Intensive Systems

Gobert, Maxime; Maes, Jerome

Award date:
2013

Awarding institution:
University of Namur

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

Innovative Techniques and Tools for Database Reverse Engineering in Large Data Intensive Systems

Maxime Gobert Jérôme Maes

Maître de stage : Jens Weber

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Anthony Cleve

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

Acknowledgements

This work is the result of a three-month internship at University of Victoria in Canada, British Columbia where we were integrated in a research project aiming to build a *Primary Care Research Network* (PCRN) integrating several *Electronic Medical Record* (EMR) software systems. The PCRN will integrate information kept in the EMR systems in order to make them accessible to medical research and data mining. We worked on one of those EMR called OSCAR for *Open Source Clinical Application Resource* on which we contributed to a database reverse engineering process.
Parts of the three months of work resulted in the publication of a paper for the 29th IEEE International Conference of Software Maintenance [1]. The paper was accepted as a short paper and was entitled : *Understanding Schema Evolution as a Basis for Database Reengineering* [15]. An extended version of this work has also been accepted for publication in Science of Computer Programming and will be published in 2014 under the title *Understanding Database Schema Evolution: A Case Study* [6].

We would like to thank our supervisors Anthony Cleve and Jens Weber for their support, their advices, their enthusiasm and their sympathy, which made this internship and thesis an inspiring and interesting moment. With their conversation full of ideas they succeeded to pass on their passion about databases. We also thank Jeremy Ho for his technical help and contribution in reflection moments.

[1] http://icsm2013.tue.nl/

Abstract

Software evolution is a standalone and complex part of Software engineering. This topic of research has been explored since the eighties with a starting roadmap by Lehman. The importance of this activity has increased in the recent years due to ever-changing user needs and target environments. Before starting to maintain or evolve a software system it is crucial to understand it. Program comprehension, due to the raising complexity of software, constitutes one of the most time-consuming and costly processes. Several methods and techniques exist to support this process. As most actual systems use data, we also have to evolve and understand the database. The database reverse engineering process aims to help recover the database semantics and documentation. But applied to large data systems the current methods may be limited.
This thesis aims to make database reverse engineering easier in presence of large data intensive systems. We make use of techniques such as program analysis, execution trace analysis, schema analysis, data analysis and software repository mining. We then report on the application of the presented techniques and tools to a large case study.

Contents

1 Introduction & Motivation **9**
- 1.1 System evolution . . . 9
- 1.2 Program comprehension & Reverse engineering . . . 10
- 1.3 Database reverse engineering . . . 10
- 1.4 Research Questions . . . 11
- 1.5 Thesis Limit . . . 11
- 1.6 Thesis Structure . . . 11

2 State of the art **13**
- 2.1 Database engineering . . . 13
- 2.2 Database reverse engineering . . . 15
- 2.3 Schema evolution . . . 18
- 2.4 Clustering & Filtering . . . 19
- 2.5 Program and Database Slicing . . . 20

3 Methodology **23**
- 3.1 A revisited database reverse methodology . . . 23
- 3.2 A modified schema refinement . . . 25
- 3.3 Understanding the schema . . . 27

4 Design **31**
- 4.1 Inferring Foreign Key . . . 31
  - 4.1.1 Where to find the foreign keys? . . . 31
  - 4.1.2 Name analysis . . . 32
  - 4.1.3 Mapping files . . . 34
- 4.2 Clustering & Summarization . . . 34
  - 4.2.1 Cluster analysing . . . 34
  - 4.2.2 Schema summarization . . . 37
- 4.3 Smart schema filtering . . . 40
- 4.4 Historical schema analysis . . . 43
  - 4.4.1 Motivations & Concepts . . . 43
  - 4.4.2 Methodology . . . 43
  - 4.4.3 Specification . . . 44

4.4.4 Algorithm . . . 45
4.5 Database slicing . . . 47
4.5.1 Downward slicing . . . 47
4.5.2 Upward slicing . . . 49

5 Implementation **51**
5.1 DBMain & JIDBM . . . 51
5.2 Foreign key generator . . . 51
5.2.1 The Mapping File Parser . . . 51
5.2.2 Output files . . . 52
5.3 Historical Schema . . . 53
5.4 Filtering Tool . . . 54
5.5 Database Slicing . . . 55
5.5.1 Downward . . . 55
5.5.2 Upward . . . 57
5.6 Utility Tools . . . 57
5.6.1 Git Extractor . . . 57
5.6.2 SQL Transformation . . . 58
5.6.3 Encryption tool . . . 58

6 Case study : OSCAR **61**
6.1 Context: The OSCAR System . . . 61
6.1.1 OSCAR's Architecture . . . 62
6.2 Results . . . 62
6.2.1 Foreign Key Extraction . . . 62
6.2.2 Historical Schema . . . 65
6.2.3 Clustering . . . 71
6.2.4 Filtering . . . 72
6.2.5 Database Slicing . . . 74

7 Additional Discussion **79**

8 Conclusion **81**

9 Future works & Applications **85**
9.1 Advanced Exploitation of Historical Schema . . . 85
9.2 Enriched Database Slicing . . . 86
9.3 Extended clustering and Filtering . . . 86

A MediaWiki Historical Schema and statistics **95**

List of Figures

2.1 Building process of a relational database 13
2.2 Detailed methodology of database reverse engineering (taken from [16]) . 16
2.3 Detail of the Schema Refinement process (taken from [16]). 18

3.1 A revisited methodology of database reverse engineering (inspired from [16]) 24
3.2 The modified Schema Refinement process (inspired from [16]) 26
3.3 Schema Refinement : inferring foreign key process 27
3.4 Schema Understanding : composition 28

4.1 Dendogram of an example of hierarchical clustering 35
4.2 Example of summarization 38
4.3 Global process 44
4.4 Schema evolution example 44
4.5 Global historical schema obtained from schema evolution of Figure 4.4 . . 45
4.6 DBSlicing - Downward : General scenario 48

5.1 Thread Implementation 53
5.2 Example of records in a table 59
5.3 Encrypted records of figure A.1 60

6.1 The OSCAR schema, as it can be viewed in DB-MAIN just after the SQL extraction 63
6.2 The property box of a foreign key 63
6.3 The OSCAR global historical schema, as it can be viewed in DB-MAIN. . 66
6.4 The OSCAR global historical schema, when zooming on a particular table. 67
6.5 Evolution of the number of tables in the OSCAR database. 68
6.6 Evolution of the number of columns in the OSCAR database. 68
6.7 Evolution of the number of creation and deletion of tables per version in the OSCAR database. 69
6.8 Evolution of the number of columns in the OSCAR database. 69
6.9 Classification of the OSCAR tables in terms of age VS size. 70
6.10 Classification of the OSCAR tables in terms of age VS number of changes. 70
6.11 Sample of the summarized schema 71

6.12 Filtering of caisi_editor and system_message 73
6.13 Log result of the upward slicing 74
6.14 Screenshot of the form to add an appointment 75
6.15 Screenshot of the subschema impacted by "adding an appointment" . . . 76

A.1 Historical schema of MediaWiki 96
A.2 Column modification of MediaWiki per version 97
A.3 Table modification of MediaWiki per version 97
A.4 Table age vs table size (MediaWiki) 98
A.5 Table stability (MediaWiki) 98

Chapter 1

Introduction & Motivation

1.1 System evolution

Nowadays, information systems are becoming more and more complex and they are of growing importance. Indeed the world is in permanent evolution and software systems have to follow this evolution. We cannot tolerate to build a software system and keep it as it is for a long period. Reasons for this are various: new technologies appear, those include middleware, web services, ORM ..., the user requirements evolve or the target environment of the software and so on. In order to meet those evolutions a specific process in *Software engineering*, the disciplined approach to build software, exists.
Software engineering is built on three main phases [21] :

- System definition : Systematic and structured analysis of the system and users in order to define the needs and produce a requirement and a technical specification.
- Implementation : Concrete realisation of the specification. This process includes the conception, the validation and the testing of each function.
- Maintenance : Modification to correct faults, to improve performance or to adapt the software to an environment change or a requirement change.

Maintenance is the process that handles the evolution of the system. The software evolution domain has been explored quite a long time ago by Lehman [19] [20], he revealed that software evolution follows certain laws, such as declining quality, increasing complexity or continuing growth.

Banker [1] shows the relation between the complexity of a software, its size, modularity and cohesion with the increasing costs of the maintenance step in a program's life-cycle.
Maintenance is then an important, complex and costly process. We will see in the next section that it has also a strong link with the reverse engineering process.

1.2 Program comprehension & Reverse engineering

As seen in the previous section, systems evolve and the maintenance process is crucial in the program life-cycle. To be effective in this process, programmers or modellers depend on the comprehension of their working environment, tools, software or databases. For instance legacy systems are old and complex, the team who created the program in the first place are often no longer there, the importance of documentation is then crucial in order to understand the system before considering to evolve it.

Unfortunately it is very rare that such documentation is available. Either the documentation has never been written or it has become obsolete. Since no change can be made to an information system before one can get a precise and detailed knowledge on its functional and technical aspects, there is a strong need for rebuilding the lost documentation of the system to be maintained.

Therefore an important part of the maintenance process (60% according to relatively old references [10], [2]) is devoted to *Program Comprehension* and *Reverse Engineering*, the goal of which is to recover proper documentation, a higher level of abstraction, from available artefacts such as source code, data or program architecture.
Program comprehension is also a complex process because it is directed by several methods, depends on the knowledge of the developer and on the available artefacts. Different techniques can be used to guide the comprehension process. We can mention for example, textual analysis, syntactical analysis, control flow or the data flow analysis ...
In [26] all methods and techniques are listed, and the complexity of the task is discussed.
Our work was motivated by a *re-engineering* process. This process, in addition to reaching a higher level of abstraction needed in program comprehension, aims to rewrite the code in a new form. Such a task may arise when user requirements or environment changes occur. In our case it was the need for a database migration that conducted our research. To fulfil this goal, we do not only need to understand the programs, but we also need to understand the database. Database comprehension is reached through a specific process called *Database revere engineering*. The next sections of this work mainly focus on this process.

1.3 Database reverse engineering

In addition to database restructuring, migration or evolution, the database reverse engineering process can be needed prior to any modification of data-intensive software. The database is indeed very present and important in each of the three *Software engineering* processes.

In the design phase, the database must represent adequately the application domain. In the implementation phase we have to build it following transformation rules to fit the specified DBMS model and manage it using the right technology in the source code and following good practices. Finally the maintenance phase will impact the database as well, since all evolution of user requirements or modification of the system may involve

the modification of the database schema or at least the way the database is used by the programs.

This is why database reverse engineering has become an increasingly important process. It is indeed crucial to recover the database documentation. Comprehension of the database can help getting rid of dead structure, improve bad design, or reduce the complexity of the source code. Therefore improving the performance and stability of a system, regaining the control of the heart of the system.
Methods and techniques already exist to support the database reverse engineering process, see Section 2.2 but those can be improved with additional steps and techniques, with the aim to take into account the growing size of actual database structures and the emergence of new data manipulation technologies used in the source code.

1.4 Research Questions

This thesis aims to propose a revisited database reverse engineering method and to present a set of new techniques and tools supporting this method. These techniques and tools aim to support the recovery and understanding of a undocumented database structure. The main novelty of this work consists in mining the past and the evolution of the database schema, in order to reach an enriched and finer-grained database comprehension. These main questions will be the conduct line of this thesis :

- Which new techniques can support the database reverse engineering process?
- How can we support the visualization and comprehension of large database schemas?
- To what extent can the history of a database schema help the understanding of its current version?

1.5 Thesis Limit

This thesis focuses on the recovery of the logical schema of a database. As we will see, the database reverse engineering process is made of several steps which ends with the conceptualization phase. Conceptualization requires a good understanding of the application domain. Our techniques and tools do not support the recovery of a conceptual database schema, but they aim to recover, read and understand a database schema at the logical level.

1.6 Thesis Structure

Chapter 2 presents the state of the art related to database engineering and reverse engineering, schema evolution, clustering techniques and program slicing. This chapter aims to give an overview of the different existing approaches in the research community of these domains. The "Methodology" chapter details a modified methodology in database

reverse engineering, and explains the changes brought in this thesis. Chapter 4 presents the design of some steps described in chapter 3. It contains the specifications of the techniques applied in a database reverse engineering process. Chapter 5 exposes a few implementation details of the developed tools. Chapter 6 constitutes in a case study to validate the developed techniques. The chosen application is OSCAR, a Electronic Medical Record (EMR) designed to help improve health care from individual to population health levels while reducing costs. Chapter 8 concludes this thesis and chapter 9 open discussion about the future works to improve and extend the techniques.

Chapter 2

State of the art

2.1 Database engineering

Since this work mainly focusses on databases and database reverse engineering, it is a prerequisite to have some basic notions of the database engineering process. We will describe this process in this section.

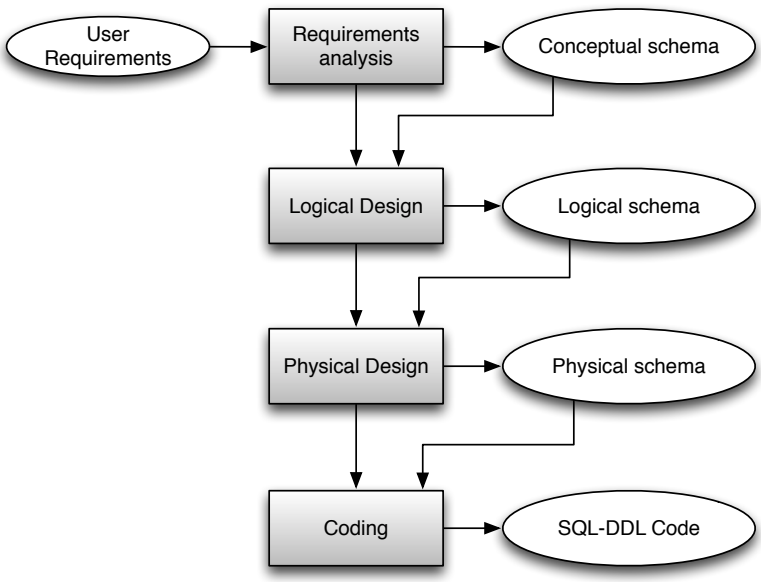


Figure 2.1: Building process of a relational database

The database engineering process aims to build a database by following a disciplined approach [18]. Several steps are needed, each step guided by specific goals and using its own artefacts as input and producing its specific outputs. Those steps incrementally decrease the level of abstraction starting from the *user requirements* and ending with

the *SQL DDL Code*.

The global process is depicted in 2.1. First, the *requirements analysis* phase takes as input the user requirements, the developers get them through interviews and documentation about the application domain. It formalises the requirements in a *conceptual schema*. This schema is expressed in the *Entity-relationship model*. It is the highest level of abstraction of all the database schemas produced during the engineering process. The database is described through three main schema constructs:

- Entity types : represent concepts of the application domain. Such as Order, Client or Product. An instance of an entity type is an *entity*. Entity types can also be part of a hierarchy of entity types, through *is-a* relations.
- Relationship Types : Relationships between entity types can exist and express other concepts of the application domain. For example a client *buys* a product. This relationship belongs to a class in the conceptual schema which is the *relationship type*.
- Attributes : Relationship or entities can have characteristics of their own, those are attributes. For instance a client has a *name* and an *address*.

The second step consists in transforming the conceptual schema into the *logical schema*. This step is called *logical design*. The logical schema is usually based on the *relational model* (for relational database management systems[1], there exist other types of databases that follow other models but they are not explored in this thesis), thus it must be comply with the available data structures. Two main structures define the relational model :

- Tables : Represent an entity type, it is a set of attributes. A database is a set of tables. An entity will be represented by a row.
- Columns : Represent the attributes of an entity type. Values of specific types can be assigned to them.

Specific constraints can be assigned to ensure the semantic preservation of the original conceptual schema. *Identifiers* are a uniqueness constraint on a particular column or set of columns. If such a constraint exists, the table cannot have two distinct rows having the same values as identifiers. *Foreign keys* represent a referential constraint between two tables. The columns that are declared as foreign keys in the source table must have values that exist in the corresponding columns of the target table.
To transform a schema to another schema following another model, there exist specific *transformation rules* described in detail in [17].

The third phase of the engineering process is the *physical design*, it is a simple task that consists of adding indexes, assigning physical spaces to tables, or defining views to

[1] called DBMS from this point

increase the robustness and the efficiency of the database. The output is the *Physical schema*.
Finally the *coding* phase takes the physical schema and translates it into an executable code for a particular DBMS, for relational ones the language will be SQL-DDL. This phase is generally fully-automated.

2.2 Database reverse engineering

Understanding a database schema is important to know the conceptual representation, to understand the application domain. Without available documentation, reverse engineering is necessary. It is generally intended to redocument, convert, restructure, maintain or extend legacy applications. But as the main goal is to recover the documentation, this documentation being then used to serve further goals. From [16], those main objectives are :

1. *System maintenance.* To maintain the system (fixing bugs, modifying the implementation, ...), the understanding of the system is needed.

2. *System reengineering.* Changing the internal architecture without modifying the external specification to obtain a cleaner implementation. The technical aspect must be the focus.

3. *System extension.* To add some features to a system, the existing functions must be known.

4. *System integration.* Merging two systems together is not a simple operation. The specifications of both systems are required, even for the database.

5. *Quality assessment.* Analysing the code and the data structures of a system in some detail can bring useful clues about the quality of this system, and about the way it was developed.

6. *Data extraction/conversion.* In some situations, the only component to salvage when abandoning a legacy system is its database. To use this data in another system, they usually have to be converted in another format. The semantics of these data must be known.

7. *Data Administration.* DBRE is also required when developing a data administration function that has to know and record the description of all the information resources of the organization.

8. *Component reuse.* Adding new components into a legacy system can bring some already existing functions. Reverse engineering can give to the developer the current functions of the system to avoid double component, and support the reuse of the existing ones.

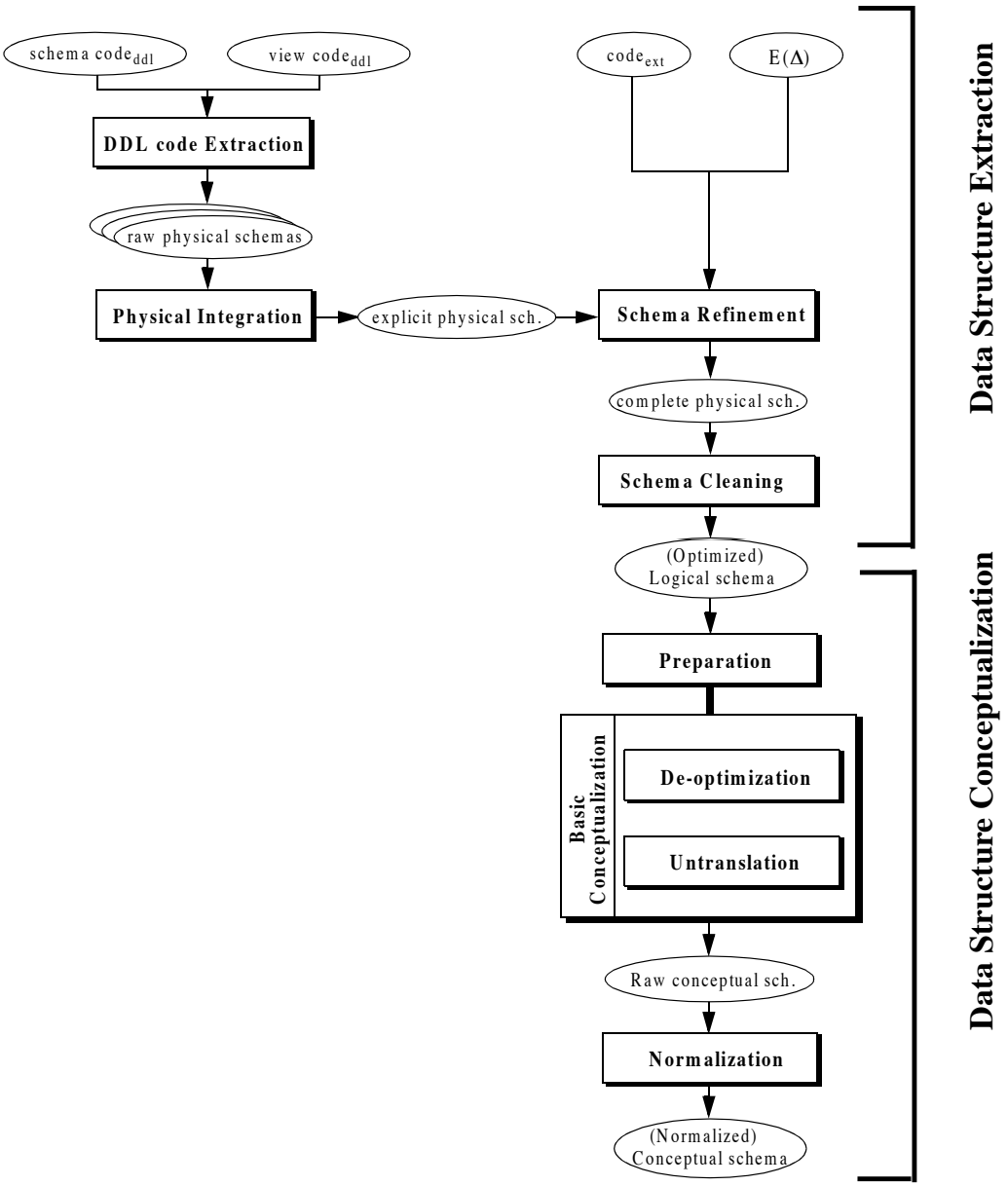


Figure 2.2: Detailed methodology of database reverse engineering (taken from [16])

As mentioned in [16], the general database reverse engineering methodology is considered as the reverse of the forward engineering process. Figure 2.2 shows the different phases with their corresponding inverse.

The main process of the reverse methodology is divided in 2 main phases with subprocesses :

1. Data structure extraction : This step aims to rebuilt the physical and the logical schema including all the implicit and explicit structures and constraints. For this we have to use as input the available code, the ddl[2] code as well as the triggers, procedures, application code... The main challenge is to recover the implicit constructs.

2. Data structure conceptualization : This process tries to specify the semantic structures of this logical schema as a conceptual schema. The logical schema must be untranslated. Through this process, the analyst identifies the traces of such translations, and replaces them with their original conceptual constructs. Then, the result is de-optimized : the logical schema is searched for traces of constructs designed for optimization purposes. And finally, the process restructures the basic conceptual schema.

In our work we focus on the ***Data Structure Extraction*** process. The first part of the figure (figure 2.2, page 16) explain the different phases of the process :

DDL code Extraction : This phase consists in extracting the code of every part of the schema (all the different partial views, subschemas, source code files, ...).

Physical integration : When more than one DDL source has been processed, the analyst is provided with several, generally different extracted (and possibly refined) schemas. The final logical schema must include the specifications of all these partial views, through a schema integration process.

Schema refinement : The input explicit schema is only based on the explicit constructs found in the system. But this is not enough; all the useful information are not always expressed in the DDL code.

> *An explicit construct is a component or a property of a data structure that is declared through a specific DDL statement. An implicit construct is a component or a property that holds in the data structure, but that has not been declared explicitly. In general, the DMS is not aware of implicit constructs, though it can contribute to its management (through triggers for instance). The analysis of the DDL statements alone leaves the implicit constructs undetected.[16]*

Indeed, the application itself can help to find other information, such as the GUI, screens, forms, ... There is a large set of potentially implicit schema constructs, as shown in Figure 2.3. They are crucial in recovering the logical schema as the set of explicit construct can be very small compared to the set of implicit ones. According to [16] up to 50% of the schema structures and constructs can be implicit. One simple example for those implicit constructs are the foreign keys. In old legacy systems relying on old database management systems such as MyIsam,

[2]Data Definition Language

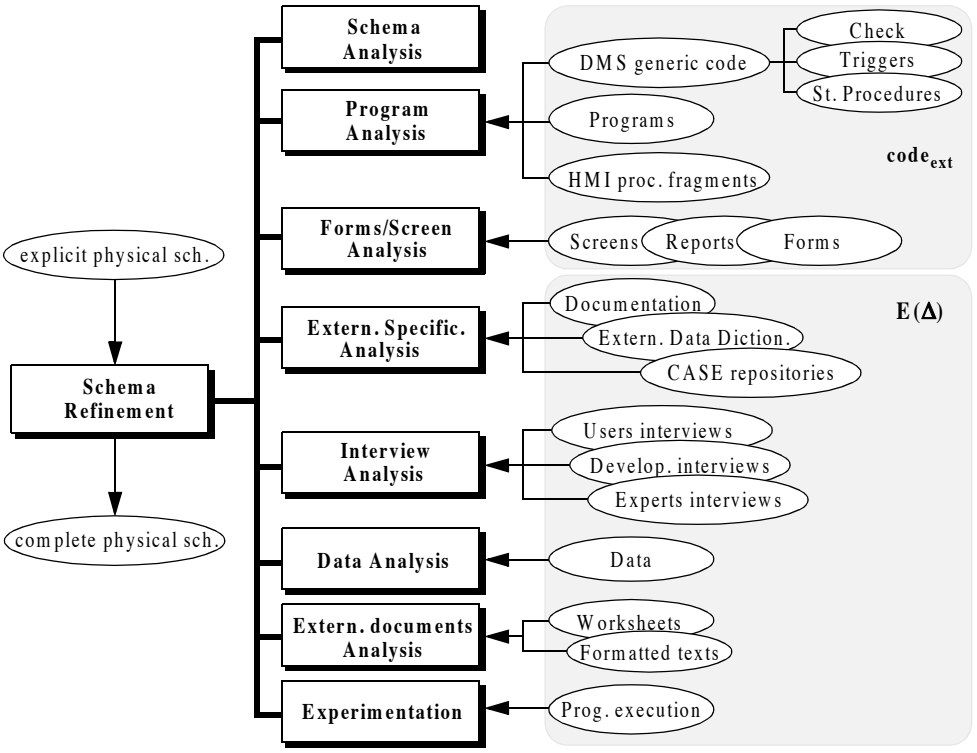


Figure 2.3: Detail of the Schema Refinement process (taken from [16]).

the foreign keys could not be handled, and therefore it was not possible to declare them in the DDL code. Other techniques had to be used. Since it is difficult to imagine a database schema without referential constraints, the recovery of such implicit constructs is mandatory. Due to the emergence of new technologies and the increasing complexity of software systems, we have added new artefacts to be considered in the *Schema refinement* process, such as the property files of an ORM or the history of the DDL code. Cleve *et al.*[8] have also contributed to the enrichment of the schema refinement process. They proposed different techniques to explore SQL statement execution traces. The modified methodology of DBRE is detailed in 3.

2.3 Schema evolution

There exists a series of previous works regarding schema evolution [25]. Those include database schema evolution as well as application/software evolution or even ontology evolution. Works regarding database evolution are mostly about object oriented databases.

An example of work is Lerner [22]. They provide support for schema changes in object-oriented database. The developer uses a declarative notation to map the old version objects with the new ones. They provide a tool that applies those mappings to update the database schema as well as the data, while enforcing data correctness and completion.

In [28] the authors try to quantify the changes applied to a schema during a certain period of time and show their consequences.
The work of Curino *et al.* [11] is a first step in analysing and supporting the schema evolution of a relational database. The authors provide a support for the database administrator regarding the impact of a particular evolution of schema. The goal is to guarantee information preservation, redundancy control and invertibility. Given a set of SQL requests, a database schema and SMOs (Schema modification operators), the tool generates equivalent SQL requests and a migrated schema. This approach is operation-driven and the history of versions is used to confirm information preservation.

As a byproduct of its use this tool creates a complete, unambiguous documentation of the schema evolution history, which is invaluable to support data provenance, database flash backs, historical queries, and user education about standard practices, methods and tools.

Those works have analysed rather small schemas for a quite short period of time. Sjoberg [28] works on a schema growing from 23 to 55 tables in a one year. Curino *et al.* [11] has worked on the Wikipedia database schema, considering a period of 4 years (171 versions) during which the database has grown from 17 to 34 tables.

In our work we will present a new approach schema evolution analysis. We use the source code repository as input artefact and create a global, integrated view of the database schema history (see section 4.4). This innovative approach aims to support the understanding the current database schema and of the programs and it consistutes a basis for supporting future evolutions. Moreover our case study is much larger than the previous studies in this field, it considers a period of ten years and a current schema version including more than 400 tables.

2.4 Clustering & Filtering

Antonio Villegas and Antoni Olivé [30] focused on the problem of filtering a fragment of the knowledge contained in a large conceptual schema. There are many information system development activities in which people need to get a piece of the knowledge contained in the conceptual schema. The larger the schema is, the more difficult it is for a user to get the interesting knowledge he needs.

The goal of the authors is not simply to divide a large schema, but to fragment it into subset of entities relevant for the user. They propose a new technique to identify important entities: they do not believe all entity are equally important for a user.

Their method computes the *interest* as a combination of the *closeness* and *importance*

to obtain a ranking of the most interesting (according to the knowledge request) entity types to the user. The *closeness* between 2 elements is the distance between them, in term of distance name (for example). They define the *importance* of an element as the number of connection it has with the rest of the schema. The more connections exist, the more important is that element. A filtered conceptual schema is a subset of the original one, and thanks of its reduced size it is more comprehensible to the user.

The authors assume that finding relevant information for a user from a conceptual schema ($\mathcal{CS}$) includes three components :

1. Focus Set ($\mathcal{FS}$) : what the user is interested in about the schema.
2. Rejection Set ($\mathcal{RS}$) : the entities that the user do not want to be part in the result (the filtered schema).
3. Filter Size ($\mathcal{K}$) : the number of entities in the filtered schema, e.i. how much knowledge the user wants to obtain.

The importance of a table belonging to the $\mathcal{CS}$ is a real number. The chosen methods by Antonio Villegas and Antoni Olivé are those based on *link analysis* : the more important the entity types connected to an entity type are, the more important such entity types will be. In [29], the authors explain the way they compute the importance (Ψ).

The closeness (Ω) between a candidate entity type e and the focus set $\mathcal{FS}$ should be directly related to the inverse of the distance of e to $\mathcal{FS}$. The interest takes into account both the measure of the importance and the closeness, with a balancing parameter (α). This is used to set the preference between closeness and importance.

$$\Phi(e, \mathcal{FS}) = \alpha \times \Psi(e) + (1 - \alpha) \times \Omega(e, \mathcal{FS}) \tag{2.1}$$

The result set will contain the $\mathcal{K} - \mid \mathcal{FS} \mid$ top candidates entities.

The authors provide a set of transformation for the constraints : the integrity constraints, IsA relationships and relationship types are maintained in the filtered schema, even if some entities are not in the filtered schema.

2.5 Program and Database Slicing

Program slicing has been a prolific research area since Mark Weiser invented it in 1981 [31]. The original idea is to decompose a program, obtaining the statements related to a specific value or a specific statement called a *slicing criterion*. There exists two kinds of slicing, the static one that only analyses the statements regarding a statement and a set of variable only and giving as results the statements that affects this set for any execution. And the other way is the dynamic way that takes into account different instances of variables, and thus provides the statements regarding a specific execution of the program. This technique is very useful for debugging.

Following this original idea lots of different algorithm and techniques were developed, *Backward slicing*, finds the statements that could influence the slicing criterion, *Forward Slicing*, finds all statements that could be influenced, *Chopping*, *Relevant Slicing*, and so on, [27] lists and compares the most common techniques.

An interesting aspect of the program slicing is the *Program Dependence Graph* (PDG) in [13]. This graph represents the dependencies between the successive operations of the program. A node represents an operation. If another node is linked to it with a solid arrow, this means that the execution of this node depends on the execution of the other. Similarly a dotted arrow means there is a data dependency, one initializes some variables used by another.
But now the system not only lays on the software itself, more and more systems are data-intensive, they use data and need them to be operational. Program comprehension, evolution and maintenance now also depend on the database. But so far very few authors have integrated them into the program slicing process, only taking into account the data dependencies of the program state, the database state is missing. Some have tried to revisit the PDG and integrate new connections in order to improve the accuracy of the computed slice and integrate the DB state. It is the case of Willmor, Embury and Shao [33], they proposed to add two new connections to the PDG. The first one will represent the interaction between program and the database state. The second one is between statements that modify common parts of the database state. They then create new relations between the nodes of the PDG that can be added to it and thus creating a more complete graph. As conclusion of their work they proposed new forms of slicing to investigate:

> *For example, rather than giving a set of variables of interest as part of the slicing criterion, one could envisage supplying a list of tables or attributes from the database schema. The slice produced would be limited to only those statements that were directly or indirectly related to the manipulation of those schema elements.*

In our work we tried to contribute to this objective, our method is presented in 4.5.2.

Cleve *et al.* [5] extended the work of Willmor *et al.*, they enlightened the complexity and the different forms the *Database Manipulation Language* (DML) and tried to develop a DML-independent way of constructing the PDG. This work focusses on recovering the dependencies between program and database *variables* instead of program and database *statements*.

Most of the techniques listed above do not (sufficiently) integrate the database in the process of program slicing, however databases are more and more important in recent software systems. Few techniques integrate it, but still some aspects can be further explored by the developer who needs to understand the programs or the database. Most approaches make us of *static* program analysis techniques. We have seen that *dynamic*

analysis techniques also exists. In our work we tried to develop a new technique, inspired by existing program slicing techniques, that integrates the database in the process and uses dynamic program analysis approach. This technique is detailed in 4.5.1

Chapter 3

Methodology

Considering the classical approach to database reverse engineering, we will explain in this section the methodology we followed, the problems encountered and the techniques we developed to face those problems.

3.1 A revisited database reverse methodology

During our internship, we had to reverse engineer the database schema of the OSCAR system. This software is a large data-intensive system that is completely described in Chapter 6. Naturally, we used the methodology [16] explained in the state of art section (Section 2.2) and illustrated in Figure 2.2. We encountered some limitations of the methodology. Considering the size of the schema we found that an adapted methodology and some new supporting techniques could be useful to facilitate the global process.

We started with the extraction of the DDL code from the different files constituting the entire schema. The **DDL code extraction**, and the **Physical Integration** are rather simple and consist in executing a command line to dump the database source code.
The **Schema Refinement** process is the most important step since implicit constructs have to be discovered. Our explicit physical schema is very large, and there are only a dozen foreign keys explicitly declared. The **Schema Refinement** defined in Figure 2.3 could be extended with new artefacts due to the use of new technologies and the emergence of new techniques. As said before new technologies appear and have to be integrated in the reverse engineering process. We therefore decided to integrate those new items in an extended **Schema Refinement** process. The modified process is illustrated in Figure 3.2.

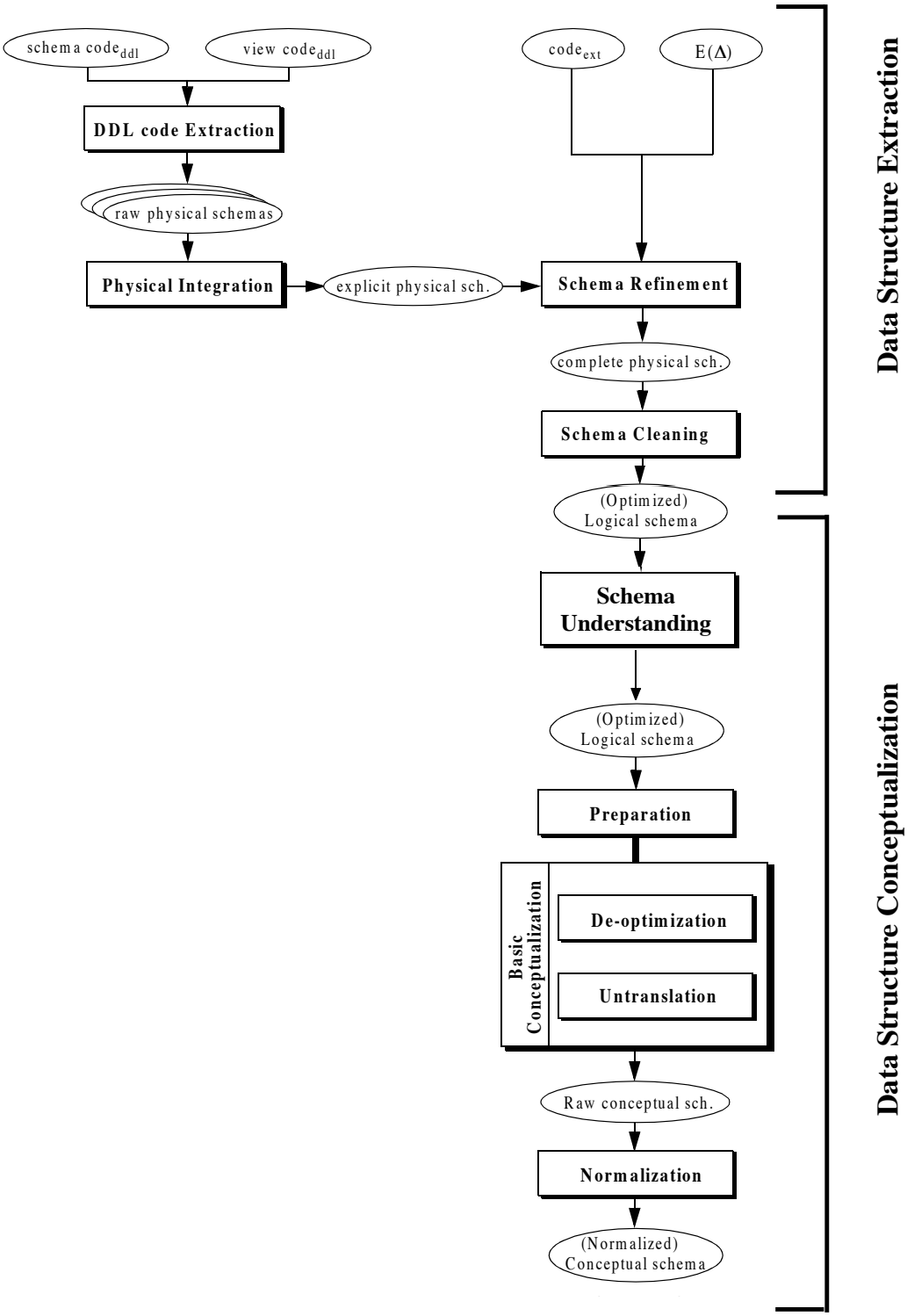


Figure 3.1: A revisited methodology of database reverse engineering (inspired from [16])

After the **Schema Cleaning** step, the schema was still huge, and complex. Reading the schema to understand it is a difficult task, specially with a very large schema. Before starting the **Conceptualization phase**, it is required to understand the schema. The available tools were not good enough in the presence of such a large schema. The **Preparation** step requires some information about the logical schema, so we needed to better understand the schema. We propose here several techniques that we applied in a new process called "***Understanding the schema***". This new step comes just before the **Preparation** step, as shown in Figure 3.1.

The Preparation step consists in preparing the schema to the conceptualization phase, that is to say, removing obsolete structures (that are no longer used) and the technical structures which do not model the application domain. The step also performs some "cosmetic" changes such as improving convention names. [16].

The new step "Understanding" is there to facilitate the job of the **Preparation** step. To identify the obsolete structures, the technical ones, ... that must be removed because they are not relevant for the conceptualization.

This new step will not transform the schema, but will bring information about database objects to help the analyst to extract semantic concepts underlying the logical schema. It could be very helpful during the **Basic conceptualization**. Understanding the schema and the application domain is necessary when performing the ***Conceptualization*** phase.

3.2 A modified schema refinement

As said above, system evolution is becoming more important and complex due to appearance of new technologies. Therefore new artefacts can emerge and those have to be integrated into the reverse engineering methodology. Nowadays, it is not uncommon for the systems to have an additional layer between the business-specific code and the data. For instance, *Object Relational Mapping*[1] frameworks such as Hibernate are more and more used. The mapping files of those tools can contain valuable information about the database schema, since these files map the relational database schema to the object-oriented structures of the application programs. So, we decided to add this aspect in the **Schema Refinement** step, as shown in Figure 3.2. This process takes the mapping files as input, but it can also take all other potential sources from the ORM.

Here, we focus on the implicit constraints of the schema. There exist different techniques to recover those constraints, as mentioned in [9]. Let us assume the main purpose of the **Schema Refinement** is to recover the foreign keys. Foreign keys form a major structuring construct in relational databases [4]. They bring important information about the domain because they represent the relationships between entity types. Some artefacts of

[1]ORM, a layer that transforms data and database structure in object-oriented, creating thus a virtual object database.

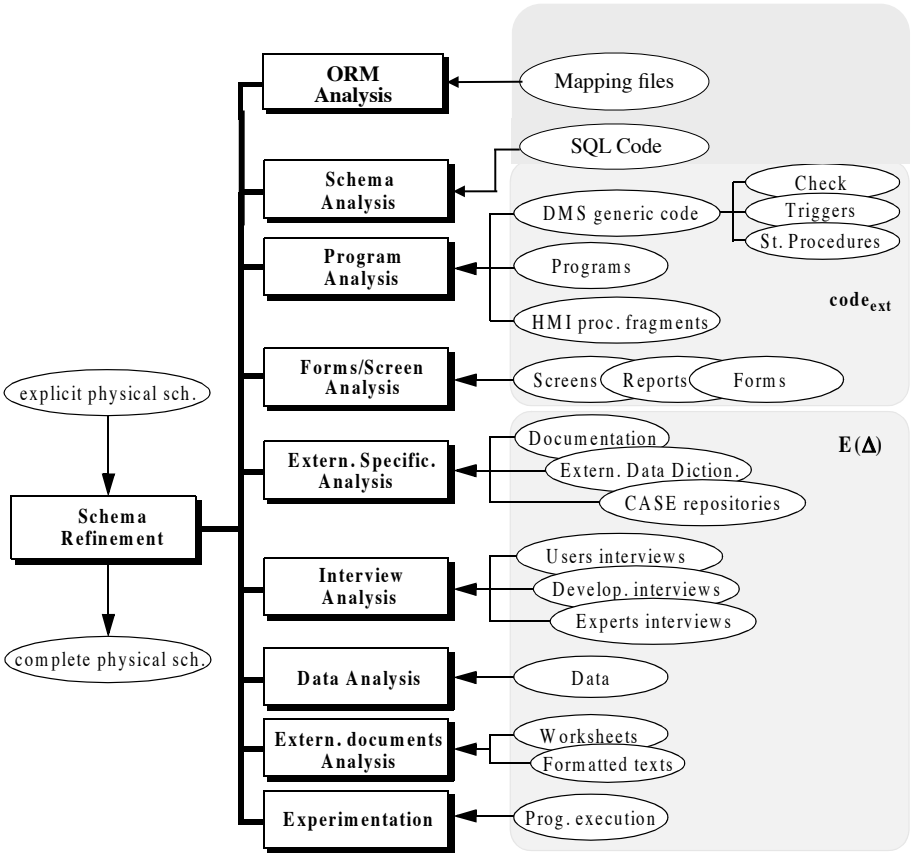


Figure 3.2: The modified Schema Refinement process (inspired from [16])

Figure 3.2 are not always available : No documentation, no other study results, no data, impossible to interview experts.... Adaptive techniques have to be found regarding the working environment.

Whatever the artefacts and the techniques used to discover implicit constructs, the method follows the same principles. It is described in Figure 3.3. The first step is *finding* them. Different techniques are available : heuristics based on column names, SQL statement analysis, program slicing, DDL code analysis, etc and our new ORM Analysis. When then extract a set of *candidate* implicit constructs. We therefore have to proceed with a *validation* step which will ensure that those hypotheses are true. We have to either validate or invalidate candidate foreign keys, by analysing the data, the data usage or the technical constructs.

In addition we bring some new sources of information to increase the techniques to find and/or validate the implicit constraints of a schema, during the Schema Refinement step. This innovative way to disprove the potential foreign keys is by using the historical schema (see section 4.4). Each column having a meta-attribute with its creation date,

Figure 3.3: Schema Refinement : inferring foreign key process

we can confront them to disprove an hypothesis. Indeed if

$\exists FK : Order.idclient \rightarrow Client.Id : Order.idclient.creationdate \geq Client.id.creationdate$

We can exclude that this constraint is actually a foreign key. A target column cannot have been created after the source column. Further use of the historical schema will be explained in the next sections.

3.3 Understanding the schema

We added a preliminary **Understanding** step in the process of reverse engineering (Figure 3.2). This was important before starting the conceptualization. Since the conceptualization requires to analyse the complex logical structures (see [4] [16]) to convert them into conceptual constructs such as *is-a* relationships, it is important to deeply understand the semantics of tables and relationships. As we work with a large database schema this step can be supported with several techniques as shown in Figure 3.4.

To understand a database schema (small or large), the first step is to read it. If that phase can be very simple in the case of a small schema, the complexity and difficulty can become very soon huge, and make the task impossible for a human being. A instinctive way to make a read easier of something large and complex, is to order it, to rearrange it or abstract it.

When you have the logical schema you may want to conceptualize it or refactoring it. For that you need to read the schema and understand it, knowing the domain related to the database is essential but in large data intensive system this task can be much more complicated due to the size of the schema. To help with this step we applied the concept of cluster to the database schema objects.

The clustering is a well-known method to group data : **Cluster analysis** or clustering is the task of grouping a set of objects in such a way that objects in the same group (called cluster) are more similar (in some sense or another) to each other than to those in other groups. It is a main task of exploratory data mining, and a common technique for statistical data analysis used in many fields.

In a reverse engineering context, the goal of such a method is to group the tables having a link together, to infer some new information. The criterion must be chosen in relation with what the developer is looking for. A cluster is a group of tables sharing a common characteristic. For instance, tables can be grouped by their name thanks to

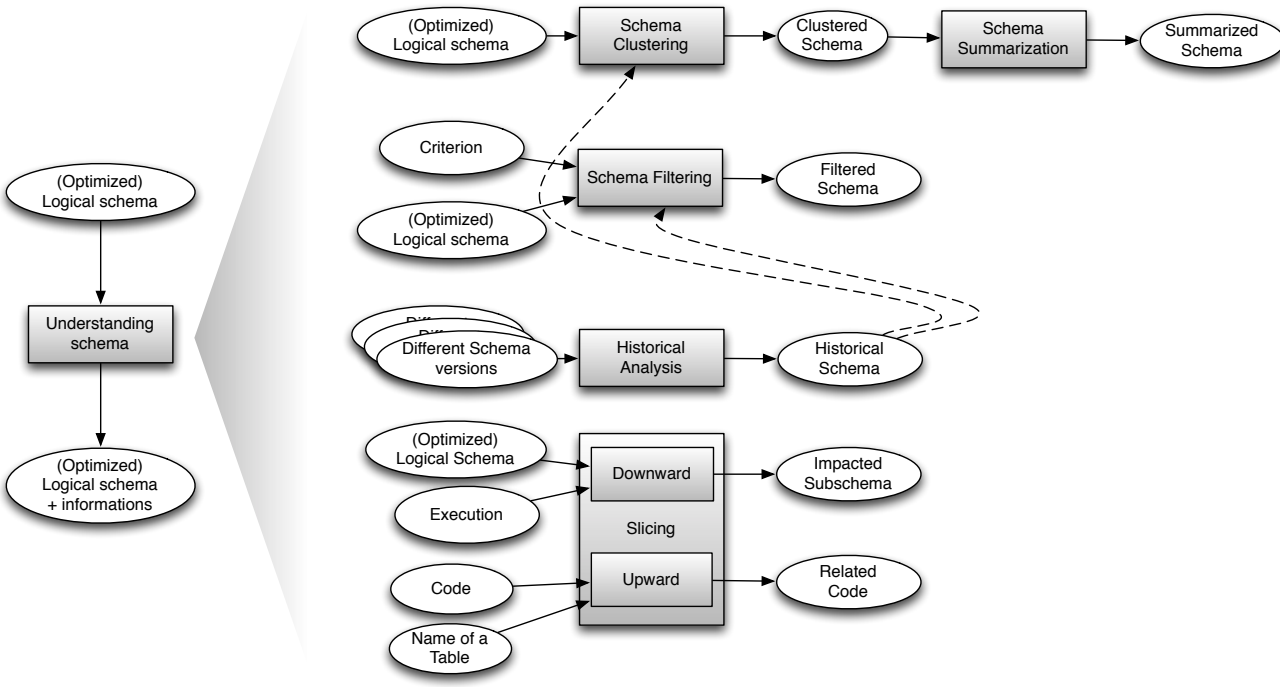


Figure 3.4: Schema Understanding : composition

techniques as alphabetic order (a cluster by letter) or distance name (with jaro-winkler distance). As said above, the developer can be interested in grouping the tables in a way where each cluster contain the table of a specific component. The clustering is also a source of statistic information (the number of column, the number of row in a table (if the data are available), ... can be used too). Each property of a table can be a criterion for the clustering. The clustering can make a schema more readable by grouping tables according to what is relevant for the modeler, developer, tester, etc. The input for this approach is simply the complete schema. We will explain in detail this technique in section 4.2.

Even when the schema is ordered, the tables regrouped, it could still be difficult to have a global view because of the schema largeness. A natural reflex, similar as when we have to retain a long text, we simply outline it. The principle here is to create a **summarized schema** from the clustered schema. A summarized schema is an abstraction of the original schema. In this schema the tables symbolize the clusters, and a column represent a particular table of the original cluster. This technique reduces the size of the schema and allows a general view. The summarization method is detailed in Section 4.2.

As we said, reading a large schema can be a difficult task. Even if the schema is ordered,

the complexity is still there. A natural way is so to read the pieces of the schema that are relevant for the followed purpose. In addition to the clustering process, we propose a **filtering technique**. The purpose of this technique is to extract only a subset of the schema. The clustering only adjusts the entire schema while the filtering focuses on reducing it by providing a subset of interest to the user. The filtering approach is inspired by the method of Villegas and Olivé [30] (described in Section 2.4), but applied to a database schema. The tool we provide creates a new subschema containing the interesting tables for a user regarding specific input criterion. This technique is described in Section 4.3.

The fourth box of the process consists of an **historical analysis**. The technique performs a comparison of all the versions of the database schema. Hence, the created and deleted objects are outlined. The goal is to expose the evolution history of the database schema, the life of all the database objects. Understanding it can indeed significantly aid and inform current and future development activities. Questions such as : *What are the most stable tables?, Which tables are the most recent?* or *Which tables were created or deleted at a certain period?* can then be answered and are a crucial support in finding the most important tables. These informations are necessary to start a migration process or a refactoring. As said in the previous section, the technique can be used for the foreign key validation but can also help detecting bad design pattern evolution, dead code fragments, ... This innovative technique is in its early stages of development and multiple applications are still to be exploited. Section 9 anticipates some of them. The historical schema can, for instance, constitute the input of the clustering and filtering processes : the clustering can arrange the schema according to temporal criterion, and the filtering can query the historical schema.
The specifications of the historical analysis will be described in Section 4.4.

The preparation step aims to detect dead structures in order to remove them for the conceptualization. The dead data structures are obsolete, but have been carefully left in the database by the successive programmers. Several hints can help identify them: they are not used anymore by any program, or they are used by dead sections of programs only, they have no instances, their instances have not been updated for a long time, etc. Finally the **Database Slicing** technique aims to fill the gap. This approach is inspired by the program slicing domain. The program slicing is the computation of the set of program statements (slice) that may affect values at some point of interest. This technique can be used in the specific context of debugging, to locate sources of error, and more generally, in the context of software maintenance, optimization, data flow control, etc. A developer may want to know which database objects are impacted by a particular action performed by the application, in order to redesign the correct tables or to verify the implementation. Moreover he could be interested in identifying all the classes in a program code that uses a particular table. This is useful to verify if this table is still used or no longer used by the application programs.

We therefore used the two kinds of program slicing : the static slicing, which analyses

only the source code, and the dynamic slicing, which works on a specific execution trace of the program.

Our slicing tool provides the two aspects. The static one is given by the *upward slicing* (detailed in Section 4.5.2). The upward slicing finds all references to a given table in program files. The tool searches the Data Access Object (DAO) corresponding to that table, since it represents a gateway to access the table, then the search is done based on that DAO. The dynamic one is given by the *downward slicing* (detailed in Section 4.5.1). The aspect focuses on an execution slice of the database. As in the program slicing, the execution of the program generates successive statements. By catching these statements, we can find the tables impacted by the execution scenario. This is very useful to see the scope of an action on the system.

We can say that our enriched methodology brings new techniques to make the understanding of large schemas easier and faster, at different levels :

- The clustering technique organizes the schema objects
- The summarization technique outlines a clustered schema
- The filtering technique returns a relevant fragment of the database schema
- The historical analysis shows the evolution of the schema (and the way it has been modelled)
- The slicing technique can bring information about the use of a table, and identify obsolete structures

Understanding the schema is important in order to extract the concept of the domain, and their semantics. In the context of schema conceptualization, this is the purpose of the **Understanding process**. But, this process can also be the preliminary step of a schema refactoring or evolution. Database migration is another typical task in the maintenance phase of a software system. In all those cases, the transformation plan of the schema must be well thought, because data and performance are at stake. An in-depth understanding of the database, in particular of its schema and of its usage is very useful in such contexts. Other applications of the different techniques we provide can be imagined, and some of them are already mentioned in Section 9

Chapter 4

Design

In this section, we specifiy the different techniques exposed in the previous section. We describe here the data structures, algorithms, patterns, etc.

4.1 Inferring Foreign Key

As said before, up to 50% of the structures and constraints in a database schema can be implicit. The explicit schema constructs are declared in the DDL code, those includes the identifier constraints or access keys. Those which are implicit may include referential constraints, inclusion constraints ... there are several reasons why they are implicit. In [16] the reasons are listed, we can for instance mention the *Non declarative structure*, this means that it is not possible to express a typical constraint in a specific DMS. For example in old systems, the engine for a relational database was *MyISAM* which does not support foreign key declaration. Since we can not easily imagine a database without referential constraints, those had to be declared or encoded somewhere else.

4.1.1 Where to find the foreign keys?

To recover the implicit constructs such as foreign keys there are several artefacts to analyse, as mentioned in [16]. There exist a static and a dynamic way to recover those constraints. The dynamic way has been explored by [7] and [3] using the embedded SQL statements. Following is a list of the different artefacts that can exploited :

- Generic database management system code : Code of the DMS such as procedures, checks or triggers.
- Schema analysis with heuristics using the names : browsing the schema to find correspondences between names and then inferring the referential constraints.
- User Interface : Foreign keys found can be validated with the GUI, and others can be discovered (forms, etc.).

- Data : Data analysis can be used to discover patterns but also to help validating the candidate forigne keys found through other means.
- Domain knowledge : Knowing the application domain is required to proceed to a reverse engineering process. Talking to expert may help comprehend the domain and thus discovering properties of the schema.
- Program code: The program code is often in charge of implementing the data integrity constraints. The (sequence of) data manipulation statements can be analysed using dataflow analysis or dependency analysis.

Adding to those artefacts, due to the emergence of new technologies we added a new artefact that can be easily analysed and can give important information :

- Configuration files : Many current applications use Hibernate to generate its DAO, so we look into the XML mapping files. Each file concerns one table of the schema. Inside a file each attribute is bound to a java variable in a particular java object that represents this table. (cf 4.1.3).

To discover our implicit constructs we then use two techniques, the analysis of the property files and the analysis of the schema based on names. For each identifier column, we look for a matching based on several rules that increase the precision and decrease the recall. See Algorithm 1 and section 4.1.2 for details.

4.1.2 Name analysis

Naming pattern : Finding foreign keys with the name of database objects starts with the definition of some rules to minimize the number of false positive and false negative foreign keys. These rules are explained below. The following examples are taken from the Oscar case study described in Section 6.

- To make sure that the attributes given are really a possible foreign key (attributes found via SQL formula or O/R mapping tags), the method always checks if one of them is an identifier.
- The name of the origin attribute (the one that will reference the id being read) contains the name of the target table. This way we can avoid to creating keys that should normally not exist (For instance, in the OSCAR case described in the section 6, *allergies (demographic_no)* $\rightarrow$ *demographicaccessory (demographic_no)* is not a foreign key even if the column have the same name). As it is more likely that it references the table " *demographic*", but as it is still possible that this key is a real key, these keys have to be manually analysed. This rule is of course context dependent, not all developers use naming conventions. But it is still interesting to integrate it.
- The name of the origin table contains the target table name (case insensitive). Ex: *app_lookuptable_fields* $\rightarrow$ *app_lookuptable*, because in these tables the attributes don't contain the table name. This rule allows us to prevent the creation of keys that are more likely to be non-existent (ex: name $\rightarrow$ name).

- The origin attribute name is like the target table name concat with " *id*" or " *_id*"(case insensitive). Because some tablse have "id" as identifier and other tables have attributes like "tablenameid".
- At the end another algorithm analyses all the attribute names looking for attributes with a name matching a pattern (ending with *id, _id, no* or *_no*). If these attributes are not already part of a foreign key, they are written to a file to be analysed.

Algorithm 1 General approach of the foreign key generator based on names

Notations.

- Let S_{ENT} be the set of entity types in the schema.
- Let att_{ID} be the set of attributes composing the identifier.
- Let $attName$ be the set of attributes names we are looking for.
- Let S_{ATT} be the set of attributes of the source entity type.

1: **for all** $t \in S_{ENT}$ **do**
2: $att_{ID} \leftarrow identifier(t)$
3: **if** $\exists a \in att_{ID} : a.name \equiv'' id''$ **then**
4: $attName \cup name(t) +'' id'' \cup name(t) +'' \_id''$
5: $attName \cup names(att_{ID}) \backslash name(a)$
6: **else**
7: $attName = names(att_{ID})$
8: **end if**
9: **for all** $s \in S_{ENT} \backslash t$ **do**
10: $S_{ATT} = attributes(s)$
11: **if** $\forall n \in attName : n \in names(S_{ATT})$ **then**
12: **if** $att_S \in identifier(s)$ **then**
13: *We have encountered a possible total foreign key. We list it in an external file*
14: **end if**
15: **if** $matchingNames(s, att_S, t, att_T)$ **then**
16: *We found a possible foreign key, we can add it to the schema*
17: **else**
18: *This could still be a foreign key but it is less certain. We add it to a text file also*
19: **end if**
20: **end if**
21: **end for**
22: **end for**

Algorithm : Algorithm 1 details our approach to name analysis. We iterate on every

entity type and we get its identifier. In line (3-7) we add a specific rule that adds to the set of possible attribute names the name of the entity type concatenated with " *id*" and ' *'_id*" in case the identifier has " *id*" as name. Once we have the set of names of the identifying attributes we iterate over each other entity types to check if they contain this set of names. If the set of attributes of this other entity type contains the *attName* set, it means there is a possible foreign key. But we have to do some additional checks. If the source attribute is also an identifier, we may have encounter a total foreign key and those needs a deeper analysis (see [4]). Line 15 we call a specific function that adds aspecific rule, detailed above. If this function returns false, it does not mean that the foreign key does not hold. It means that it is preferred to analyse it manually.

4.1.3 Mapping files

As said above, the mapping files of the ORM represent the schema to provide the programs with the illusion of an object-oriented database from a relational database. Hence, the file must contain constraints and their parsing can be relevant. To extract foreign keys representated in tags, determining the tag pattern is the first step to build a parser. Once the concerned tags are found in the file, a parsing is required to extract origin and target tables and attributes.
Adding to the specific tags representing referential constraints. There may be *formula* tags, they contain a SQL request acting like a check for a specific attribute. Analysing them can also bring information about implicit constraints. We therefore analyse them given a pattern, if the request does not fit the pattern we record this request to a text file to be further analysed.

4.2 Clustering & Summarization

4.2.1 Cluster analysing

Nowadays the systems are bigger and bigger. The systems include more and more plugins and components. Each of these element brings a set of tables in the database which makes the schema grow. The complexity of the database schema increases, and make it more difficult to read and understand. Analysing a schema or trying to understand it without documentation is a very hard task. It can be useful to know which table is associated with any other, or if they are part of the same component.

Different algorithms already exists. Clustering analysis aim to group a set of objects in such a way that objects in the same group (called cluster, that is to say, a set of element) are more similar (in some sense or another) to each other than to those in other groups. Usually, this technique is used to group data (row of a database, observations, etc), but here we used that to group columns and tables of a database schema. There are 2 different methods of clustering : hierarchical and partitioning data.

The first method seeks to built a hierarchy of clusters. The hierarchical clustering can be performed according to 2 approaches :

- Agglomerative: This is a "bottom up" approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.
- Divisive: This is a "top down" approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

A measure of dissimilarity between sets of element is required. In the case of agglomerative approach, each element is a cluster at the beginning. Every iteration merge the closest cluster, as shown in the figure 4.1. The results of hierarchical clustering are usually presented in a dendrogram.

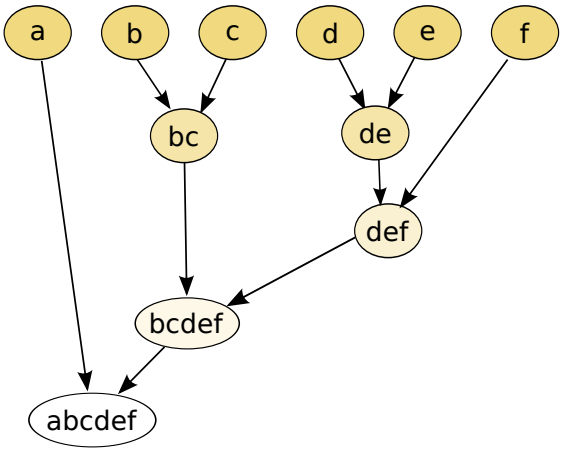


Figure 4.1: Dendogram of an example of hierarchical clustering

The second one divides the data into groups. The goal is to group data that are more similar to each other. It aims to divide a heterogeneous group of data into groups so that the data considered most similar are combined in a homogeneous group and that, on the contrary, the data considered different are found in various distinct groups. Cluster analysis itself is not one specific algorithm : the appropriate clustering algorithm and parameter settings (including values such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results.

In this research context, the data are not statistical observations. The evaluation is different. A naive approach is to create a cluster with the first table, and for every other table check if it can be in an existing cluster. If not, a new cluster is created with that table. For instance the developer's name ; a table has only one developer, so a cluster is an aggregate of table from the same developer. A table can only be in one cluster.

In the case of comparing the other characteristic of a table, this could be different.

Indeed, the function determining if a table must be in a cluster has to return a value between 0 and 1. The chosen cluster for a given table will be the one with the highest value. This value is the "distance" between the table, and all the tables already in the cluster according to the chosen criterion. For instance, to compare the table names, the Jaro-Winkler[1] distance can be used.

A threshold can be defined to specify the content of a cluster. The threshold refines the content : it makes it variable. If the threshold is 100% (strict), to accept a table in a cluster, the closeness between the two of them must be 0, meaning they must be identical. However, if the threshold is not strict, the table can be admitted in several clusters. For example, if the cluster have to contain all the tables having the same age, the threshold will be strict. But, it can contain the tables of the same age around 3 years. Thus, the threshold will not be strict. A table t can be part of different clusters, but it will be added in the closest one.

Algorithm 2 General approach of clustering

```
1:  // initialization
2:  table ← schema.getTable(0)
3:  createCluster(table)
4:  // process
5:  for i = 1 → schema.size() do
6:     table ← schema.getTable(i)
7:     j ← 0
8:     level ← 0
9:     clusterid ← 0
10:    while j < clusters.size() do
11:       c ← clusters.get(j)
12:       if level < match(c, table) then
13:          clusterid ← j
14:          level ← match(c, table)
15:       end if
16:       j ← j + 1
17:    end while
18:    if level == 0 then
19:       createCluster(table)
20:    else
21:       c ← clusters.get(clusterid)
22:       c.add(table)
23:    end if
24: end for
```

The general approach of partitioning data is described in the algorithm 2. The details

[1] http://en.wikipedia.org/wiki/Jaro–Winkler_distance

about this algorithm (about the functions, ...) are here ;

- *schema* represents the set of table in the database schema
- *schema.size*() returns the number of tables in the given schema
- *createCluster*(*table*) is a method creating a new cluster containing only the given table. This cluster is added in the list of clusters, named *clusters*.
- *schema.getTable*(i) returns the i^{th} table in the schema
- *match*(*cluster*, *table*) : this the critical point of the algorithm. This is the function returning the value between 0 and 1. (0 represent "no accepted in the cluster", and 1 means that the table is 100% similar to the content of the cluster). If the table can only be part of one cluster, the return values will be 0 or 1 (e.g. if the criterion is the age of the table), otherwise the return value will be in the interval 0..1.

In algorithm 2, the initialization consists in creating a cluster with the first table. Then, all the other tables are compared to the existing cluster. To be added in a cluster x, the table must be close to all the tables already in it. Only the highest matching value, and the number of the cluster is retained (lines 10-17). The threshold is used in the *match* function : lines 12-15 make sure that if the table can be admitted in several clusters, it will be in the closest one. In the match function, if the threshold is not reached, it has to return zero. At the end of this loop (the *while* loop), *level* contain the highest matching value, and *clusterid* the number of the cluster corresponding to this value. If *level* equals zero, it means that for all the existing clusters, the matching has returned zero, so the table can not be accepted in an existing cluster. A new cluster is created with this table (lines 18-19). Otherwise, the table is added in the cluster numbered *clusterid* (lines 21-22).

4.2.2 Schema summarization

The problem discussed here concerns the size of a schema. Even if the clustering analysis can make a huge schema more readable, and support an easier understanding for the developer, it could still be difficult to have a global view of it. The clustering help to understand subgroup of tables, and their intra-group relationships.
The idea is to summarize a schema, to make it more readable by increasing the level of abstraction. The input of the summarization is a clustered schema. The output will be the summarized schema, i.e. a schema where a table represent a cluster, and the columns symbolize the table of this cluster. So, this method bring us to a further abstraction level. A short example is shown with the figure 4.2.

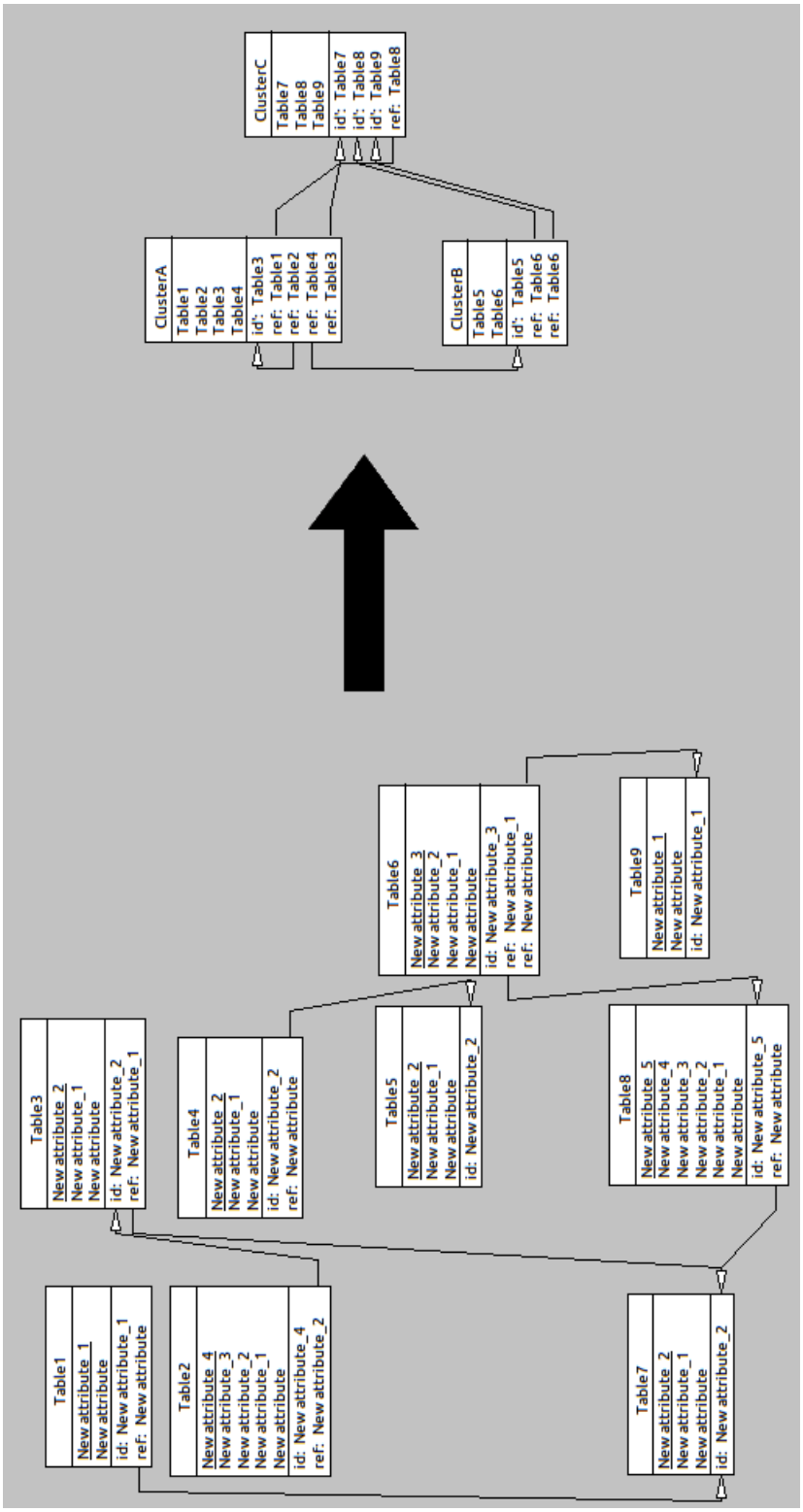


Figure 4.2: Example of summarization

The foreign keys are maintained : the initial relationships between columns are transformed into foreign keys from a column (representing a table) to another column of another table (representing a another cluster). Normally, a foreign key can be a constraint from several columns to the exact same number of target columns. In the summarized schema, the foreign keys are only one-to-one column. Preserving these constraints highlights the relationships and dependencies between clusters.

Algorithm 3 Summarization algorithm

Require: *clusters*: list of clusters from a clustered schema. S is the complete clustered schema.

```
1:  // initialization
2:  S_s ← createNewSchema()
3:
4:  // process
5:  for i = 0 → clusters.size() do
6:     currentCluster ← clusters.get(i)
7:     t ← createNewTable(currentCluster.getName())
8:     j ← 0
9:     while j < currentCluster.size() do
10:       tabletmp ← currentCluster.getTable(j)
11:       t.addColumn(tabletmp.getName())
12:       j ← j + 1
13:    end while
14:    S_s.addTable(t)
15: end for
16:
17: // add the referential constraints
18: for all t ∈ S_s do
19:    for all col ∈ t do
20:       t' ← S.getTable(col.getName())
21:       for all fk ∈ t' do
22:          target_table ← fk.getTargetTable()
23:          target_cluster ← target_table.getCluster()
24:          createFK(t, col, S_s.getTable(target_cluster), S_s.getTable(target_cluster)[target_table])
25:       end for
26:    end for
27: end for
```

The algorithm 3 contains several functions :

- *createNewSchema()* : creates a new empty schema
- *clusters.size()* : returns the number of clusters in the input schema (here, the size of the cluster's list)

- *cluster.getName()* : the table representing the cluster must have a name. The function return the name of the cluster. It may be possible that the cluster does not have a name, the $getName()$ function must be customized according the needs of the analyst.
- *table.addColumn(c)* : adds the column c to the table *table*. The column c is the last one.
- *schema.addTable(t)* : adds the table t to the schema *schema*
- *foreignkey.getTargetTable()* : returns the target table of the current foreign key.
- *table.getCluster()* : returns the cluster containing the current table. Only applied on the table from the clustered schema.
- *createFK(table_origin, column_origin, table_target, column_target)* : creates a foreign key from the $column\_origin$ of $table\_origin$ to the $column\_target$ of $table\_target$.

The algorithm 3 shows how the summarized schema is built. The input must be a clustered schema, whatever the criterion chosen. The initialization (line 2) consists in creating a new schema (the summarized one, named S_s). Then, we add the new tables symbolizing the clusters. At the line 5-15, for all clusters, a new table is created, and filled with columns (one for each table in the cluster). When the new table is filed, it is added to the summarized schema S_s (line 14).
The last phase is adding the foreign keys. For all columns of each table in the new summarized schema, we find the table from the initial clustered schema S (having the same name of the column). Every foreign key of the real table (t', line 21) is transposed in the summarized schema, by finding the table and the cluster of the original schema (line 22-24). A foreign key is created from the current table and column (in S_s) to the column having the same name of the target table in S, contained in the table symbolizing the cluster $target\_cluster$.

4.3 Smart schema filtering

The approach is directly inspired bythe filtering method of Villegas and Olivé [30]. We transposed their method to the logical schema. The purpose remains the same :

> *We focus on the problem of filtering a fragment of the knowledge contained in a large conceptual schema. The problem appears in many information systems development activities in which people need to operate with a piece of the knowledge contained in that schema.[30]*

The method we provide also starts with a Focus Set $\mathcal{FS}$, a Rejection Set $\mathcal{RS}$, and a Filter Size $\mathcal{K}$. The first is a set of tables belonging to the logical schema $\mathcal{LS}$. The $\mathcal{FS}$ can't be empty : it contains the tables corresponding to what the user is interested in. The $\mathcal{RS}$ contains the tables that can't be in the result (the ones the user is not interested

in) and the last, $\mathcal{K}$, is the size of the result set $\mathcal{R}$. As the method applied to conceptual schema, the result set $\mathcal{R}$ has a size of $\mathcal{K}$, and so includes the tables in the $\mathcal{FS}$, implying $\mid \mathcal{FS} \mid \leq \mathcal{K}$. Another constraint is $\mathcal{RS} \cap \mathcal{FS} = \emptyset$.

Our technique is also based on the interest computed with closeness and importance of the element of the schema (here, a table, and no more an entity). The computation methods are different because, in our case, we are working on a logical schema $\mathcal{LS}$ and not on a conceptual schema. Some changes were required :

- ***Importance*** **(Ψ)** : The chosen method is the one based on *link analysis*. We define the importance of a given table in the entire schema as the sum of the foreign keys from this table, and the foreign keys having this table as target. The importance must be normalized : the importance remains a real number in the range $[0, 1]$. The sum of the foreign keys is divided by the maximum of the sum of foreign keys in the schema. So, we can write

$$\Psi(t) = \frac{\#FK_{in}(t) + \#FK_{out}(t)}{max\{(\#FK_{in}(t') + \#FK_{out}(t')) : \forall t' \in \mathcal{LS} \wedge t' \notin \mathcal{RS}\}} \tag{4.1}$$

  where $\#FK_{in}(t)$ represents the number of foreign keys having t as target and $\#FK_{out}(t)$ the number of foreign keys having t as origin.

- ***Closeness*** **(Ω)** : The closeness is computed according to a defined criterion (name, date, color, ...). Due to the large kind of closeness, there is no unique equation to compute the closeness between the Focus Set and the given table. The closeness between two elements can be strict or not. In a strict case, the closeness is 1 or 0 only. We can define it as in the equation 4.2. As we can see, if the table shares an identical value for the chosen criterion the result is 1, 0 otherwise.

$$\Omega(t, \mathcal{FS}) = \begin{cases} 1 & \text{if } criterion(t) = criterion(t'), \exists t' \in \mathcal{FS} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

  A non strict closeness is the distance between the given t and an element of the $\mathcal{FS}$ or all the $\mathcal{FS}$. An offset can be defined to keep down the scope of the closeness. The offset build a ray from all the point of the $\mathcal{FS}$, and every element outside this circle has a null closeness. The inside element have a non strict closeness : the most they are near the center, the most high their closeness is.

  The closeness can also be found by computing the average of the distance from the given element to all the element of $\mathcal{FS}$. In the equation 4.3, the $d(t, t')$ means the distance between the 2 tables. It can be based on the names, the color, etc. For example, it can be the Jaro-Winkler distance between the names of the 2 tables.

$$\Omega(t, \mathcal{FS}) = \frac{\sum_{t' \in \mathcal{FS}} d(t, t')}{\mid \mathcal{FS} \mid} \tag{4.3}$$

– ***Interest*** **(Φ)** : The formula here is the same as the equation 2.1 from [30]. For this reason, the interest can still written as

$$\Phi(t, \mathcal{FS}) = \alpha \times \Psi(t) + (1 - \alpha) \times \Omega(t, \mathcal{FS}) \tag{4.4}$$

The algorithm 4 shows the general way to apply the filtering method to a logical schema, using the functions and formula defined above.

Algorithm 4 Filtering algorithm

Require: $\mathcal{FS}, \mathcal{K}$ and $\mathcal{RS}$ must be initialized and can not be empty (except for $\mathcal{RS}$) such as $\mid \mathcal{FS} \mid \leq \mathcal{K}$ and $\mathcal{RS} \cap \mathcal{FS} = \emptyset$.
α must be in the range $[0, 1]$.

Ensure: $\mathcal{R}$ contain the top $\mathcal{K}$ table interesting for the user.

1: **for** $i = 0 \rightarrow schema.size()$ **do**
2: $t \leftarrow schema.get(i)$
3: **if** $t \notin \mathcal{RS} \wedge t \notin \mathcal{FS}$ **then**
4: $t.interest \leftarrow \alpha \times \Psi(t) + (1 - \alpha) \times \Omega(t, \mathcal{FS})$
5: **end if**
6: **end for**
7: $orderByInterest(schema)$
8: $\mathcal{R} \leftarrow \mathcal{FS} \cup take(\mathcal{K} - \mid \mathcal{FS} \mid, schema)$

The input of this algorithm (algorithm 4) must respect the constraints announced above. For all the tables in the schema, which are not in the rejection set, we compute the interest according to the equation 4.4 (line 4). Then, when every interest is computed, a ranking is made to order the tables from the most interesting ones to the less ones. Finally, we build the result set $\mathcal{R}$ combining the top $\mathcal{K} - \mid \mathcal{FS} \mid$ tables and the $\mathcal{FS}$. We assume *orderByInterest(schema)* makes a descending rank about the interest of all the tables in the schema. The function *take(i,schema)* returns the i first tables in the schema.

The condition in line 3 refuses the table from the $\mathcal{FS}$ because we can't guarantee they will be in the result set. Indeed, if the α parameter is 1, the interest will be equal to the importance and the tables in the $\mathcal{FS}$ are not necessarily the most important. So, we explicitly add them in the result set, at line 8. The rest of the relevant tables are taken among the tables having their interest computed in the schema.

4.4 Historical schema analysis

In order to help the schema comprehension we brought a novative technique exploiting an often forgotten artefact : the source repository and the history of code. This section presents the concepts, the specification and the methodology of a new source of information : *The Global Historical Schema.*

4.4.1 Motivations & Concepts

Software repositories can provide valuable information, facilitating software reengineering efforts with analyses of the evolution histories of legacy software systems. In recent years, many researchers have started to follow a holistic approach, considering multiple diverse software artifacts and the links existing between them (including source code, bug reports, documentation, mailing lists, etc.). However, when analyzing data-intensive systems, comparatively little attention has been devoted to the analysis of an important system artifact: the database. Even fewer approaches attempt to uncover facts about the evolution history of database schemas in order to inform software reengineering and maintenance. We have developed a tool-supported methodology for analyzing and visualizing database schema history

Our general approach consists of extracting and comparing the successive versions of the database schema from the versioning system, in order to produce the so-called *global historical schema*. The latter is a visual and browsable representation of the database schema evolution over time. It contains all database schema objects (i.e., tables, columns and constraints) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime, which in turn serve as a basis for the visualization of the schema and its further analysis. This historical schema can be queried in order to derive valuable information about the evolution of the database, potentially raising other interesting system-specific questions to investigate.

4.4.2 Methodology

In this section we describe the global process that we follow to build the historical database schema of a system. This process, depicted at Figure 4.3, consists of several steps:

1. *SQL code Extraction & Cleaning*: We first extract all the SQL files corresponding to each system version, by exploiting the versioning system (using the extractor depicted in section 5.6.1). Those files may then need to be slightly edited, in case the SQL syntax used does not match with the SQL parser considered[2].

2. *Schema Extraction*: We extract the logical schema corresponding to each SQL file obtained so far, by means of a dedicated SQL parser.

[2]We use the schema extractor of DB-MAIN (http://www.db-main.eu)

3. *Schema comparison*: We compare the successive logical schemas while incrementally building the resulting historical schema.

4. *Visualization & Exploitation*: The historical schema can then be visualized and further analyzed, depending of the project-specific needs.

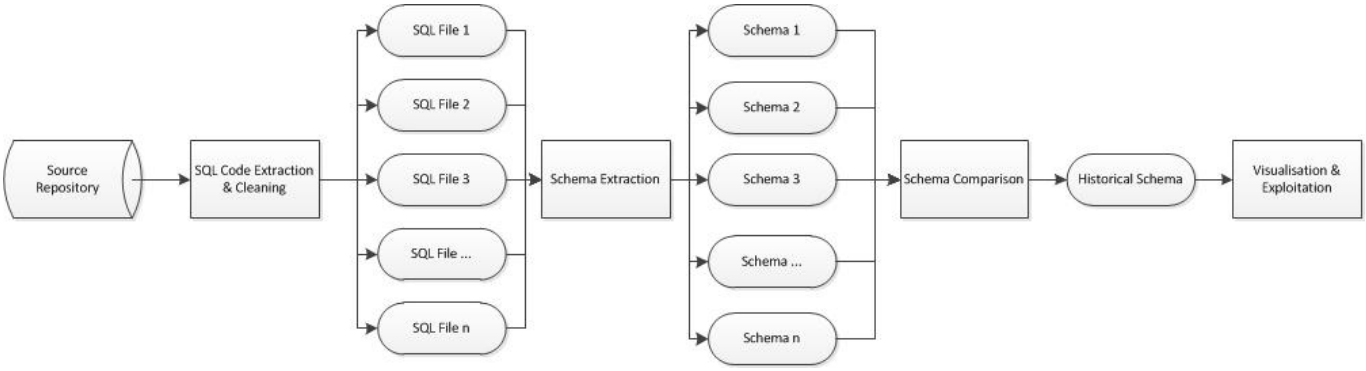


Figure 4.3: Global process

4.4.3 Specification

In this section we further specify the *Schema comparison* process. Figure 4.4 gives an example evolution of a database schema, involving three successive schema versions. Schema S_1 is the oldest one and schema S_3 is the most recent one. We can see that between version 1 and version 2 column A_2 has been deleted, column B_2 has been created as well as table D and its columns. Moreover the entire table C has been dropped. In version 3, Table B has disappeared, table D has been left unchanged, and table C has *re-appeared*. Indeed, it used to exist in version 1, it had been removed in version 2 and it is now back in version 3. We will refer to that phenomenon by saying that a schema object may have *several lives*.

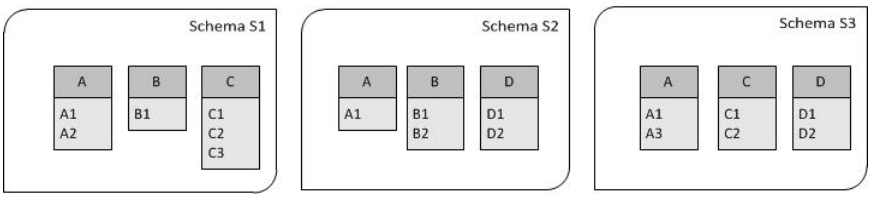


Figure 4.4: Schema evolution example

The historical schema derived from the above schema evolution example is depicted at Figure 4.5. The historical schema is a global representation of all previous versions of a database schema. As we can see, it contains *all* objects that have ever existed in the entire schema history.

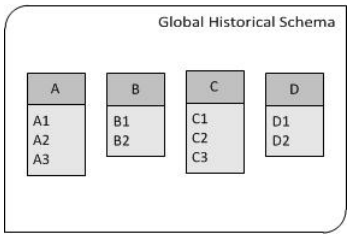


Figure 4.5: Global historical schema obtained from schema evolution of Figure 4.4

Each historical schema object (table, column) is annotated with several meta-attributes:

- *NbVersions*: the total number of versions of the schema where the object can be found.
- *isDead* : true if the object is not present in the latest (current) version of the schema, false otherwise.
- *creationSchema*: references the first (oldest) schema version where the object appears.
- *creationDate*: the date the oldest schema version where the object appears (the date of creationSchema).
- *lastAppearanceSchema*: the last (most recent) schema where the object appears.
- *lastAppearanceDate*: The date of the *lastAppearanceSchema*.
- *severalLives*: true when the object has existed, has been removed, before being restored.

4.4.4 Algorithm

Algorithm 5 formalizes our procedure for deriving a historical database schema S_H from n successive schema versions. This derivation algorithm is based on a pairwise comparison of all those schema versions in reverse chronological order.

The *initialization* step of the algorithm (lines 1-7) consists of considering the most recent schema S_n (augmented with its meta-attributes) as the initial historical schema S_H. We then iterate on all the previous schemas in reverse chronological order (lines 8-32), while comparing the current schema S_i with the current historical schema S_H. The comparison is made by iterating on each schema object (table or column) of both schemas.

Algorithm 5 Deriving the global historical schema from n successive schema versions.

Notations.

- Let S_i be a database schema version, defined as a set of schema objects (including a set of tables and their respective columns).
- Let $date(S_i)$ be the release date of schema version S_i.

Require: $S_1, S_2, \dots S_n$: n successive schema versions.
Ensure: S_H: the corresponding global historical schema.

// initializing S_H
$S_H \leftarrow S_n$
for all $o \in S_H$ **do**
  $lastAppearanceSchema(o) \leftarrow S_n$
  $lastAppearanceDate(o) \leftarrow date(S_n)$
  $nbVersions(o) \leftarrow 1$
  $isOpen(o) \leftarrow true$
end for
// iterating from the last version to the initial version
for all $i \in \{n-1 \dots 1\}$ **do**
  // objects o appearing in more than one version
  for all $o \in S_i \cap S_H$ **do**
    $nbVersions(o) \leftarrow nbVersions(o) + 1$
    if $isOpen(o) == false$ **then**
      $severalLives(o) \leftarrow true$
      $isOpen(o) \leftarrow true$
    end if
  end for
  // objects o deleted before the last version
  for all $o \in S_i \setminus S_H$ **do**
    $S_H \leftarrow S_H \cup o$
    $lastAppearanceSchema(o) \leftarrow S_i$
    $lastAppearanceDate(o) \leftarrow date(S_i)$
    $nbVersions(o) \leftarrow 1$
  end for
  // objects o created after the initial version
  for all $o \in S_H \setminus S_i$ **do**
    if $isOpen(o) == true$ **then**
      $creationSchema(o) \leftarrow S_{i+1}$
      $creationDate(o) \leftarrow date(S_{i+1})$
      $isOpen(o) \leftarrow false$
    end if
  end for
end for
// Final step
for all $o \in S_H$ **do**
  if $isOpen(o) == true$ **then**
    $creationSchema(o) \leftarrow S_1$
    $creationDate(o) \leftarrow date(S_1)$
  end if
end for

Several situations may occur for a given schema object o:

1. o belongs to S_i and belongs to S_H (i.e., $o \in S_i \cap S_H$). In this case, o has appeared in more than one schema version. For such an object, we increment the *nbVersion* meta attribute. If its meta-attribute *isOpen* is $false$, this means that o has *several lives*: it had been removed after version i before reappearing later on, and version i corresponds to the end of (one of) its *previous life*. Therefore, we set its *severalLives* meta-attribute to true.

2. o belongs to S_i but does not belong to S_H (i.e., $o \in S_i \setminus S_H$). In this case, o has been deleted after version i, and it is the first time we encounter it. We then add o to the global historical schema together with its initial meta-attribute values.

3. o belongs to S_H but does not belongs to S_i (i.e., $o \in S_H \setminus S_i$). In this case, we can derive that o has been created in version $i + 1$. We set its *isOpen* meta-attribute to $false$, in order to be able to identify its previous lives, if any.

The *Final* step is performed once all schema versions have been compared to the current historical schema. This step considers all schema objects of the historical schema for which the *isOpen* meta-attribute is still *true*. All those objects have actually been created in the initial schema version S_1. One therefore needs to initialize their *creationSchema* and *creationDate* meta-attributes accordingly.

4.5 Database slicing

Following the idea of *Program slicing* and *Program Comprehension* we designed the *Database slicing*.
For the developer it is legitimate for him to wonder when working on application code *Which tables are impacted?, Which files do I have to take into account when modifying this table?* or *Which User Interface (mainly forms) corresponds to this table?* It's such questions that conducted us to develop what we call the *Database Slicing*. The idea is to find all database objects that are involved in a particular action performed by the user on the application (Downward Slicing) or to find all files in the program source code that refers to a specific table (Upward Slicing).

4.5.1 Downward slicing

The problematic here is to know which table are affected by some actions on the system : what are the real impacts on the database ? The figure 4.6 shows the general flow of the Downward approach.

First, the user start making operations with the system. Then, the user come close to the particular action he wants to analyze. So, the user can start the recording. The recording will save all the actions applied on the database. The user then executes the concerned action, and informs the downward tool when to stop the recording (action

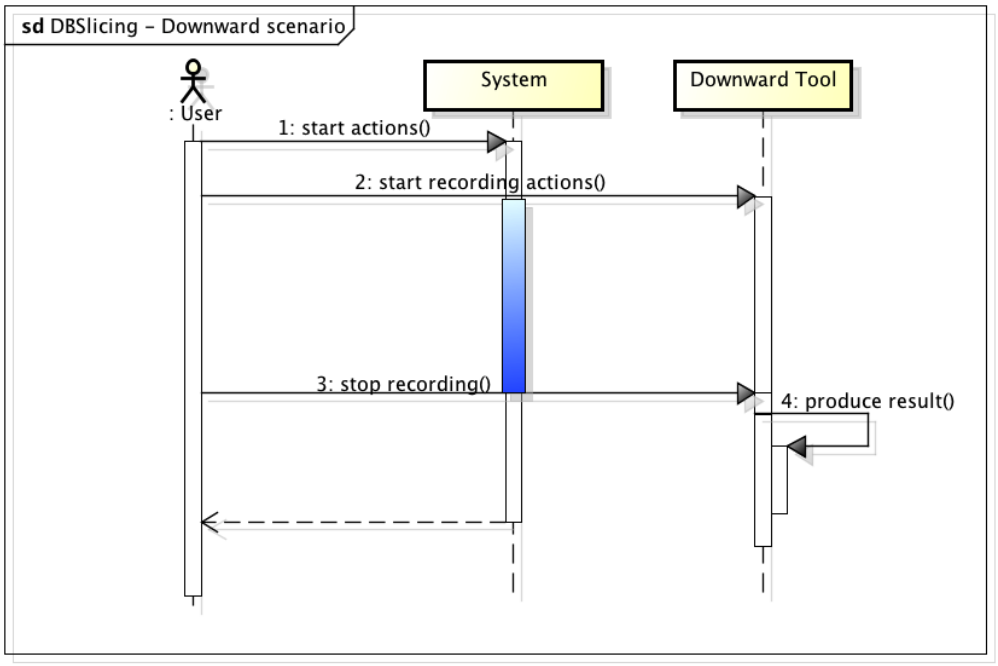


Figure 4.6: DBSlicing - Downward : General scenario

3, on the diagram 4.6). The tool analyses the observations and produces a schema containing the tables impacted by the recording sequence of actions.

The schema provides also some additional information. Indeed, it could be interesting to know **how** the database objects were impacted : by a reading, a writing or both ? Moreover, some tables or columns may be more solicited than others. This information can be useful to the analyst to detect critical database objects. So each object of the result schema has a counter of the number of times the object was read and/or written during the user's actions.

To be very effective and avoid to record other actions not desired, the tool suppose that the user is alone using the system. If other people are using the system at the same time, their actions will be recorded too since the log is for the whole server, and distort the result expected by the analyst.

By catching all the queries applied on the system, a parse is required to analyze them : the goal of this parsing is to find the tables and the columns contained in the queries. It may seem like a simple task, but SQL has a complex grammar. A syntactical analysis of a query can't identify all the columns. A column is identified by its table and the name of the column. The knowledge of the whole schema is necessary. For example, in a join, a column in the select has not necessary the name of the table ahead (since only one of the table contain the name of this column), but only the query is not enough to

identify which table. Moreover, the SQL language add the alias shortcut[3]. A table is represented by an alias in the request. The language SQL allows nested queries : so, a select can be in a select clause, in the from clause (only to form a new table, like the result of a join), or in the where clause.

The following request contain a select clause (a join) in the from clause, and contain aliases.

```
SELECT alias1.col2, B1 FROM table1 as alias1, (SELECT col5 as B1, col2 as
B2 FROM table2, table3 WHERE table2.id = table3.col3) as alias2 WHERE
alias1.col4 = "0";
```

The column in the main select are read once : col2 and B1. The role of the parser is to identify the table and the column. The first one is easy : col2 refers to the table called "alias1", ie "table1". Assuming all the column are named "colX" (where x is a number), the B1 column belong to the table "alias2" . This table is a join. The remaining question here is : which table (table2 or table3) contain the column "col5" called "B1" ? With the SQL request only, this is impossible to answer that question. The knowledge of the all schema is required. The DBMS work as well with the entire schema to clarify this ambiguity.

4.5.2 Upward slicing

The upward slicing technique has as goal to find all elements in the code that refers to a particular table. Indeed in an application the ways of accessing a particular table in a database are numerous. We can either directly use static or dynamic SQL, or even use an ORM layer.
To try and find those elements we wrote a tool that runs like a well known command line, the *grep* command or the eclipse environment search tool, but we added some rules to be more precise than those two other alternatives. Those additional rules consist of searching for the class being the data access object (DAO) representing this table via name transformation.

So the different steps of the upward slicing are the following :

1. Enter one or several input table names.
2. Finding the files in the project mentioning this table.
3. Based on name heuristic we try to find a DAO class.
4. We rerun the search with the name of the DAO class.
5. The results are all files referencing a particular table and its DAO.

[3]You can rename a table or a column temporarily by giving another name known as alias. The use of table aliases means to rename a table in a particular SQL statement. The renaming is a temporary change and the actual table name does not change in the database.

Numerous improvements can be done to this technique. We explore them in Section 9.

Chapter 5

Implementation

We present here the different tools used to implement the techniques presented in the previous section.

5.1 DBMain & JIDBM

Our tools have been developed as extension of the CASE[1] tool DBMain [12], a generic modelling tool dedicated to database application engineering, and in particular to support database design, reverse engineering, re-engineering, integration, maintenance and evolution. We used the Java API JIDBM provided in the DBMain installation to manipulate the .lun files. Each of our tools can be launched without the need of an installed version of DBMain, only the .lun files are needed.

5.2 Foreign key generator

5.2.1 The Mapping File Parser

Our foreign key extractor tool focuses on two main artefacts to raise new referential constraints hypothesis, first the configuration file of the data access layer of the application and second the name analysis of attributes in the schema.
The first technique is described in this section. We have a simple tool that requires as input a file containing the paths of those mapping file (in our test case it reads Hibernate files). Then it proceeds to read all those files and detects the tags declaring a referential constraint. Below is a list of the those tags and how we handled them.

- Many-to-one tags :

```
<many-to-one name="issue" class="org.oscarehr.casemgmt.model.Issue"
    column="issue_id" update="false" insert="false" lazy="false" />
```

Listing 5.1: Tag "many-to-one" in mapping file

[1] Computer Aided Software Engineering

With this tag we get the source attribute of the referential constraint to create, but we don't have the exact name of the target table. Instead we have the name of the java object that represents this table. So based on this name (here Issue) the tool that adds the foreign keys will try to find the exact table name. It's usually the same but with more complex names, some pattern apply (add of "_" between words).

- Sets tags :

```
<set name="notes" table="casemgmt_issue_notes" lazy="true" inverse="
    true">
  <key column="id" />
  <many-to-many column="note_id" class="org.oscarehr.casemgmt.model.
      CaseManagementNote" />
</set>
```

Listing 5.2: Tag "many-to-many" in mapping file

The sets tags are used to represent the many-to-many or one-to-many keys.

- Formula attribute :

```
<property name="programName" type="string" formula="(select p.name
    from program p where p.id = program_no)" />
```

Listing 5.3: Tag "property" in mapping file

Some attributes have the formula attribute, they have a SQL request in them that have to be valid for the attribute.
Those requests can be a sign of a foreign key. The parser here can't define the orientation of the foreign key, it only gives the name of the tables and attributes to the adding tool. It will then check whether the attributes are identifiers or not and create the possible foreign key.

5.2.2 Output files

During the analysis of the property files or during the schema name analysis, uncertainties happened, so our tool produces some text files as output containing potential foreign keys that need to be manually analysed.
We distinguish the following cases :

1. The request in formula tag does not match the specified pattern but could still be a foreign key. In this case the request is written to a text file to be manually analysed.

2. Foreign keys that reference another entity identifier are complex ones (namely *total or equality foreign keys* cf [4]). A lot of potential of those were detected but still have to be validated. We therefore added them to a file to be further analysed.

3. Complex situations where the validation of the constraint depends strongly on the conceptual interpretation were found, those have to be manually analysed.

4. Finally when the analysis is over, there can be remaining attributes that have an "id-like" name. Finishing with sequence such as "id" or "no". We decided to put those in a file too.

In section 6.2.1, we give examples of such scenarios.

5.3 Historical Schema

Our initial (naive) implementation of Algorithm 5 was not sufficiently scalable to analyse long histories of large database schemas in satisfactory time. We therefore implemented a multi-thread version of the algorithm. In this version, an independent thread is used to analyse each distinct table of the schema, and is responsible for iterating through all the schema versions in order to derive the historical information about that table. All parallel threads share a common resource, namely the historical schema, that they can all update when they discover information about their respective tables.

The main program thread iterates over all tables of all successive schema versions. Each time the main thread encounters a table t that does not correspond to a running or terminated thread, it starts a new thread for t. As we will see below, this multi-thread implementation allowed us to significantly improve the efficiency of our historical schema derivation tool.

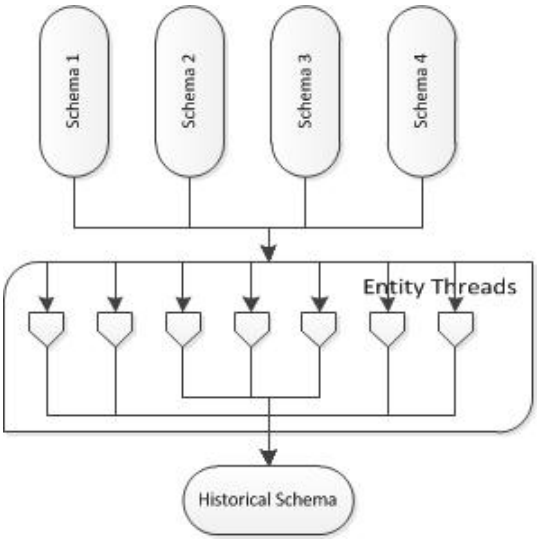


Figure 5.1: Thread Implementation

We analysed the history of the OSCAR (see Section 6) database schema during a period of almost ten years (22/07/2003-27/06/2013). During this period, a total of 517

different schema versions can be found in the project's GitHub repository. The earliest schema version analysed includes 88 tables, while the latest schema version considered comprises 460 tables. When applied to this dataset, our single-thread schema comparison implementation took more than 2 hours[2] to derive the historical schema, while the multi-thread version completed the process in 17 minutes. This constitutes a significant performance improvement.

5.4 Filtering Tool

The filtering tool respect the specification of Section 4.3. the algorithm computing the importance and the closeness is represented by different classes. The interest, the importance and the closeness are represented by meta-attributes of the entity object in DBMAIN.

The tool includes a GUI, where the user can introduce the tables of the focus set and the rejection set. The algorithm can be chosen thanks to this windows.The available algorithms to compute the closeness according to different criterion are :

- *The names* : the closeness is computed according to the names of the tables. The Jaro-Winkler distance is used : it returns a number in the range $[0, 1]$. The distance is computed between the table names of the focus set and the candidate. The equation 4.3 is used.

- *RemoveWhenRemove* finds the tables removed at the same time as at least one of the focus set. If there is no removal date in common between all the remove date of the focus set, and the candidate table, the closeness is zero, otherwise, the closeness is 1. This approach can include an offset : the closeness will be a number in the range [0,1].

- *CreateWhenCreate* : this technique, as the previous one, returns a closeness strict (if there is no offset), and not strict if an offset is defined. Here, only the creation dates are taken into account.

- *CreateWhenRemove* finds the tables created when those from the focus set were removed. Again, if no dates are common between the creation date of the candidate and the removal dates of the focus set, the closeness is 0, and if an offset is defined, the closeness will be in the range [0,1].

- *RemoveWhenCreate* is the opposite of the previous one. We are looking for the tables removed at the same time as at least one table of the FS was created.

- *RemovalVersion* returns a closeness maximal (closeness = 1) if the candidate shares its number of version with at least one table of the focus set, otherwise, the closeness is null.

[2] A computer with with an Intel i5 CPU and 6 Gb of RAM has been used for our test.

- *CreationVersion* : this approach is the same as the previous one, but applied for the number version creation.

These techniques were developed to query the historical schema. For some of them, the specification was not exactly respected. When we talk about dates, it was difficult to transpose the closeness in term of days or month. For the sake of performance, we take one version at regular interval, so we decided to transpose the closeness in term of difference of version. The tool produce a new subschema with the relevant tables in the same dbmain project.

5.5 Database Slicing

The tool we developed about the DBSlicing include the two aspects detailed in Section 4.5.

5.5.1 Downward

This tool is based on the specification set in the section 4.5.1. This tool does not record the request applied on the database at the time they are executed. Instead it analyses the log of the MySQL server for the given period. So, it requires the general query log of the MySQL server to be activated[3] : *The general query log is a general record of what mysqld is doing. The server writes information to this log when clients connect or disconnect, and it logs each SQL statement received from clients.* Note that the activation of this log causes a great loss of performance. This log registers the connections, disconnections, the queries, the changes of preferences, ...

The code 5.4 shows a sample of the General Query Log. The first line shows a kind of "section" : the queries are grouped by time. The following queries are dated 17 december 2012, at 2:05 pm. Multiple queries can appear at particular timestamp also ID (624, 626, 633 and 629 in the example) indicates MySQL connection thread id which has executed a query. The next column is the type of the command that has been executed. It could be "Query", "Connect" or "Init DB". The last column is the body of the command. In case of a query, there will be the query itself (select e.g.). The typical pattern of a log line is :

yymmdd hh:mm:ss thread_id command_type query_body

```
121217 14:05:55    624 Query SET autocommit=0
           626 Connect root@localhost on oscar_12
           626 Init DB oscar_12
           626 Query select round('inf'), round('-inf'), round('nan')
         633 Query SET autocommit=1
```

[3] http://dev.mysql.com/doc/refman/5.1/en/query-log.html

```
633 Query select roleUserGroup,privilege,priority from
    secObjPrivilege where objectName = '_appointment' order by
    priority desc
633 Query select roleUserGroup,privilege,priority from
    secObjPrivilege where objectName = '_site_access_privacy' order
     by priority desc
633 Query select roleUserGroup,privilege,priority from
    secObjPrivilege where objectName = '_team_access_privacy' order
     by priority desc
633 Query select roleUserGroup,privilege,priority from
    secObjPrivilege where objectName = '_appointment' order by
    priority desc
629 Query select 1
629 Query SET autocommit=0
```

Listing 5.4: Example of MySQL General Query Log

Once the log is activated, the user can press "start" on the tool during the execution of the program. The tool will save the timestamp. The same happens at the end of the recording. When the "stop" button is pressed, the tool analyses the MySQL log : it finds all the lines executed during the period.

A first parse is needed to keep only the queries of the period. We only keep the select, delete, update and insert statements. So the set, connect, init db etc will be dropped. Then, a second parse is needed to identify the tables and columns, according to the description made in Section 4.5.1. As already mentioned, the entire schema is required. Hence, our tool take the path to the DBMain project containing the entire schema of the database. During the parsing, we count how many times the database object was read and/or written :

- insert : the elements in the into clause are counted as written once.
- delete : the table in the from clause is written once.
- update : the table (in the update clause), and the columns in the set clause are written once too.
- select : the elements in the select clause are read, such as the from clause.

For all the query types, every database object in a where clause is considered as read. Each object of the resulting schema has two meta-attributes : "Read", and "Write".

To realize the parsing, we use the parser *net.sf.jsqlparser*[4]. Around this parser, we built the TableFinder and the ColumnFinder classes, which identify the database objects from the parsed queries, and in the entire schema.

The resulting subschema is coloured according the meta fields (read and write) :

- brown : it the object is only read

[4]http://jsqlparser.sourceforge.net/

- orange : if the object is only written
- purple : if the object is read and written
- grey : if the element was not used during the recorded execution slice. For example, a column of a used table can be unused.

The resulting schema keeps the foreign key between the objects of the subschema. The dependencies are then clearly marked.

The tool we provide allows to take some screenshots. This could be interesting to map a form (GUI) to the impacted objects. The screenshots can be taken by pushing a simple button. They are then saved in a defined directory.

5.5.2 Upward

The user provides one or several table names and the path to the project directory. Like we said in the design section. This tool was implemented based on the *grep* command. The program then searches a first time for all the files where the table name appears. We then transform the table name, following the name convention used in the domain application and rerun the search. Finally we try to find the DAO class by applying a simple name pattern and search for it also. The output is a navigation tree listing all the files containing either the table name or the dao name. Unfortunately the rules applied are very case specific, they apply to the name patterns used in our case study. We will compare the results of this tool to the search tool of Eclipse in the results section 6.2.5.

5.6 Utility Tools

The section contains all the secondary tools we developed to address technical problems encountered during the development of the techniques presented. We here explain why we developed those tools, and how.

5.6.1 Git Extractor

In order to construct the historical schema, as mentioned in 4.3 the first step is to get the SQL files from the source repository. In our case study (see section 6) the code was hosted on a Git server. Git [14] is a distributed version control system that allows you to have multiple local branches that can be entirely independent of each other. Git works like a filesystem and does not have a version number on each file, each data object has a hash corresponding to a specific commit.

To get a file we first have to get the hashes of the commits and then proceed to recover the file based on the hashes. This has to be done with different command lines.

We then wrote a small program allowing the user to easily retrieve those files. The user can choose to get the records one per week, one per month, one per year or all versions.

- *Input :* The period, the path of the root git repository, the paths of the files in the git repository, the output directory.

- *Output :* The files, the name of which are concatenated with a chronological version number (when all versions are extracted) or with the corresponding date.

5.6.2 SQL Transformation

The resulting SQL code from the extraction does not always fit the extraction tool of DBMain. We therefore need to refactor it. The grammar used by DBMain is a subset of the actual MySQL grammar. Unfortunately the parser does not ignore the word when it does not know it. It skips the entire line or even the entire table definition. We then had to carefully analyse the log of the parser to write a script that allows to modify or delete the problematic words. Here is a list of the detected problems and the solution applied. We recommended the DBMain developers to adapt their parser to the actual SQL grammar.

- *COLLATE* : This word is used to override the default collation. It has no influence on the actual data. We simply removed it.

- *UNIQUE KEY* : The parser does not tolerate the "KEY" keyword after unique. We had to remove it.

- *SET/ENUM* : We removed them and listed them in an external file in order to add them when building the conceptual schema.

- *KEY* : This word is used to declare an index but the DBMain parser only uses INDEX keyword.

- *UNSIGNED* : Deleted.

- *BOOLEAN* : This keyword was replaced by *tinyint(1)*.

- *Length of names :* Once an object with a name of 36 characters is parsed the DBMain application crashes.

5.6.3 Encryption tool

The encryption tool has been written in the context of a case study 6. In order to validate the hypothesis foreign keys discovered thanks to the foreign key generator. We decided to try and validate them based on the data. The data Oscar handles are very sensitive since the consist of medical information. As we were in an application domain which uses sensitive data we had to ensure that the data extracted was encrypted and thus unreadable.

Specification

As an illustration of what the extraction tool does here is a short example: We have taken 3 records of one table, (in order to be readable not all the attributes of that table have been taken). Figure A.1 below represents the data as it is in the database.

last_name	first_name	address	city	province	postal	phone	phone2	email
IDLE	ERIC			ON		905-		
COPPER	COLEEN	55 Applewood	Toronto	ON	A1B1B1	416-123-4567	647-123-4567	coleen.copper@hotmail.com
ANT	ANNA			ON		905-		

Figure 5.2: Example of records in a table

Then you extract the data with the tool and re-inject it into a similar database. Figure 5.3 shows the results you will have for the data given above. The *null* value and the empty fields remain as they are.

Implementation

As input the program needs the database name, its address, the user login and password. It can also take a text file containing a list of table names that will be excluded from the extraction.

Once the connection is made with the database. The program search the *information_schema* to get all the table names of the database to extract. Afterwards it proceeds to remove the tables included in the given list of tables to exclude. Then we iterate over all tables and executes a series of Select statements. The results of those select are row by row and column by column passed in a SHA1 algorithm. The encrypted data is injected into an Insert statement and then written in the output file.

last_name	first_name	address	city	province
7373faf3b43bbcf971b4b9bcf93473b1f374bc71	fafcba79b3f9727ab4317cb1b2fa79bc3979f93a			397a7b7afcbbfc313b3cbbf3bbfa7174baf2fbf3
bbbc3272f93c327973b931bab939bcb931f37bf4	b1f2fcbcbb317ab47cb4b473f33abab271747af1	b4fa73f939bcbc3a3373737abb72b2f4743af2f4	b4f43cf27a3cfc397332ba31b1f3b27971fab174	397a7b7afcbbfc313b3cbbf3bbfa7174baf2fbf3
727b7973b3f4ba323271fcf2343132b9b3323cb9	7bb27cfb7b32f1baf3b17bb371b97af979317ab4			397a7b7afcbbfc313b3cbbf3bbfa7174baf2fbf3

Figure 5.3: Encrypted records of figure A.1

Chapter 6

Case study : OSCAR

In this section we present the results we produced with our tools in a concrete study case. This study case was conducted during our internship at the University of Victoria in Canada. The main objective is to migrate the relational database to a NoSQL model to improve response time. In order to do a migration it is mandatory to have documentation on the current database. Unfortunately this was not available. So our first step was to do a complete database reverse engineering process. Inspired by the classical methodology of DBRE we exploited unused artefacts and brought new techniques to this methodology. The process of database reverse engineering has not been completed (ending with a complete and normalized conceptual schema) but we contributed to recover the schema, improve the readability, help the understanding and eventually improve the quality of the logical schema.

6.1 Context: The OSCAR System

OSCAR (*Open Source Clinical Application Resource*) is full-featured Electronic Medical Record (EMR) software system for primary care clinics. It has been developed since 2001 and is now widely used in hundreds of clinics across Canada. As an open source project, OSCAR has a broad and active community of users and developers.
Development on OSCAR has recently branched out to different centres around the country. In British Columbia, the Universities of B.C. and Victoria have started a collaboration on developing software in support of a primary care research network. The declared purpose of this network is to securely integrate health data from hundreds of clinics for the purpose of answering research questions and improving medical practice. The experiences presented in this section arose in the context of this project and address real world challenges encountered when attempting to comprehend and integrate the OSCAR information system.

6.1.1 OSCAR's Architecture

OSCAR has a Web application architecture following the classical 3-tier paradigm. It employs a Java-based technology stack, making use of Java Server Pages (JSP), Enterprise Java Beans (J2EE) and several frameworks such as Spring, Struts and Hibernate. The source code comprises approximately two million lines of code with a rough distribution of 600 kLOC for the application logic, 1200 kLOC for the presentation layer and 100 kLOC for the persistence layer. OSCAR uses MySQL as the relational database engine and a combination of different ways to access it, including Hibernate object-relational middleware, Java Persistence Architecture (JPA) and dynamic SQL (via JDBC). The reason for this combination of technologies is due to the constant and ongoing evolution history of the product, which originated from JDBC, via Hibernate to JPA.

The OSCAR database schema has over 480 tables and many thousands of attributes. At the time of conducting our study, the database schema of the OSCAR distribution did not contain any information on relationships between tables (foreign keys) and no documentation was available about the schema. We later learned that the missing relationships were due to the evolution history of OSCAR, which has been using the older MyISAM database engine provided by MySQL, which does not support foreign keys.

Other questions about the database schema and its semantics raised during the development of new functionalities, therefore before starting a reengineering process, it was necessary to recover the schema and have a better understanding about it. This will then help to redesign it and eventually improve the performances.

6.2 Results

6.2.1 Foreign Key Extraction

Starting from scratch we extracted the database schema directly from the database using the MySQL dump command. Afterwards we transformed (using the tool presented in section 5.6.2) the code to fit the extractor of DBMain. The resulting schema had 480 entity types, 15829 attributes, with one entity type having 1155 attributes. The output of the SQL extractor is unfortunately unreadable for such a large schema. Figure 6.1 shows the result. And sorting this by hand would have been quite painful and time consuming. The clustering and filtering tools are a major help in reading the schema (see Section 6.2.3).

After having run our foreign key extraction tool, which analyses the names of attributes and the data access property files, we found 440 foreign keys added directly. 23 were added by parsing the hibernate mapping files and 417 by the name analysis. Reason for such low number of keys found in the mapping files is that the migration of the data access layer was under going. To help the user we added a meta-property to each referential constraint providing information about where does this foreign key comes from. Figure 6.2 shows that this key comes from the hibernate file, from which

Figure 6.1: The OSCAR schema, as it can be viewed in DB-MAIN just after the SQL extraction

specific file and even from what kind of tag.

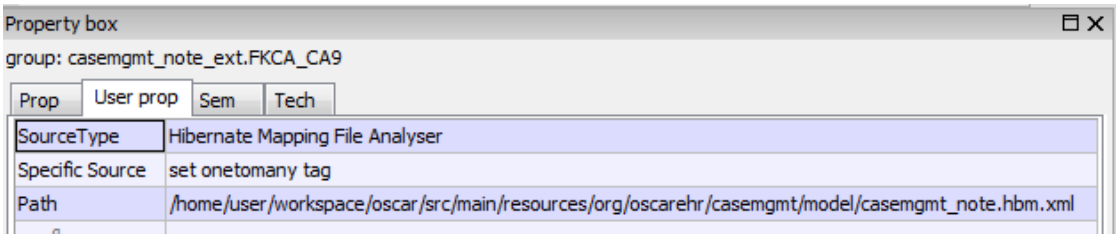


Figure 6.2: The property box of a foreign key

We mentioned in Section 5.2.2 that some of the possible foreign keys were listed in a file to be analysed instead of directly adding them to the schema. We analyse some of those keys below. Below are some entries of the file registering the identifier to identifier foreign keys. It is structured as :

SourceEntity:SourceAttribute:TargetEntity:TargetAttribute

A sample of such a file is shown in the code below.

```
1 demographic:demographic_no:demographicaccessory:demographic_no
  demographiccust:demographic_no:demographicaccessory:demographic_no
3 oscarKeys:name:func:name
  lst_orgcd:code:lst_gender:code
```

We can see with these examples that those keys have great chances of not being one actually. The first one may be a foreign key but extra information are needed, asking a domain expert may be required. The second one is more difficult, we see from the names that it is between 2 peripherals entities of the entity *demographic*. But is there a foreign key between those two? We still have to ask a domain expert. Third and fourth ones indicate that those tables have *name* or *code* has identifier. But should they be connected together? It is most unlikely.
Another file produced is the one containing the keys where the source entity, the source attribute, the target entity and the target attribute did not satisfied the more precise name rules (see Section 4.1).

```
formCESD:demographic_no:demographicaccessory:demographic_no
formchf:demographic_no:demographicaccessory:demographic_no
formConsult:demographic_no:demographicaccessory:demographic_no

waitingListName:name:oscarKeys:name
access_type:name:plugin:name
appointment:name:plugin:name
```

The first three ones have *demographic_no* as source attribute and *demographicaccessory* as target entity. Since it exists a *demographic* entity. It is more likely that the source entity references this one instead of the peripheral entities. The rules implemented also assure that foreign keys between attributes names such as *name* or *code* are not established.

When the analysis is completed we finally list all the attributes that have an "id like" name. Those finishing by "id" or "no" that are neither identifier nor part of a referential constraint. Here are some examples :

```
appointmentArchive:creatorSecurityId
bed:team_id
bill_recipients:billingNo
```

And finally resulting from the parsing of the hibernate mapping files, we produce a file containing the SQL requests that could not be parsed automatically.

```
select count(*) from admission a where a.client_id=demographic_no and a.
    admission_status='current'                and a.program_id in (select p
    .id from program p where p.type='Bed' )
```

All those files contain a lot of possible foreign keys but we thought it was better to analyse those ones manually before inserting them in the schema because of the higher probability that they are not referential constraints after all.

After having discovered and added all the possible foreign keys to the schema. The question about the validation appeared. We indeed were not 100% sure about those keys. To validate the hypothesis implicit constructs we need to prove and disprove them. Several techniques are mentioned by Hainaut in [16]. In our case as the code was several millions of lines and several different patterns were used to access the data, we figured that the simplest technique was to analyse the data.
To do that we would have used the tool *Key Analysis* developed by Rever, which based on a set of foreign keys and a database runs SQL requests to check if those constraints are verified on the data set or not. Unfortunately we were not able to obtain sufficient data in time and therefore only a few keys have been discarded.

However we used the new technique that the historical schema brought us (detailed in Section 3.2). For remind, it consists of checking the date of creation of the target and the source attribute. If the target attribute has been created after the source one we can conclude that this key is not verified.
Applied to the 440 keys we found that 26 could be rejected.

6.2.2 Historical Schema

We analysed the history of the OSCAR database schema during a period of almost ten years (22/07/2003-27/06/2013). During this period, a total of 517 different schemas versions can be found in the GitHub repository.

Once the historical schema was derived, we applied a dedicated procedure allowing to colourize each historical schema object, depending on its age and its liveness. Figure 6.3 shows a colourized version of the historical schema of OSCAR, that has been automatically derived by our tool, and that can be browsed and queried using DB-MAIN. All schema objects depicted in green constitute the tables and columns that are still present in the latest schema version of OSCAR. All red schema objects have been deleted. The color shade corresponds to the age of the objects. A dark red schema object is a table or a column that has been deleted a long time ago. A light red object is an object has been recently removed from the schema. An object depicted in green corresponds to a column or a table that is still present in the latest schema version. The darker the green, the older the corresponding table or column is, and vice versa. A schema object colored in orange is a deleted object that had several lives.

Figure 6.4 zooms on a particular table of the global historical schema of OSCAR, namely the *integratorconsent* table. The table itself is green, which means that it belongs to the latest schema version of OSCAR. It includes 21 columns that are in red, meaning that they have been deleted. The table also contains one deleted column that had several lives, as well as 9 columns that still belong to the latest schema version. Among those 9 *active* columns, 4 are present in the table since its creation, while the 5 other ones have been added more recently. When selecting a given schema table or column of the

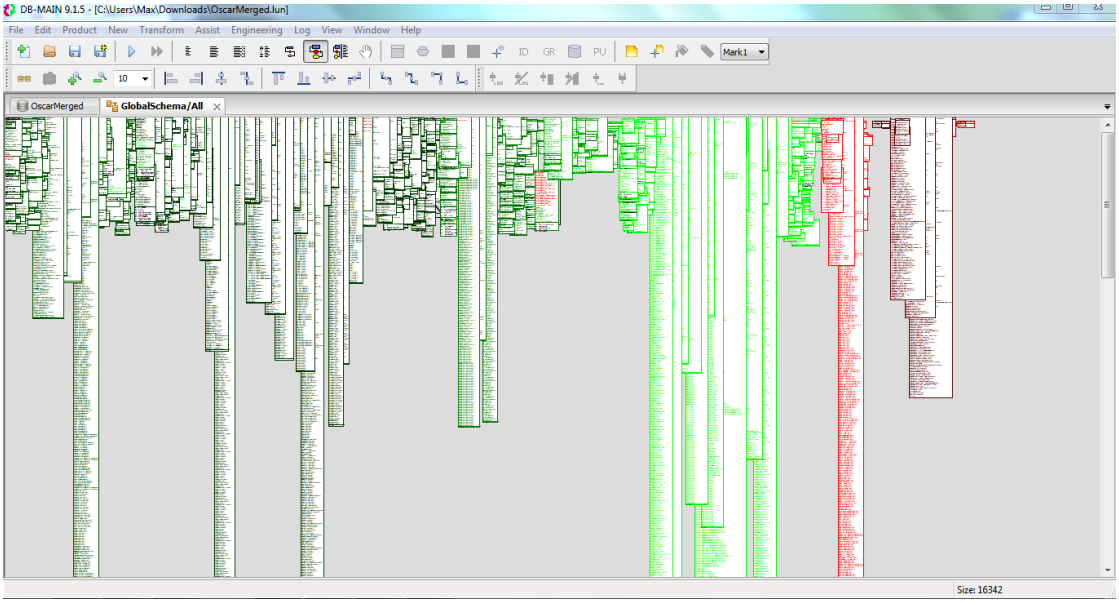


Figure 6.3: The OSCAR global historical schema, as it can be viewed in DB-MAIN.

historical schema, the user may inspect its associated meta-attributes (creation date, number of versions, etc.) via the *property box* of DB-MAIN.

We also provide the user with a historical schema querying tool, allowing the extraction of interesting statistics regarding the evolution of the schema of interest. Some of those statistics for the OSCAR database are given below.

Figure 6.5 depicts the evolution of the number of tables in the OSCAR schema. As expected, this number keeps increasing, due to the lack of table deletion. Indeed, in our test case, people tend to preserve very old tables in order to achieve backward compatibility and avoid the important impact of a schema refactoring on the data and the application programs. From the same figure, we can also easily identify those schema versions that could be considered as "major releases", i.e., those versions where an important number of tables have been added and/or deleted.

Figure 6.6 represents the evolution of the total number of columns in the OSCAR schema. Fortunately, this number follows a similar trend as the evolution of tables. In the last versions we can see that the number of added columns was relatively high compared to the number of tables. This translates the addition of huge tables in the schema, some of which comprising more than a thousand columns. The evolution of the average number of columns per table is quite stable over time (around 30).

Figure 6.7 provides some finer-grained information about the creation and deletion of tables. One can easily notice that the OSCAR tables are not often removed. The expansion of the schema is caused (most of the time) by adding tables, while not replacing or splitting them up. It is indeed very rare when tables are deleted. The total number of deleted tables is around 30, and we can again quickly identify the major release time

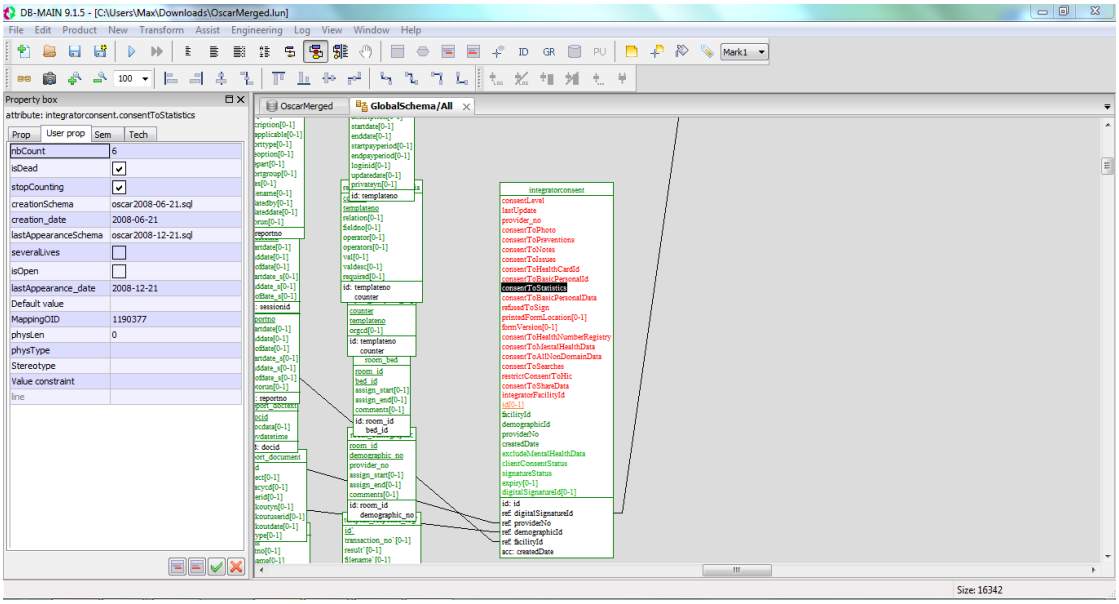


Figure 6.4: The OSCAR global historical schema, when zooming on a particular table.

periods.

The observation is the same for the ratio of created and deleted columns, as shown in Figure 6.8. The number of column creations is, indeed, often greater than the number of column deletions. During the last releases, however, the removal of columns becomes more intensive: 964 columns were deleted between schema version 452 and schema version 453. The explanation is the following: at release 452, huge tables (each including hundreds of columns) have been added to the database, and the peak that we observe originates from the deletion of some of those tables. During our test period, a total of 3 872 columns were removed, while 14 793 columns were created.

In Figure 6.9) we can see the repartition of the tables based on the average age of their columns, expressed in number of versions (X axis) and the number of columns of the table (Y axis). We can notice that the large tables with more than 500 columns are created throughout the whole life of the system. This means that those tables are not an early design problem but are rather is still in use now.

In context of database migration, another interesting property of the tables are their stability. A table that has been created a long time ago, and that was not subject to frequent modifications can be considered as stable. In Figure 6.10 we characterize each table with respect to the number of times it has been modified since its creation, and we relate this information to the age of the table. We can see that the database schema is globally stable, most of the tables have less than 4 modifications. As expected, it is mainly the oldest tables that have the higher number of changes. We can say that a stable table is a good candidate for a migration.

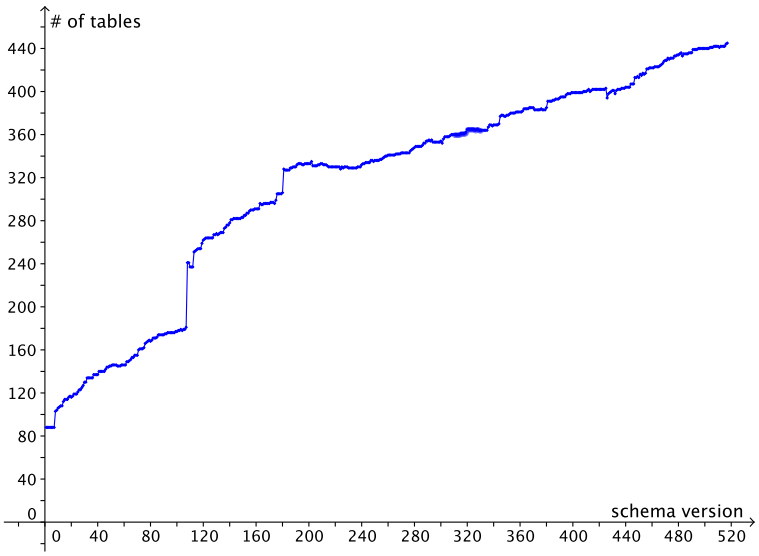


Figure 6.5: Evolution of the number of tables in the OSCAR database.

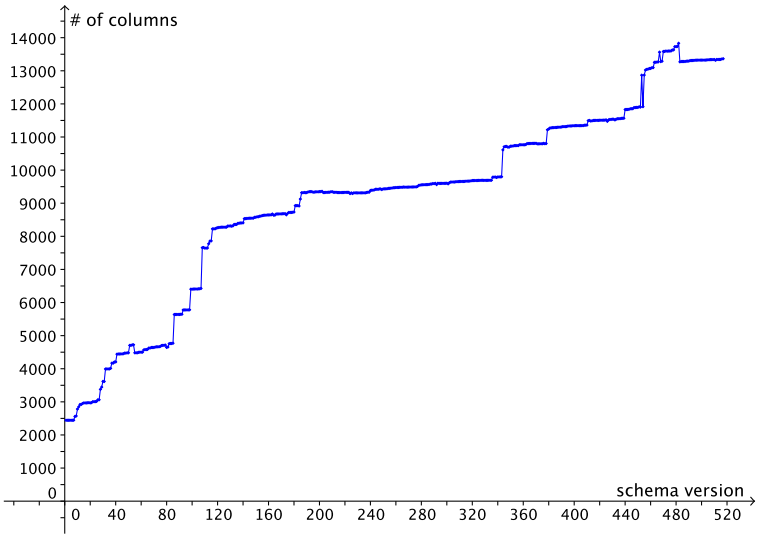


Figure 6.6: Evolution of the number of columns in the OSCAR database.

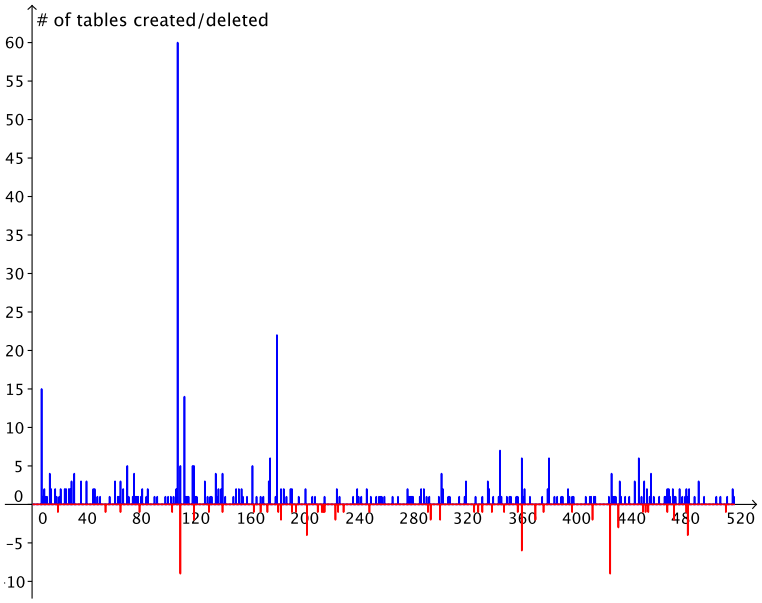


Figure 6.7: Evolution of the number of creation and deletion of tables per version in the OSCAR database.

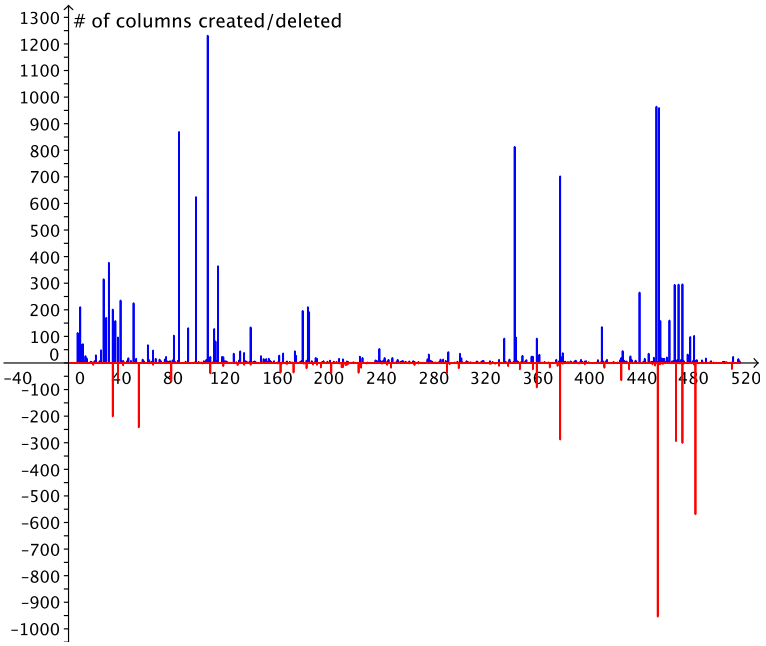


Figure 6.8: Evolution of the number of columns in the OSCAR database.

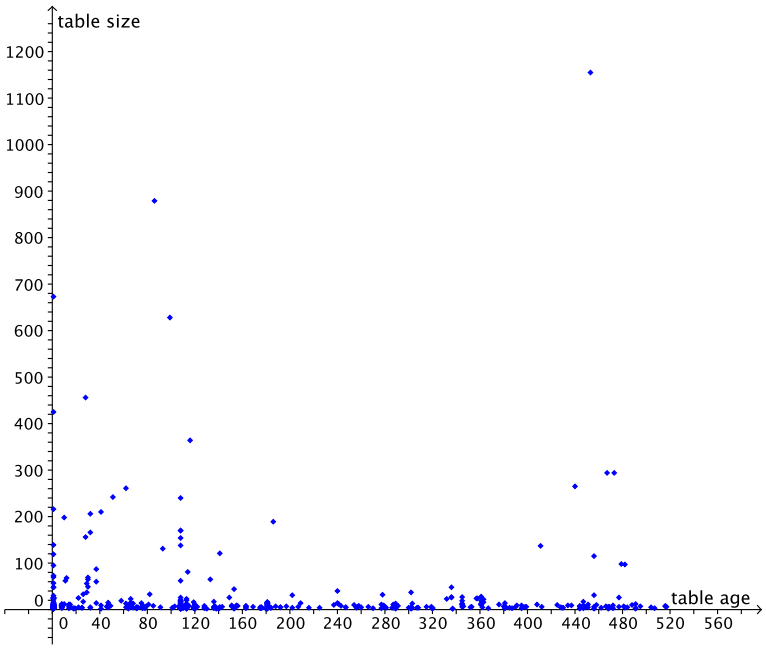


Figure 6.9: Classification of the OSCAR tables in terms of age VS size.

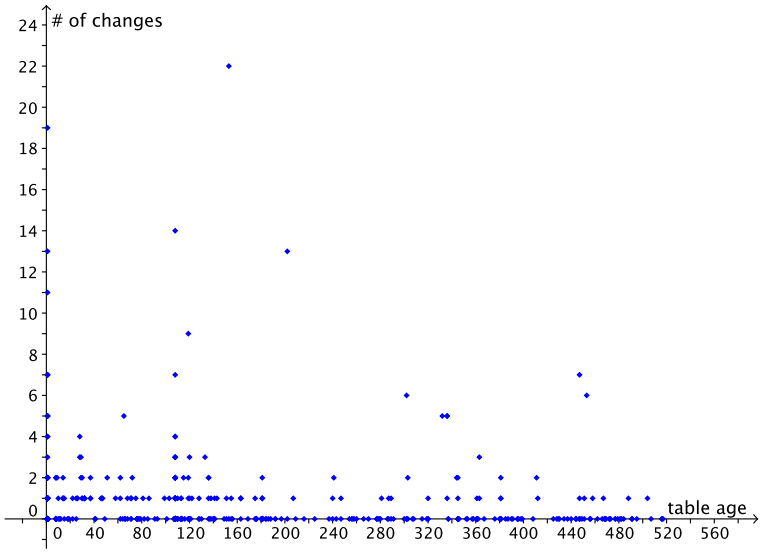


Figure 6.10: Classification of the OSCAR tables in terms of age VS number of changes.

6.2.3 Clustering

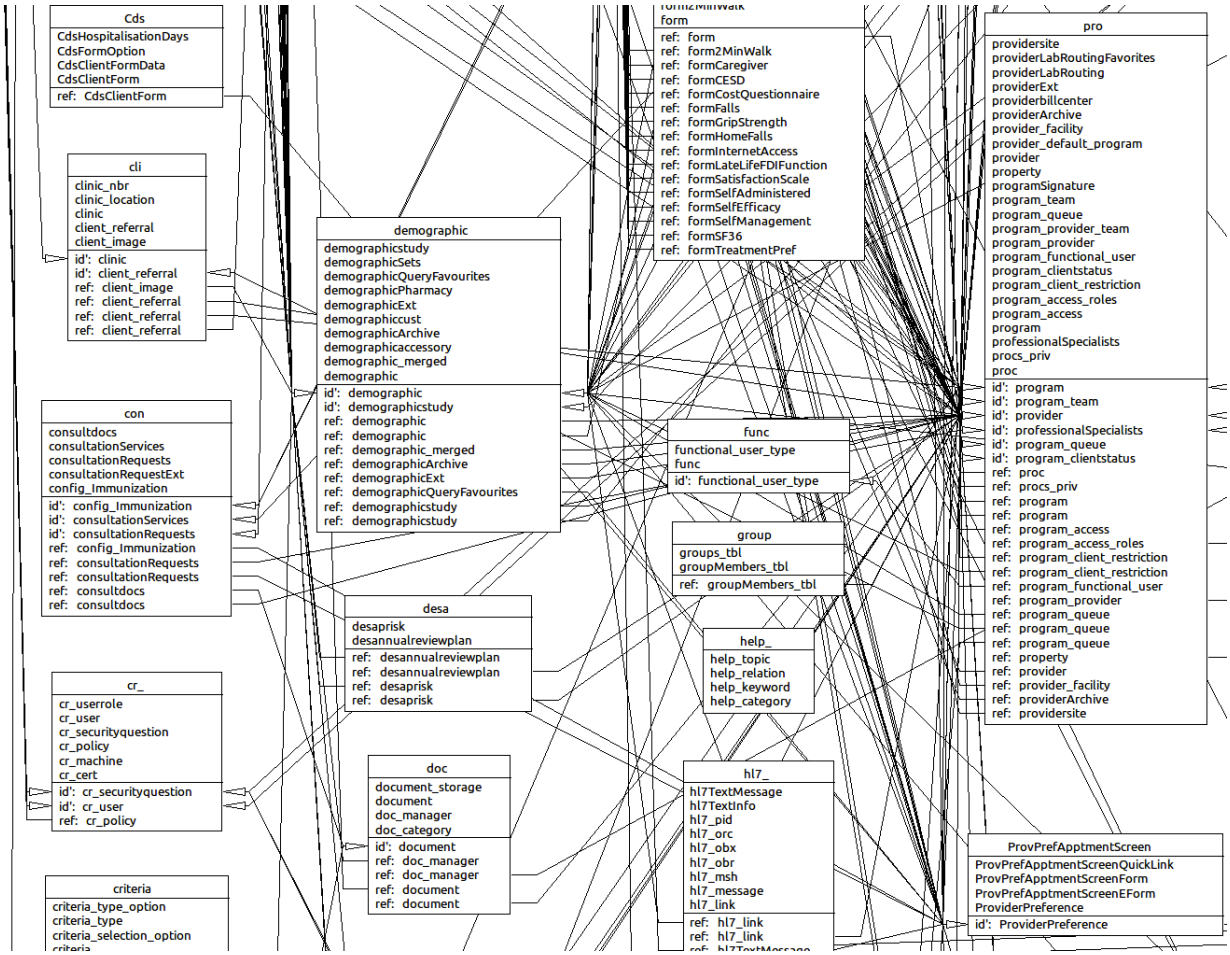


Figure 6.11: Sample of the summarized schema

After extracting the whole schema, the DBMain disposition was not optimal : we could not clearly read the tables (as already exposed in the figure 6.1). We decided to order the schema using the clustering technique. The chosen criterion was the name of the tables, considering that some tables have a prefix. The tool regroups the tables to build clusters, but also place it in a readable way on the DBMain grid. For example, the tables prefixed by "*hl7_*" belong to the HL7 plugin. A plugin which provides standards for interoperability.

In the Jaro-Winkler distance, we put some weight on the tables having a prefix. The tables beginning by "*hl7_*" belong to the same cluster. The conclusion was the same for the prefixes *"HRM", "hsfo_", "ctl_", "cr_"*,... In that case, the name of the cluster is

the prefix itself. The cluster analysis of the OSCAR schema bring 65 clusters, including one containing all the isolated tables (a table forming a cluster only with itself). This particular cluster contain 73 names (of real tables), which must be manually analysed to create other clusters, or to be added to some existing one.

For readability reasons, the complete clustered schema can not be shown in this paper. Even to read the reorganized complete schema, the task was difficult. On the clustered schema, we applied the summarization (section 4.2.2. A sample of the result is shown in the figure 6.11. Remember a real table of the schema is represented by a column, and the tables of this picture symbolize the clusters. The summarized schema confirm that the "*provider*" and "*demographic*" cluster are the most important of the schema. This conclusion make sense because the "*demographic*" table represent the concept of a patient, since the "*provider*" table symbolizes the concept of a doctor.

6.2.4 Filtering

If you want to understand a particular table or a group of the them, you need to look after their relation, their importance in the entire schema, their neighbors, etc. In that case, we try to understand the table *billing*. This table constitutes our Focus Set. The chosen algorithm for the closeness is the naming analysis, and the alpha factor (balance the weight between importance and closeness) is 0.5. The result set size is settled to 10. The filtering tool provide us a sub schema of 10 tables having the meta attributes as illustrated in the table 6.1.

Table name	Closeness	Importance	Interest
billing	1.0	0.098	inf
billingdetail	0.92	0.0	0.4644
billinginr	0.97	0.0	0.485
billingmaster	0.92	0.061	0.495
billingnote	0.95	0.0	0.477
billingservice	0.91	0.02	0.47
billingtypes	0.942	0.02	0.48
billingvisit	0.942	0.0	0.47
demographic	0.41	0.925	0.668
provider	0.42	1.0	0.711

Table 6.1: Filtering Results for the "billing" table

The *billing* table has an infinite interest because it is the Focus Set. We can see the tables containing "billing" in their name have a stronger closeness, since the chosen algorithm use the Jaro-Winkler distance. On the other side, their importance in the schema is weak. This means those tables are not linked by many foreign keys. The 2 last tables have opposite results : indeed, their names are not very closed to the "billing" table, but their importance is huge in the schema. *Demographic* and *provider* are in fact

the real most important table in the schema since they represent key concept in the application domain. This is proved by the high value of their importance : *provider* is actually the most referenced table in the schema.

From the result, we can understand that the billing system has several table representing different concepts such as details, notes, visit, etc. The bills are also linked to the main concept of the entire system : *demographic* and *provider*. Some relation between these tables exists that have to be explained.

With the creation of the Historical Schema, the Filtering querying found another purpose. Indeed, it can be useful to query the historical schema to discover which table were created at the same time, etc. For this example, we want to know the tables that were removed at the same time as *caisi_editor*[1], to try to understand why they have been removed. So we performed the filter with an offset of 3, and a result set of 10 elements. The alpha factor was zero to only take into account the closeness based on the date and not the importance of the table in the schema. The offset of 3 means that we are looking in a period including the 3 previous version and the 3 next version. In this case, the historical schema was made with 1 version per month, so the offset represent 3 months.

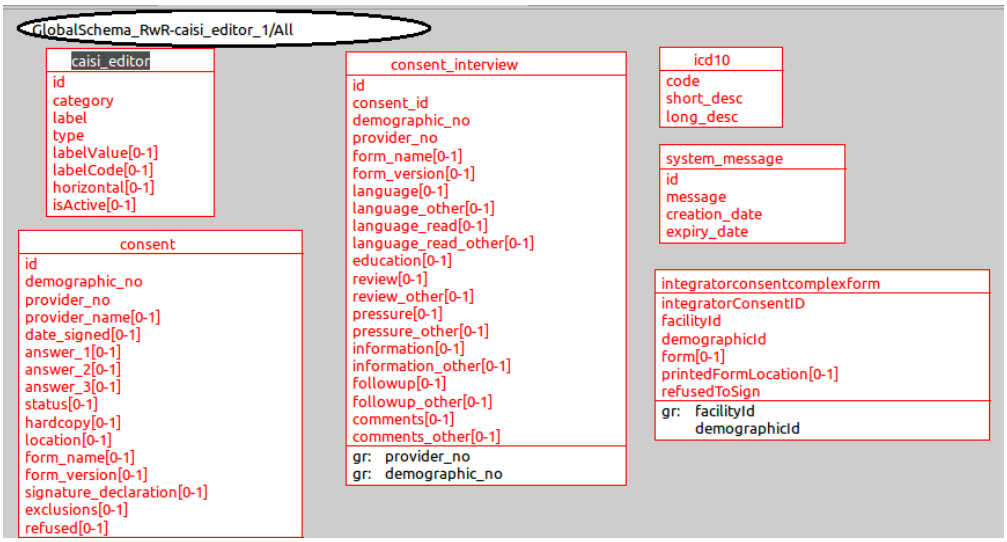


Figure 6.12: Filtering of caisi_editor and system_message

This result subschema is compose by *consent*, *consent_interview*, *icd10*, *integratorconsentcomplexform*, *system_message* and *caisi_editor*, so 6 tables removed between 21/06/2008 and 21/08/2008. Afterwards, we decided to look at the tables created at the same time *caisi_editor* and *system_message* were removed, still with an offset of 3. The result schema is composed of : *caisi_editor*, *system_message*, *favoriteprivilege*, *IntakeRequiresFields*, *integratorconsent*, *integratorconsentcomplexexitintervie*, *integratorcon-*

[1]CAISI means Client Access to Integrated Services and Information.

sentcomplexform, teleplan_response_log and *teleplan_submission_link*.

The table *integratorconsentcomplexform* belong also to that schema : indeed, it was created on the 21/06/2008 and removed on the 21/07/2008. We can assume it was replaced by *integratorconsentcomplexexitintervie* created on the 21/07/2008, and sharing some common columns with *integratorconsentcomplexform*.

6.2.5 Database Slicing

Upward Results : Here we will describe how the upward slicing works.

The different inputs of the program are :

- *Table names* : You enter one or more table names to search. For exemple : *demographicarchive*.
- *Root project path* : The path to the project workspace containing all source code.
- *Directories to exclude* : The names of the directories you want to exclude from the search. For exemple if there is a "test" directory that contains all the classes regarding the unit tests.

When the search is done you end up with an exploring tree containing all the results. You can click on a particular file to open and edit it.

```
[AWT-EventQueue-0 @ FilesFinder.process:36] - Searching files for the table demographicarchive
[AWT-EventQueue-0 @ FilesFinder.findFiles:81] - grep -i -r -w -l --exclude-dir={target,updates,test} demographicarchive /home/user/workspace/oscar/*
[AWT-EventQueue-0 @ FilesFinder.findFiles:85] - 0
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/database/mysql/oscarinit.sql
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/release/oscar10_12_to_Oscar11.sql
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/webapp/demographic/EnrollmentHistory.jsp
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/webapp/demographic/demographiceditdemographic.jsp
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/webapp/demographic/demographicupdatearecord.jsp
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/org/oscarehr/common/dao/DemographicArchiveDao.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:95] - Found corresponding dao : DemographicArchiveDao
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/org/oscarehr/common/model/DemographicArchive.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/oscarDemographic/pageUtil/DemographicExportAction4.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/oscarDemographic/pageUtil/ImportDemographicDataAction4.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/oscarPrevention/reports/PreventionReportUtil.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:81] - grep -i -r -w -l --exclude-dir={target,updates,test} DemographicArchiveDao /home/user/workspace/oscar/*
[AWT-EventQueue-0 @ FilesFinder.findFiles:85] - 0
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/webapp/demographic/EnrollmentHistory.jsp
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/webapp/demographic/demographiceditdemographic.jsp
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/webapp/demographic/demographicupdatearecord.jsp
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/org/oscarehr/common/dao/DemographicArchiveDao.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/org/oscarehr/managers/DemographicManager.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/oscarDemographic/pageUtil/DemographicExportAction4.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/oscarDemographic/pageUtil/ImportDemographicDataAction4.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/oscarPrevention/reports/PreventionReportUtil.java
[AWT-EventQueue-0 @ FilesFinder.findFiles:88] - /home/user/workspace/oscar/src/main/java/oscar/eform/actions/AddEFormAction.java
[AWT-EventQueue-0 @ ValidationAction.actionPerformed:143] - Search done
```

Figure 6.13: Log result of the upward slicing

The log 6.13 shows us all the files found. Compared to the eclipse search tool, we found the .jsp files. Those are useful because they represents where the interaction with the user is made. We could imagine to recreate the user interface based on the results obtained. Moreover we found some other files linked to the *DemographicArchiveDao* class.

The disadvantages from the implementation chosen is that we may get false positive results when the name of the table we are looking for is a common name (for

exemple "Billing"). Indeed those can be just printed on the user interface with no regards to the table.

Downward Results : The downward approach aims to discover the impacted tables of an execution scenario. We expose an example here. When logged as a doctor in the OSCAR system, the homepage is the daily agenda we therefore chose to expose the scenario of adding an appointment.
As shown in the figure 6.14, a form need to be filled to create the appointment. This screenshot was taken during the "recording" phase.

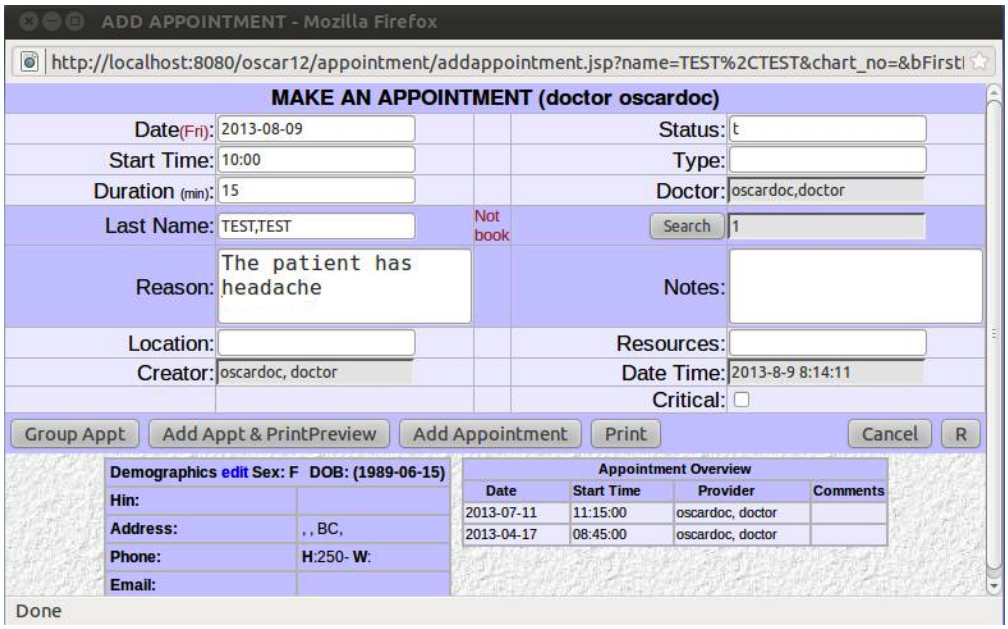


Figure 6.14: Screenshot of the form to add an appointment

Submitting the form will launch some requests on the mysql server. In this case, the subschema resulting of the parsing of those requests is exposed in the figure 6.15. In this subschema, we can find the following tables *admission, appointment, appointment_status, billing, billingmaster, demographic, demographic_merged, demographiccust, demographicstudy, facility_message, health_safety, lst_gender, messagelisttbl, msgDemoMap, mygroup, MyGroupAccessRestriction, oscarcommlocations, other_id, program, property, provider, providerLabRouting, providerPreference, providersite, scheduledate, scheduletemplate, scheduletemplatecode, secObjPrivilege, study, sytemMessage, tickler,* and *waitingListName*. Most columns of those 32 tables were read several times during this scenario. Only some columns of the *appointment* table (e.g. *start_time, end_time, name, demographic_no, program_id, location,* etc) were written, and these are the only ones. The column *appointment_date*, selected in the figure 6.15, was written once and read 16 times.

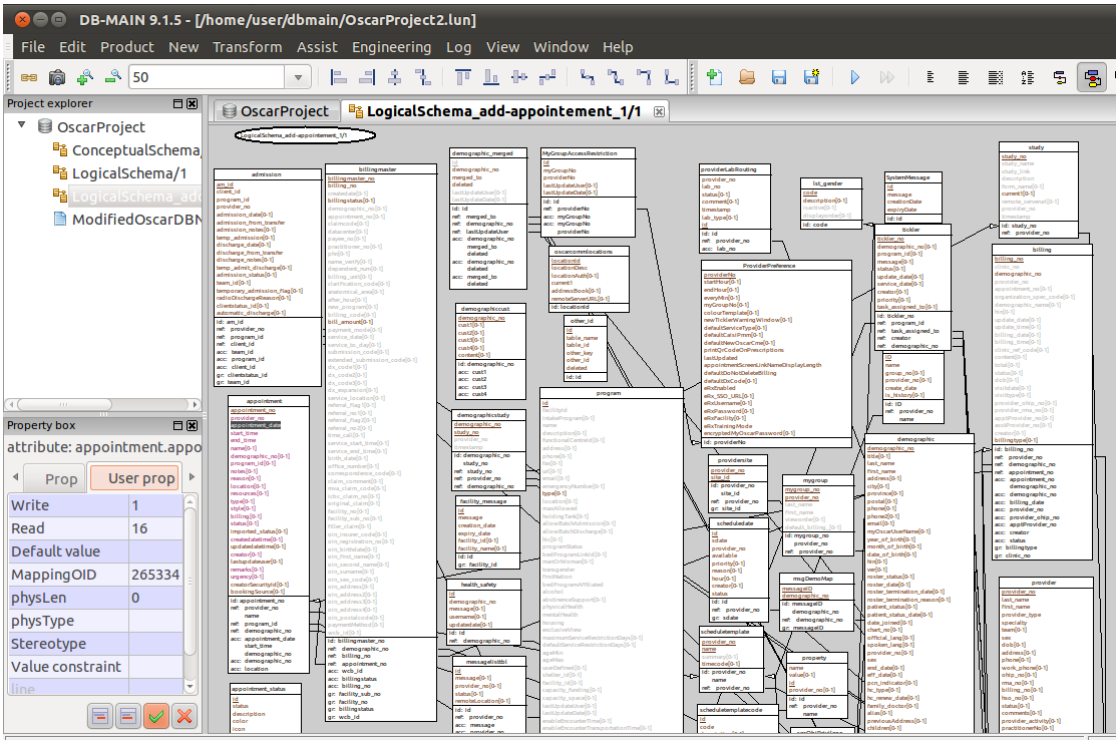


Figure 6.15: Screenshot of the subschema impacted by "adding an appointment"

The parsed queries for this scenario are saved in a log file : it contains 95 queries including the insert query, as shown in the code 6.1.

```
select subrecord0_.merged_to as merged4_0_, subrecord0_.demographic_no
    as demograp2_0_ from demographic_merged subrecord0_ where (
    subrecord0_.deleted = 0) and subrecord0_.merged_to=1
insert into appointment (provider_no,appointment_date,start_time,
    end_time,name, notes,reason,location,resources,type, style,billing
    ,status,createdatetime,creator, remarks, demographic_no,
    program_id,urgency) values('999998','2013-08-09','12:00:00','
    12:14:00','TEST,TEST', '','The patient has a headache','','','',
    null,null,'t','2013-8-9 8:32:14','oscardoc, doctor', '','1','0','')
select count(*) as col_0_0_ from waitingListName waitinglis0_ where
    waitinglis0_.is_history='N' limit 1
select appointment_no from appointment where provider_no='999998' and
    appointment_date='2013-08-09' and start_time='12:00:00' and
    end_time='12:14:00' and createdatetime='2013-8-9 8:32:14' and
    creator='oscardoc, doctor' and demographic_no='1' order by
    appointment_no desc limit 1
```

Listing 6.1: Sample of the parsed queries (Downward Slicing)

The error log produced by the tool is shown in the code 6.2, and contain 2 requests unparsed with our parser, because the sql keyword "regexp" is not recognized by the *net.sf.jsqlparser* (used in the tool).

```
###### Impossible Parsing for : select demographic_no,first_name,
    last_name,roster_status,sex,chart_no,year_of_birth,month_of_birth,
    date_of_birth,provider_no from demographic where regexp '^' and
    patient_status not in ('IN','DE','IC','ID','MO','FI') order by
    last_name, first_name
###### Impossible Parsing for : select * from demographic where lower(
    last_name) regexp '^' order by last_name, first_name
```

Listing 6.2: Sample of the parsed queries (Downward Slicing)

Some other mistakes appear in the eclipse console : 3 tables were not found in the dbmain project. Indeed, *providerpreferenceappointmentscreenform, providerpreferenceappointmentscreenquicklink* and *providerpreferenceappointmentscreeneform* had too long names for the dbmain parser, as already said in the section 5.6.2. But the result (read and written time per database object) about these 3 tables saved in the third log produced by the tool. A sample of this output log is exposed below :

```
secobjprivilege
          objectname                          42   0
          priority                            32   0
          privilege                           32   0
          roleusergroup                       32   0
providerpreferenceappointmentscreenform
          appointmentscreenform               1    0
          providerno                          2    0
```

The first column is the number times read, and the second is the number times written.

Chapter 7

Additional Discussion

When implementing the different techniques described, as we mentioned we worked with DBMain and the API furnished JIDBM. Few experiences were made manipulating such large schemas and thus large DBMain files (.lun file).
To derive the global historical schema each versions of the SQL files had to be added to the DBMain file, we could end up with a file with a size over 900Mb. Opening such a file would take more than 20 minutes. It is when we manipulated this file that we discovered weird behaviour. Some functions of the API (*findEntityType*) didn't work as expected. They should only search in one particular schema and somehow we experienced that the time of execution was related to the size of the file.

Prior to exploiting the different schema version we add to use the SQL Extractor of DBMain, this parser doesn't integrate the complete SQL grammar and doesn't support the too long table names. Names longer than 32 characters caused the DbMain application to completely crash.

Those problems have been shared with the development team and the application is now faster and more stable. As example with a .lun file of 200Mb the opening time was 2 min and 30 sec and the execution of the function was 30 sec. Thanks to the correction of the problems discovered those respective time are now 15 sec and less than a second.

Chapter 8

Conclusion

In the introduction, we presented the purpose of the thesis. We showed that software maintenance is an important step in the software development life-cycle. This step required a good understanding of the program and the database. Databases are more and more crucial in the maintenance process. Unfortunately, documentation rarely exists, we therefore had a strong need to recover it. Research questions of the thesis has been exposed.

In chapter 2, we established the state of the art of the different domain used in this thesis. We saw that the database engineering process has become well documented and rigorous. The database reverse engineering process tend to follow this line, but due to the increasing size of systems and raising of new technologies some limitations were found. The schema evolution is a domain in expansion that currently focuses on the impact of a change and on rather small schema. We proposed to analyse the past of all changes through a tool supported method. To improve the readability we proposed to apply a method of clustering and filtering. We then saw that the database was rarely an objective of the program slicing and was a novative way to explore.

In chapter 3, we proposed a revisited methodology of the database reverse engineering including new artefacts such as the history of code and the mapping files, and new techniques to help the understanding. This method was made to help the process in a case where large database schemas were encountered.

Chapter 4 described the high level specification of our proposed techniques. Chapter 5 is devoted to their implementation.

Finally we exposed the results obtained with the proposed techniques and tools on a large and concrete case study. The OSCAR case allow us to test the developed techniques on a real and complex application. It allowed us to validate, somehow, these tools. We will establish in the next paragraphs if each technique is fully generic and can be applied to any case study or if they are scalable and how. We will then answer the questions asked in Section 1.4.

The technique presented for the extraction of possible implicit constructs, specially **foreign keys** is not fully generic. It has indeed been built regarding the OSCAR appli-

cation context where a specific ORM has been used (Hibernate), in case of other ORM layer used in other application, the tool may not work as the property files may not be the same. The algorithm in charge of the names heuristic has been built using the name conventions used in OSCAR, however those seemed rather generic, and therefore can be applied to other schemas. To perform the validation of hypothesis foreign keys, the encryption tool and the validation developed by Rever are fully generic.

The **clustering technique** aims to order a database schema to make it more readable, or to dispose the element in a way to make the understanding easier for the analyst. This technique is fully generic, since it handle generic database object such as columns and tables. Every database schema can be clustered and summarized. The **filtering** purpose is to extract piece of knowledge about some chosen tables. It generates a subschema containing tables related to the given table according some defined criterion. This technique is not specific to the OSCAR context for the same reasons as the clustering one. These tools are scalable : some new criterion of clustering or filtering can be implemented in an universal way for all database schemas.

The **historical schema** method is fully generic and can be applied easily to other databases. As long as it is possible to extract the SQL code from the source repository. In our case study we developed a stand alone tool 5.6.1 exploiting a Git repository. Rewriting a tool to extract code from other systems such as SVN may be necessary, but the same tool to construct the global historical schema can be used. To provide more results of the historical schema in the written papers we extracted it for the MediaWiki database. The result schema and statistical graphs are shown in Appendix .

The **database slicing** (downward approach) aims to apply the principle of the program slicing. It results a subset of the complete schema containing the objects impacted by an execution scenario. This approach is generic, but was implemented based on the parsing of a MySQL server general log. Some adaptation may be possible to adapt it to other DBMS. The upward approach is based on the grep linux command, so it is not really generic.

This thesis attempts to answer the research questions defined in chapter 1 :

Which new techniques can support the database reverse engineering process?

As mentioned in Chapter 3 & 4. We have seen that besides the usual artefacts and process used for database reverse engineering, we can moreover add new techniques and artefacts. The new artefacts are coming from new technologies (ORM layer) or exploitation of source repositories. The new techniques such as as the database slicing or the clustering come from existing concepts that do not take the database much into consideration. We have found out that adapting such techniques can bring significant in the database reverse engineering process.

How can we support the visualization and comprehension of large database

schemas?

In order to help the visualization of particularly large schemas we introduced the clustering and summarization techniques that elevate the level of abstraction of a schema or regroup tables regarding particular criterion.

To what extent can the history of a database schema help the understanding of its current version?

The historical schema and the filtering tool combined bring important information about which objects were created or deleted at the around same time and therefore allowed us to easily determine which schema structure superseded which other schema structure. Another important insight created by our schema evolution analysis method was a better understanding of the role of the many large-scale tables that contain hundreds or even thousands of attributes. The information content of these tables overlaps significantly with other parts of the transactional database. Therefore, our initial hypothesis (before considering the evolution history) was that these tables were relicts of early database designs. The evolution analysis, however, refuted this hypothesis and indicated that these tables are indeed being generated throughout the system lifetime. Numerous other applications can be added and are detailed in chapter 9.

The historical analysis brought to light the link between the evolution of the database and the evolution of all the other software artefacts, and that this domain remains largely unexplored. This work is the subject of a PhD thesis recently started by Loup Meurice.

Chapter 9

Future works & Applications

During the conception of the presented new techniques, a lot of ideas came to our minds. Some of them are in the continuity, further exploitation, others are new techniques that could also contribute to the understanding of large database schema. We tried to implement a subset of these techniques in our case study but several problems appeared due to the specific environment or missing artefacts. This section presents those ideas.

9.1 Advanced Exploitation of Historical Schema

Regarding the Historical Schema since it is a new technique. A lot of further work can be done. We first can add numerous attributes to each data objects. Indeed we only used the basic information provided by the version repository. Git or SVN provide other useful information that could be integrated to the schema for example the developers manipulating an object. Integrating the history of each person who have contributed to a particular table could help us on one hand to identify who is responsible of a particular change, and on the other hand help the reverse engineer to find the person who is expert with this table. Improving the filtering tool could then help answering those questions. If the application context integrates migration scripts between each version of the schemas, those can also be integrated to the historical schema. Thanks to the version repository the source code of the application itself can be linked to the evolution of the database schema. And therefore we could imagine discovering recurring patterns and eventually developing automated scripts migration based on a schema evolution. Answering question like *What changes in the application code if a table or a column is added?* The co-evolution of code application and database schema has been the subject of [23] where they present the main challenges and possible solutions to explore.

Another major improvement will be to develop a proper visualisation tool of this historical schema, likewise the work of [32], we could imagine that the tables would be represented has *buildings* and the different clusters would be *districts*, and following the importance of changes or of relations between them we would have bigger elements.

One could also imagine having a dynamic representation of the evolution of the schema, we could via a cursor navigate throughout all the history of the whole schema.

One further and ambitious development in the evolution domain would be to create a representation of the evolution of the data instead of the structures. Data evolution would help us discovering implicit constraint, intensity of use, dead structures and therefore improve the performance of the database.

9.2 Enriched Database Slicing

The database slicing can be improved by integrating the two approaches and providing a user friendly visualisation tool. The user does a particular action and when it is done we will have a visualisation tool integrating the code corresponding to the action linked to the user interface, to the SQL executed and finally to the tables affected by this action. A way to do that would be to analyse the trace of client browser such as GUI events using FireDetective[1].

In case of long sequences of actions for the user to perform. The idea was to use a crawler such as Crawljax [2]. The work of [24] explores a technique to automatically test ajax based web application by generating and analysing all the different states of the DOM tree. We would have used the crawler to automatically do a specific series of action. A further application of such a crawler is to execute the whole application in addition of the downward slicing tool. This way would obtain a full database schema of the used tables. Comparing this schema with the whole schema and we obtain the possible *Dead Structures*.

Finding the *Dead Structure* is a major problem in database re-engineering. Some can be discovered using a combination of our techniques, the historical schema with the filtering tool and data. But another solution would be to use the upward database slicing in combination with a tool such as CodePro [3], this tool can detect the dead code in an application. Combining the dead code detection with the classes discovered thanks to the slicing could also give us good clues about the dead structure in a schema.

9.3 Extended clustering and Filtering

As said above, the database objects can contain several other attribute, or more precisely meta-attribute, such as the developer names, a particular color, etc. All these can be new criterion for the clustering and the filtering. New algorithms may be made based on the new characteristics, to extend the existing tool. However, a tool able to fill these meta-attributes from the Git repository or other sources (documentations, rapports, etc) is necessary.
The filtering can be improved. Indeed the existing algorithm about creation and removal

[1] http://swerl.tudelft.nl/bin/view/Main/FireDetective
[2] http://crawljax.com/
[3] https://developers.google.com/java-dev-tools/codepro/doc/

dates compute the closeness takes into account the difference between versions, we could adapt it to consider the times in term of days or weeks.

Bibliography

[1] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, November 1993.

[2] B.W. Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, SE-10(1):4–21, 1984.

[3] A. Cleve and J-L Hainaut. Dynamic analysis of sql statements for data-intensive applications reverse engineering. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 192–196, 2008.

[4] A. Cleve and J-L Hainaut. What do foreign keys actually mean? In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 299–307, 2012.

[5] A. Cleve, J. Henrard, and J-L Hainaut. Data reverse engineering using system dependency graphs. In *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*, pages 157–166, 2006.

[6] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 2014. to appear.

[7] Anthony Cleve, Jean-Roch Meurisse, and Jean-Luc Hainaut. Journal on data semantics xv. chapter Database semantics recovery through analysis of dynamic SQL statements, pages 130–157. Springer-Verlag, Berlin, Heidelberg, 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 IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 297–321. Springer Berlin Heidelberg, 2013.

[9] 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 IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 297–321. Springer Berlin Heidelberg, 2013.

[10] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[11] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. In *Very Large Data Base (VLDB)*, 2008.

[12] DB-MAIN. The DB-MAIN official website. http://www.db-main.be, 2010.

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[14] Github. The Git official website. http://git-scm.com/, 2010.

[15] Maxime Gobert, Jerome Maes, Anthony Cleve, and Jens Weber. Understanding schema evolution as a basis for database reengineering. In IEEE Computer Society, editor, *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013)*, 2013. to appear.

[16] Jean-Luc Hainaut. *"Introduction to Database Reverse Engineering"*. 2002.

[17] Jean-Luc Hainaut. The transformational approach to database engineering. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 95–143. Springer Berlin Heidelberg, 2006.

[18] J.L. Hainaut. *Bases de données: Concepts, utilisation et développement*. Sciences sup. Dunod, 2009.

[19] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, September 1984.

[20] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, pages 20–, Washington, DC, USA, 1997. IEEE Computer Society.

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

[22] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. *SIGPLAN Not.*, 25(10):67–76, September 1990.

[23] Dien-Yen Lin and Iulian Neamtiu. Collateral evolution of applications and databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 31–40, New York, NY, USA, 2009. ACM.

[24] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Erhard Rahm and Philip A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.

[26] Spencer Rugaber. Program comprehension, 1995.

[27] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[28] Dag Sjøberg. Quantifying schema evolution, 1993.

[29] Antonio Villegas and Antoni Olivé. On computing the importance of entity types in large conceptual schemas. In CarlosAlberto Heuser and Günther Pernul, editors, *Advances in Conceptual Modeling - Challenging Perspectives*, volume 5833 of *Lecture Notes in Computer Science*, pages 22–32. Springer Berlin Heidelberg, 2009.

[30] Antonio Villegas and Antoni Olivé. A method for filtering large conceptual schemas. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson Woo, and Yair Wand, editors, *Conceptual Modeling – ER 2010*, volume 6412 of *Lecture Notes in Computer Science*, pages 247–260. Springer Berlin Heidelberg, 2010.

[31] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[32] 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.

[33] D. Willmor, S.M. Embury, and Jianhua Shao. Program slicing in the presence of database state. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 448–452, 2004.

Appendices

Appendix A

MediaWiki Historical Schema and statistics

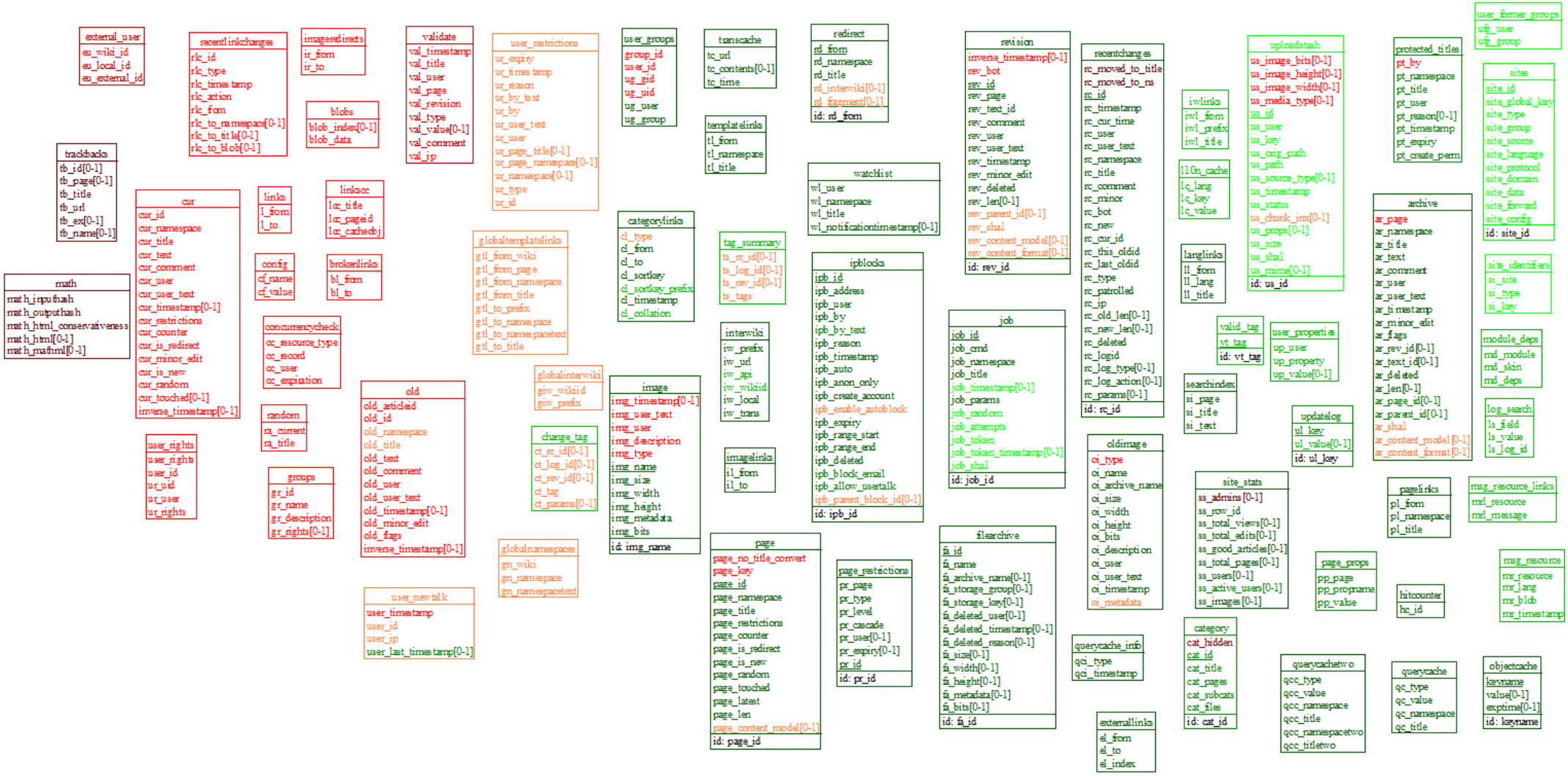


Figure A.1: Historical schema of MediaWiki

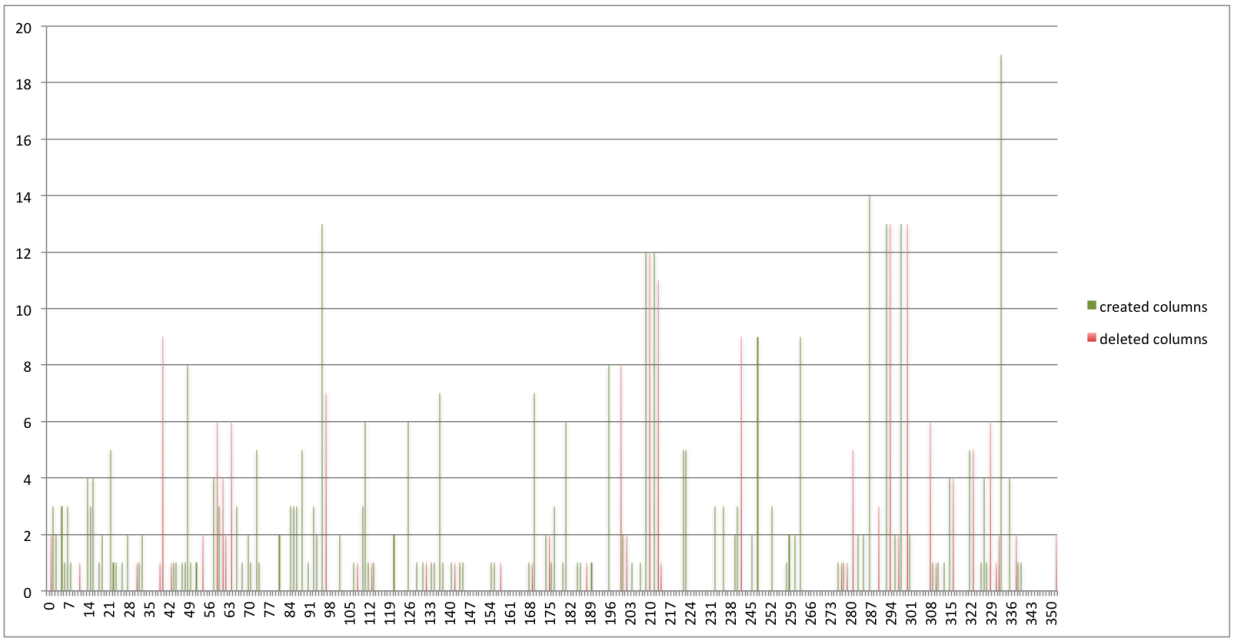


Figure A.2: Column modification of MediaWiki per version

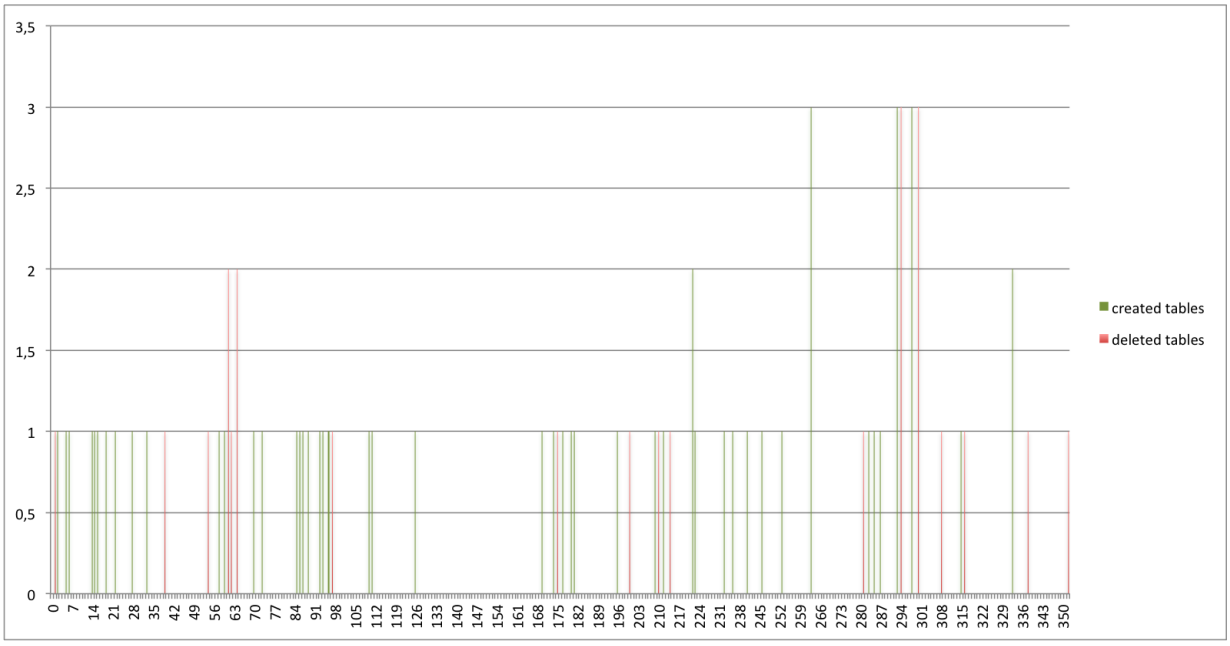


Figure A.3: Table modification of MediaWiki per version

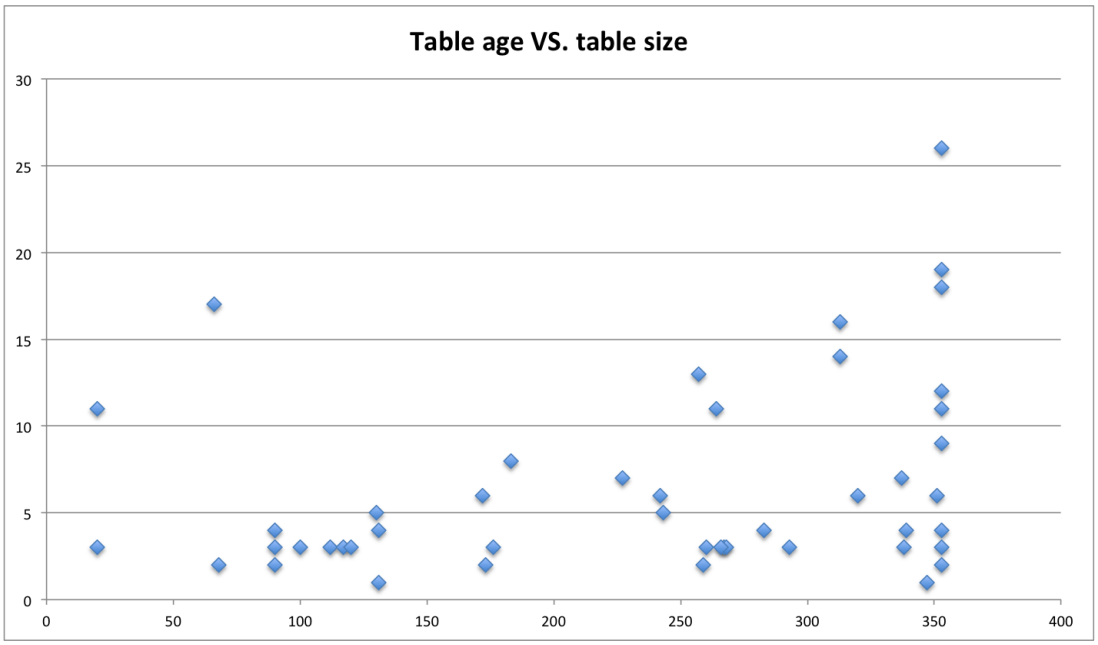


Figure A.4: Table age vs table size (MediaWiki)

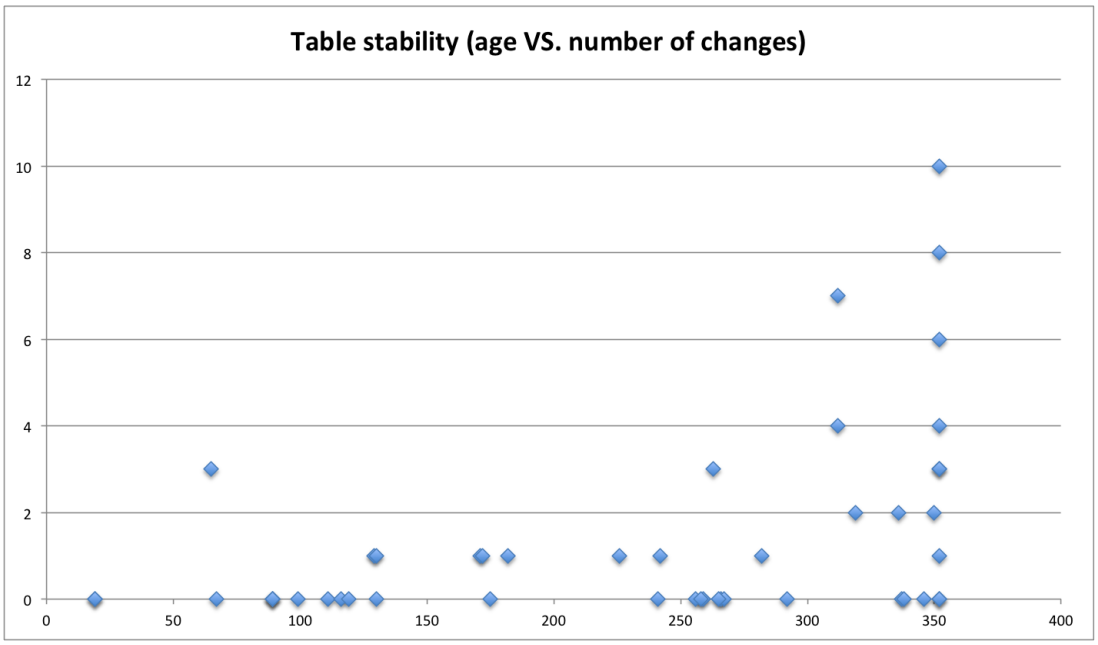
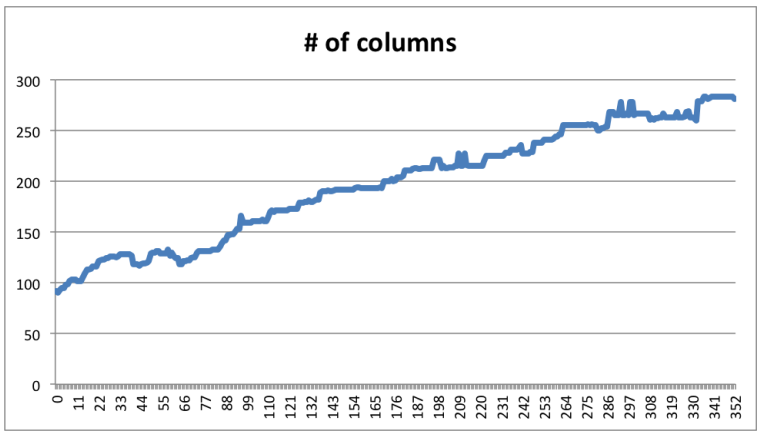
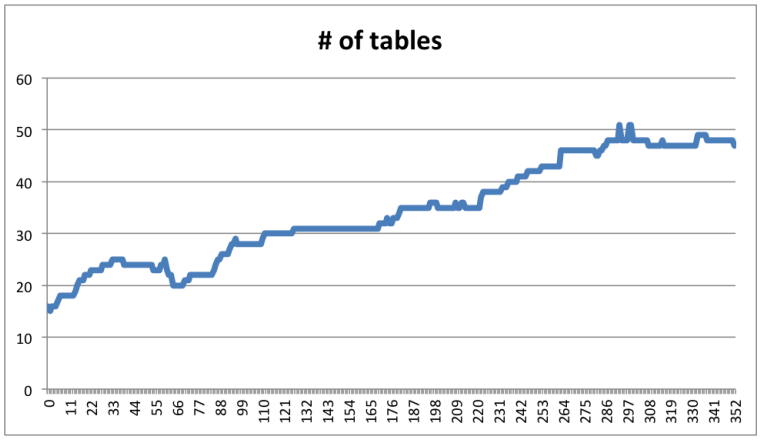
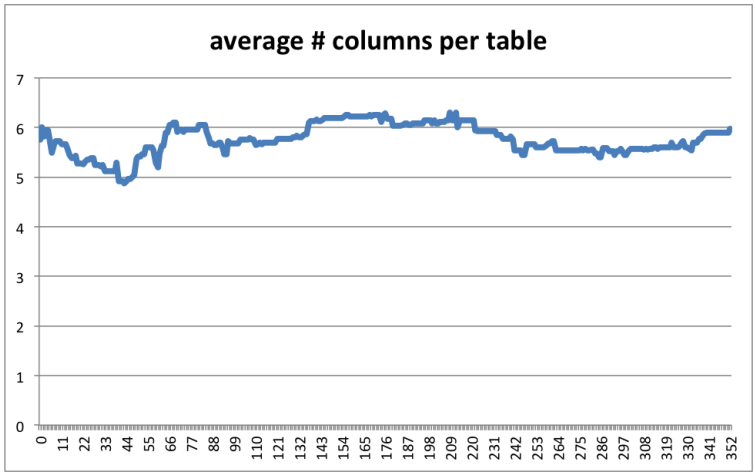


Figure A.5: Table stability (MediaWiki)


# of columns
300
250
200
150
100
50
0
0 11 22 33 44 55 66 77 88 99 110 121 132 143 154 165 176 187 198 209 220 231 242 253 264 275 286 297 308 319 330 341 352


# of tables
60
50
40
30
20
10
0
0 11 22 33 44 55 66 77 88 99 110 121 132 143 154 165 176 187 198 209 220 231 242 253 264 275 286 297 308 319 330 341 352


average # columns per table
7
6
5
4
3
2
1
0
0 11 22 33 44 55 66 77 88 99 110 121 132 143 154 165 176 187 198 209 220 231 242 253 264 275 286 297 308 319 330 341 352