Managing structural and behavioral evolution in relational database: Application of Software engineering techniques

Julien Delplanque

Université de Lille

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Managing structural and behavioral evolution in relational database:

Application of Software engineering techniques

Gérer les évolutions structurelles et comportementales des bases de données relationnelles:

Applications des techniques d'ingénierie logicielle

THÈSE

présentée et soutenue publiquement le 28 Septembre 2020

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Julien Delplanque

Composition du jury

Président :	Kathia Oliveira	(Professeur - Université Polytechnique des Hauts-de-France)
Rapporteurs :	Olivier Barais	(Professeur - Université de Rennes 1)
	Anthony Cleve	(Professeur - Université de Namur)
Examinateur :	Tom Mens	(Professeur - Université de Mons)
Directeur de thèse :	Anne Etien	(Professeur - Université de Lille)
Co-Directeur de thèse :	Nicolas Anquetil	(Maître de Conférences - Université de Lille)

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe

Remerciements

Je remercie Anne Etien et Nicolas Anquetil pour avoir été mes superviseurs de thèse. Je remercie les membres du jury Olivier Barais, Anthony Clève, Tom Mens et Kathia Oliveira pour leurs relectures et commentaires. Je remercie Stéphane Ducasse pour m'avoir accueilli au sein de l'équipe Rmod et pour nos différentes collaborations en dehors de la thèse.

Je remercie Olivier Auverlot pour le précieux temps qu'il m'a accordé, pour sa participation aux diverses expériences, pour son aide et pour les différentes pauses café qui ont forgées notre amitié.

Je remercie l'ensemble des personnes que j'ai rencontré à Rmod durant la thèse. En particulier, merci à mes camarades Vincent Aranega, Clément Béra, Vincent Blondeau, Christophe Demarey, Thomas Dupriez, Cyril Ferlicot, Christopher Fuhrman, Pavel Krivanek, Guillaume Larchevêque, Pierre Misse, Damien Pollet, Benoît Verhaeghe et Oleksandr Zaitsev pour les nombreuses discussions très intéressantes en rapport de près ou de loin avec ma thèse et pour les liens d'amitié que nous avons créés durant ces trois années. Chacun d'entre eux, à sa façon, m'a aidé à aller jusqu'au bout de cette épreuve.

Je remercie ma famille pour tout ce qu'ils ont fait pour moi jusque maintenant et pour tout ce qu'ils feront encore, j'en suis sûr, dans le futur. En particulier, je remercie mes parents pour les valeurs qu'ils m'ont transmises. Finalement, je remercie ma fiancée, Émeline, pour son soutien, sa patience et sa présence chaque jour à mes cotés.

Abstract

Relational databases have been at the core of many information systems for decades and continue to be used in new software development. Many of these databases reflect human or societal activities, for example, processes related to human resources, insurances, banks, etc. Reflecting such activities induce frequent evolutions of both the software system and the relational database. Relational databases do not only store and ensure data consistency, they can also define behavior taking the form of views, stored procedures, triggers, etc.

Implementing behavior directly inside a database has the advantage to prevent code duplication when multiple programs using it perform similar tasks. However, the evolution of such database becomes complex and few approaches in the literature address this problem. Most of the literature addressing relational database evolution focus either on the evolution of the database schema or on its co-evolution with software interacting with it. Approaches to reverse-engineer and evolve both structural and behavioral entities of relational databases are missing.

In this thesis, we address this gap in the literature with four main contributions: (i) we report our observation of a relational database evolution made by an architect and identify problems from our observations; (ii) we propose a meta-model representing both structural and behavioral parts of a database and simplifying dependencies analysis; (iii) we propose a tool to find quality issues in a database schema; and (iv) we propose a semi-automatic approach to evolve a relational database (its structural and its behavioral parts) based on recommendations that can be compiled into a SQL script.

The results of the research reported in this thesis provide building bricks for the development of behavior-aware integrated development environment for relational databases.

Keywords: relational databases, behavior-aware meta-model, reverse-engineering, impact analysis, recommendations

Résumé

Depuis plusieurs décennies, les bases de données relationnelles sont au coeur de nombreux systèmes d'information et continuent à être utilisées lors du développement de nouveaux logiciels. Beaucoup de ces bases de données reflètent des activités humaines ou sociétales. Parmi ces activités, on peut citer les processus liés aux ressources humaines, aux assurances, aux banques, etc. Etant en constante évolution, elles induisent des évolutions fréquentes du logiciel et de la base de données relationnelle qui lui est associée. D'autre part, les bases de données relationnelles ne se contentent pas de stocker et d'assurer la cohérence des données. Elles permettent également de définir du comportement pouvant prendre la forme de vues, procédures stockées, triggers, etc. Ce comportement, quand il est défini directement à l'intérieur d'une base de données, présente l'avantage de réduire la duplication de code lorsque plusieurs programmes utilisant celle-ci effectuent des tâches similaires. Cependant, l'évolution de cette base de données est rendue plus complexe et peu d'approches traitent ce problème dans la littérature scientifique. La plupart des articles traitant de l'évolution des bases de données relationnelles portent soit sur l'évolution de leurs schémas soit sur leurs coévolutions avec les logiciels interagissant avec elles.

Dans cette thèse, nous répondons à ce manque via quatre contributions principales : (i) nous rapportons notre observation de l'évolution d'une base de données relationnelle et identifions les problèmes apparents durant celle-ci ; (ii) nous proposons un méta-modèle représentant à la fois les entités structurelles et comportementales d'une base de données et qui simplifie également l'analyse de dépendances ; (iii) nous proposons un outil pour trouver les problèmes de qualité dans un schéma de base de données ; (iv) nous proposons une approche semi-automatique pour faire évoluer une base de données relationnelle (en incluant ses entités structurelles et comportementales) via des recommandations qui peuvent être compilées dans un script SQL.

Les résultats présentés dans cette thèse sont utiles à la construction d'un environnement de développement intégré pour les bases de données relationnelles. Tout ceci en prenant en compte les entités structurelles et comportementales.

Mots-clés: bases de données relationnelles, meta-modèle, retro-ingénierie, analyse d'impact, recommandations

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Contents

In this thesis, we address relational database evolution from a software engineering perspective. This chapter sets the context of the thesis, lists problems we identified and introduces our approach and contributions.

1.1 Software Engineering

Software Engineering emerged from a need to rationalize the way software is built. The term was used for the first time in 1968 as the title of a conference organized by NATO [Naur 1969]. This research field aims to provide techniques and tools to help developers during the development and maintenance of their software as highlighted by the following definition.

Software Engineering: *"The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."* [Naur 1969]

After its deployment, software is subject to changes. At the beginning of Software Engineering research field, the scientific community thought that those changes were only bug fixes or minor enhancements. This intuition led to the waterfall process of software development proposed by Royce [Royce 1987]. The waterfall process spread in industrial practices. However, the scientific community later realized that it is possible that software require major changes to adapt to new

requirements or to improve quality. This phenomenon is called *software evolution* and was studied and theorized by Lehman [Lehman 1980] in the late seventies.

Two main views of software evolution have been identified [Lehman 2000] as stated by Mens [Mens 2008]:

- the "what and why" focusing on understanding the nature of software evolution phenomenon; and
- the "how" focusing on more pragmatic aspects of software evolution phenomenon, aiming to help in day to day development of software.

In this thesis, we mainly address the later dimension in the context of relational databases: the "how".

Software evolution is complicated to handle. Software can strongly depend on real-world activities because it is part of these activities (think about software related to human resources, insurances, banks, etc.). Such software, called *E-type software*, need to be adapted often to follow the evolution of requirements.

E-type software: *"Programs that mechanize a human or societal activity."* [Lehman 1980]

Some properties of E-type software, listed in the famous "Laws of software evolution" [Lehman 1980], are found to be hard to address. Namely, the complexity keeps increasing if nothing is done to counter this phenomenon (law II), the functional content keeps increasing to satisfy user needs (law VI) and the quality keeps decreasing if nothing is done to counter this phenomenon (law VII). A lot of energy is required from developers to counter those phenomenons.

To tackle these E-type software properties, re-engineering emerged. Re-engineering is a process composed of three activities identified by the horseshoe process [Kazman 1998] (see Figure 1.1).

First, reverse engineering helps developers to understand existing software systems and to abstract the way it works leading to good mental models. A good mental model is critical for further development. Second, software is re-structured to improve it and/or add a functionality. Third, a new version of the software is built to replace the original one.

This thesis is related to software re-engineering. Thus, the horseshoe process is used by our approach as it is the standard way to implement software re-engineering.

1.2 Relational databases

Relational model: Relational databases are databases complying to the relational model as introduced by Codd in the seventies [Codd 1970a]. The relational model describes how to organize data using a few fundamental concepts:

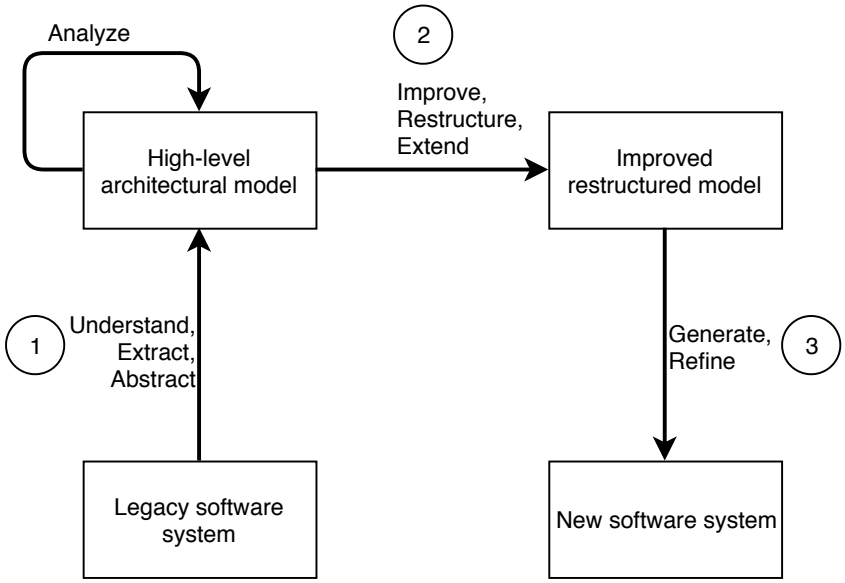

Figure 1.1: Horseshoe process, inspired from Figure 1.3 in [Mens 2008].

- Tables are composed of a set of attribute and domain pairs (named the table's schema) and a set of rows containing data.
- Attributes are named columns of a table, the name of an attribute must be unique across all attributes of the table.
- Domains (or data types) set constraints on data to be stored.
- Integrity constraints are assertions concerning data. They need to be true at any time. Codd's relational model introduces two kinds of integrity constraints: the primary key which ensures that the values of a set of attributes are unique in the table and the foreign key allowing one to specify that the set of values taken by a set of attributes is a subset of the set of values taken by another set of attributes potentially located in another table and constrained by a primary key.
- The database schema is the set of tables and integrity constraints present in a database. It describes how data are structured overall in the database.

Relational database: The storage of data is called the relational database. Basically, it contains data and the database schema. Users never interact directly with the database in order to prevent data corruption. Instead, interactions between users and the database are made through the relational database management system. The latest ensures that data stored in a relational database stay consistent.

Relational database management system: The software managing a database is called the Relational DataBase Management System (RDBMS). Its purpose is to manage one or many relational databases. This management consists of multiple tasks:

- Manage users and permissions (one can control users' access to data).
- Create, delete, or alter the schema of a database.
- Parse, analyze and execute queries, a query being able to either read or modify data.
- Ensure that integrity constraints are always fulfilled when a query is executed. If a constraint is violated, the RDBMS raises an error.

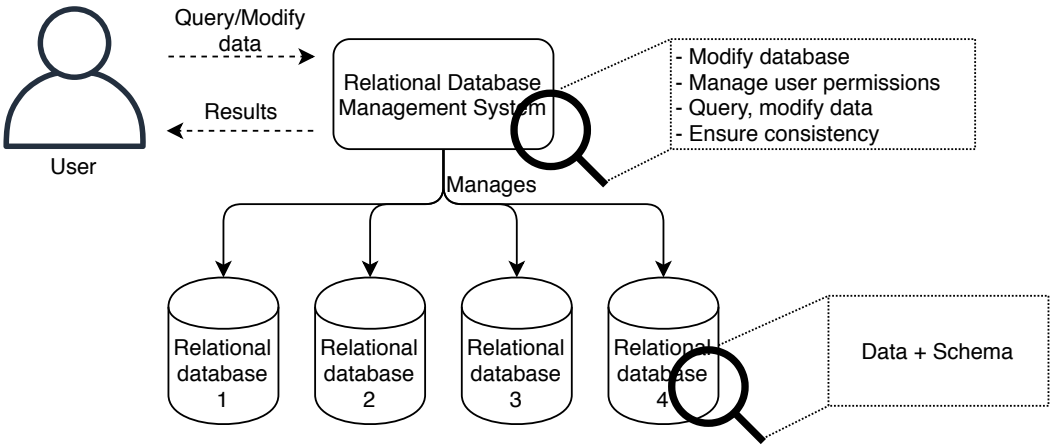

Figure 1.2: Responsibility of database and RDBMS.

Figure 1.2 summarizes the relationship between a RDBMS and its databases. A database contains data with their description (the database schema) and the RDBMS has the responsibility to manage databases by letting user execute queries on them and ensure that data stored all databases stay consistent according to their schemas.

Modern RDBMS: Since the first RDBMS implementations, a lot of new features were added. Modern RDBMS are not only about database management, permissions management, data querying/modification and consistency checking.

For instance, it is possible to define entities performing computation directly in a database. We qualify such entities as "behavioral" in opposition to "structural" entities that do not define computation but describe how data are structured. We identified the following behavioral entities:

- View: A named SELECT query that can be reused in other SELECT queries.
- Stored procedure: A function defined directly in the database via a programming language provided by the RDBMS.
- Trigger: An entity that watches events occuring on a table such as row insertion, update and deletion and that can perform computation either before, instead of or after this event.
- Check constraint: A developer-defined constraint on data implemented as a SQL expression. If the expression evaluates to true, the constraint is satisfied and the row can be inserted/updated. Else, an error is raised. The expression checking for the constraint can involve a stored procedure.

Thesis focus: RDBMS are used in many information systems around the world. This is true for open-source as for proprietary software. While NoSQL DBMS usage spreads nowadays, it is unlikely that relational databases get entirely replaced.

In this thesis, we focus on relational databases. Addressing evolution in the case of relational databases is already complex and raises interesting research topics.

PostgreSQL as reference implementation: While the research conducted in this thesis aims to apply to any RDBMS, we use PostgreSQL[1] implementation as a reference. This choice is motivated by two main reasons.

First, we have access to a real-world database used in our laboratory as well as to its main developer. The features of this database are detailed in Chapter 2.

Second, PostgreSQL is a mature open-source RDBMS providing advanced features related to behavioral entities. Furthermore, according to Stackoverflow yearly surveys related to software technologies usage[2], its usage keep increasing over the years. In 2017, Stackoverflow started to explicitly ask developers about their RDBMS usage (before this question was not explicitly asked). Since 2017, PostgreSQL usage report keeps increasing reaching the second place behind MySQL in 2019.

1.3 Problems

We identified two main problems arising when it comes to evolve a relational database: relational databases are hard to understand and hard to evolve. These two problems are briefly introduced in this section and are more deeply examinated in Chapter 2.

[1] https://www.postgresql.org
[2] https://insights.stackoverflow.com

Relational databases are hard to understand As stated previously, modern RDBMS provide various features to describe data organization and to process data. We identified two reasons why understanding a relational database is complicated.

The first one concerns *the complexity to query RDBMS meta-data*. Indeed, RDBMS are usually meta-described in a set of system tables that can be queries, as any table, using SQL queries. However, retrieving the relationship between the database entities from these tables can be cumbersome as querying graphs data structures stored in tables is not straightforward in SQL.

The second difficulty is that *some meta-data are missing*. For example, we identified that it is not possible to query system tables of a database to gather the callers of a stored procedure.

Relational databases are hard to evolve The effects of a change on software are hard to predict for a human. This is true for relational databases as well but relational databases have two particularities that make their evolution even more challenging.

The first particularity is that *dependencies between a database and a program using it are usually implicit*. This is true for stored procedures depending on tables as well. It is possible to make a stored procedure crash at run time if one change the database schema. The invalidity of the stored procedure source code after the change will not be noticed by the RDBMS.

The second particularity holds for all entities except stored procedures: *the database schema can not be in an inconsistent state at any moment*. An example of an inconsistent state can be a view that references a non-existing table in its source code. This feature makes the relational database evolution different from other software evolution. Indeed, a Java software system can be set in an inconsistent state during its evolution as long as, when the software is recompiled, the source code is in a consistent state. This is not true for RDBMS, the database schema must be in a consistent state at any moment during the evolution.

Summary 1: Problems during relational database evolution

1. Relational databases are hard to understand.
   - 1.1. Complexity to query meta-data.
   - 1.2. Missing meta-data.
2. Relational databases are hard to make evolve.
   - 2.1. Implicit dependencies.
   - 2.2. No inconsistency is allowed at any moment during the execution.

1.4 Software Engineering for Relational Databases

To tackle the above problems, we apply software engineering techniques to relational databases. By "software engineering techniques" we mean that we adapt techniques of re-engineering to work with relational databases. It includes modeling the software, analyzing the model and generating a new version of the database. By writing "relational databases", we mean both structural and behavioral entities.

We propose the approach illustrated in Figure 1.3 as an answer to the problems we identified. This figure splits the process into three parts (each part in dashed rectangles). Each rectangle references a corresponding chapter in the thesis.

The approach takes a relational database as input and first creates a model of its schema. This model is an instance of the meta-model we describe in Chapter 4. Then, there are two possibilities depending on the knowledge of the developer:

1. If the developer has little knowledge of the database and needs to analyze its structure, then one queries the model and eventually search for bad patterns in it. This part of the approach is described in Chapter 5.

2. Else, if the developer has a good knowledge of the database structure and wants to perform a change on it, one makes a change request and gets recommendations on how to perform the change. The recommendation system guides the DBA through the decisions process required to accommodate the schema with the initial change. Once all decisions are taken, a SQL script implementing the evolution and fulfilling the RDBMS consistency constraint is generated. This part of the approach is described in Chapter 6.

Additionally, as suggested by the arrow between the query part and the recommendation part, one might request a change on the model because one detected a bad pattern in the database.

1.5 Contributions

This thesis introduces a toolkit to deal with relational database reverse-engineering and evolution. This toolkit was built over knowledge gained via empirical experiments on relational databases and observations of developers' behavior.

The main contributions of the thesis are:

- The identification of concrete problems encountered during relational database evolution via the observation of a developer (Chapter 2).

- A meta-model designed to help in dependencies analysis related to entities defined inside a database (Chapter 4).

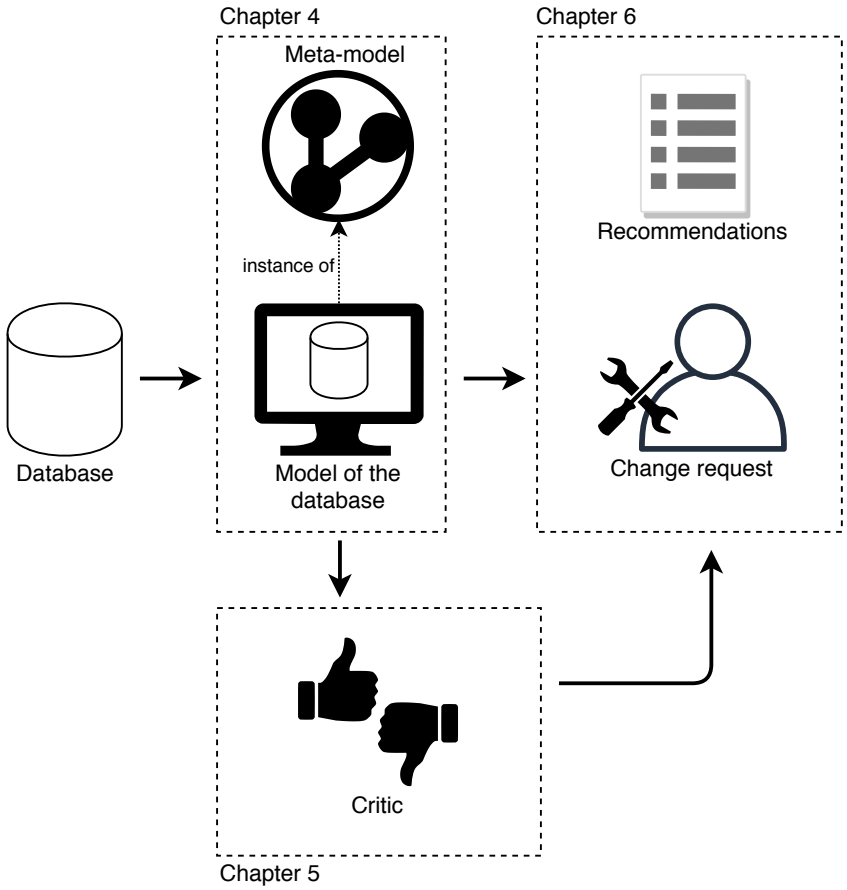

Figure 1.3: Summary of our approach.

- A tool to find quality issues in a database schema and the study of quality issues evolution on 2 case studies (Chapter 5).
- A semi-automatic approach based on recommendations that can be compiled into a SQL script fulfilling RDBMS constraints (Chapter 6).

1.6 Structure of the Thesis

The thesis is organized as follow:

- Chapter 2 analyses problems faced by database architects when evolving a relational database.
- Chapter 3 reports our exploration of the scientific literature around research topics related to this thesis.

- Chapter 4 presents a behavior-aware meta-model for relational databases and the approach we use to create models (*i.e.,* instantiate the meta-model). This meta-model is then validated on two case studies.
- Chapter 5 shows how an instance of the meta-model (a model) can be used to find quality issues in an existing database schema.
- Chapter 6 presents an approach that, from the description of a change to apply on a database schema, provide recommendations to accommodate the database schema with this change.
- Chapter 7 summarizes and concludes the work presented in this thesis and propose future work.

1.7 List of Publications

The list of papers written in the context of the thesis are listed below. The focus of this thesis is to apply software engineering techniques to relational databases. However, as I have the tendency to explore various path I find interesting, I took some opportunities during the thesis to work on slightly different research topics. Still, these topics that do not fall into the theme of this thesis, contributed to my scientific training. Thus, I included these publications in the following list.

Except if stated otherwise, all publications listed below have been published.

Conferences and Journal:

- Vincent Aranega, Julien Delplanque, Matias Martinez, Andrew P. Black, Stéphane Ducasse, Anne Etien, Christopher Fuhrman, Guillermo Polito. Rotten Green Tests: A Replication Study. Empirical Software Engineering, 2020. *(in submission)*
- Julien Delplanque, Anne Etien, Nicolas Anquetil and Stéphane Ducasse. Recommendations for Evolving Relational Databases. International Conference on Advanced Information Systems Engineering, CAISE'20, 2020.
- Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P. Black and Anne Etien. Rotten Green Tests. International Conference on Software Engineering, ICSE'19, 2019.
- Julien Delplanque, Anne Etien, Nicolas Anquetil and Olivier Auverlot. Relational Database Schema Evolution: An Industrial Case Study. International Conference on Software Maintenance and Evolution, ICSME'18, 2018.

- Julien Delplanque, Anne Etien, Olivier Auverlot, Tom Mens, Nicolas Anquetil and Stéphane Ducasse. CodeCritics Applied to Database Schema : Challenges and First Results. International Conference on Software Analysis, Evolution, and Reengineering Early Research Achievement track, SANER'17, 2017.

Workshops:

- Julien Delplanque, Stéphane Ducasse and Oleksandr Zaitsev. Magic Literals in Pharo. International workshop of Smalltalk Technologies, IWST'19, 2019.
- Julien Delplanque, Stéphane Ducasse, Andrew P. Black and Guillermo Polito. Rotten Green Tests, A first Analysis. International workshop of Smalltalk Technologies, IWST'18, 2018.
- Julien Delplanque. Software Engineering Techniques Applied to Relational Databases, International Conference on Automated Software Engineering Doctoral track, ASE'18, 2018.
- Julien Delplanque, Olivier Auverlot, Anne Etien and Nicolas Anquetil. Définition et identification des tables de nomenclatures. INFormatique des ORganisations et Systèmes d'Information et de Décision, Inforsid'18, 2018.
- Julien Delplanque. Software Engineering Issues in RDBMS, a Preliminary Survey BElgian-NEtherlands software eVOLution symposium, BENEVOL'17, 2017.

CHAPTER 2

Motivation

Contents

In this chapter, we present our work to analyze and understand the problems faced by database architects when evolving a relational database. To do so, we analyze a real-world evolution of the AppSI database used in our university. This chapter is an extension of our publication at the International Conference on Software Maintenance and Evolution (ICSME) [Delplanque 2018].

2.1 Introducing AppSI database

AppSI is a PostgreSQL database used for managing members, teams, funding support, etc. in the laboratories of our university. It is a proprietary database mostly developed by a single database architect.

2.1.1 Properties of the schema

This database is used by software systems written in different programming languages. Because of that, the database architect decided to implement, as much as possible, the behavior of client applications directly inside the database as stored procedures. This decision aims to avoid the duplication of behavior implementation across multiple programming languages but also to ensure consistency of all client applications. As an illustration of AppSI database features, Table 2.1 shows the number of entities for each type of entity in the database schema.

Table 2.1: Number of entities for each type of entity in the database schema.

Entity type	Count
Table	95
Column	515
Primary key constraint	93
Foreign key constraint	125
Unique constraint	6
Check constraint	10
Default value constraint	102
View	62
Trigger	20
Stored procedure	64
Trigger function	19
Aggregate function	3

Another particularity of AppSI is that the schema is used in multiple laboratories at the university. Each laboratory has its database instance based on the initial schema. However, developers of other laboratories modify their versions of the schema to adapt it to the specific needs of their users. Laboratories that adopted AppSI benefit from maintenance to accommodate new features and bug fixes. Each change made to AppSI needs to be ported to "forked" databases while handling the fact that all databases continue to evolve separately, thus drifting further apart. To do that, modifications are stored as SQL scripts and applied to the other instances of AppSI.

2.1.2 An evolution of AppSI

The University of Lille uses the Lightweight Directory Access Protocol (LDAP), a standard application protocol for accessing and maintaining distributed directory information services. The administration decided to evolve the LDAP schema to enable users to have multiple identifiers. The idea is that each person has a personal account and zero or many accounts that represent a function. These "function accounts" can be assigned to let a person use a specific software for example.

As AppSI stores the unique LDAP identifier of people working at the laboratory, the database is directly impacted by this change. AppSI database also has its own way to identify a person in the database uniquely implemented as a column constrained by a primary key constraint.

Before digging into technical details of AppSI evolution, let us set the concepts one need to know to understand the evolution:

- *person*: When we use the term "person", we mean a human for who the data has been encoded in the information system.
- *person table*: The table in the AppSI relational database that contains data of people working at the laboratory.
- *id column*: A column of the *person table* in AppSI database that is constrained by a primary key constraint. The values held by this column are used to uniquely identify data of a *person* stored in the *person table*.
- *uid column*: A column of the *person table* in AppSI database that stores the LDAP identifier of each *person* working at the laboratory. The *uid column* is used as the login of the users for web applications using AppSI database. Thus, the pair of values (*id*, *uid*) are unique in AppSI database.
- *uid attribute*: An attribute of the LDAP schema allowing one to refer to a person's data uniquely before the evolution.

Additionally to the LDAP schema evolution to enable users to have multiple identifiers, the *uid attribute* was renamed as `login`. This evolution necessitates to similarly rename the *uid column* of the *person table* as `login`. This column contains the primary identifier of a person. Such an induced evolution aims to ease the understanding and the maintenance of the database and its client applications.

To support multiple LDAP identifiers in AppSI, a new table named `account_alias` is created. It gathers all the secondary identifiers of a person in the `login_alias` column. The `id_person` column is a foreign key to the *id column*, primary key of the *person table*.

Before this evolution, it was possible to find the id of a person from their LDAP identifier. After the evolution, since a person may have several LDAP identifiers, it is necessary to use a stored procedure to find the id of a person from one of their LDAP identifier (the main or a secondary one). For this purpose, a view was created to ease the correspondence between one of the LDAP identifiers of a person and his/her primary key. Figure 2.1 provides a sketch of this evolution.

Of course, once this modification is integrated, entities of the database using the *uid column* of the *person table* have to be adapted in order either to use the new *login column*, or to use the new `account_alias` table with its `login_alias` column and a join with the `person` table.

Although this evolution is rather simple to understand, it is not trivial to implement. To plan this evolution, the database architect established a roadmap of what he needed to do. During the whole evolution, he uses this roadmap to keep track of what was done and what remained to do. The roadmap was also updated when the database architect discovered he had forgotten something or when some step turned out to be more complex than originally planned.

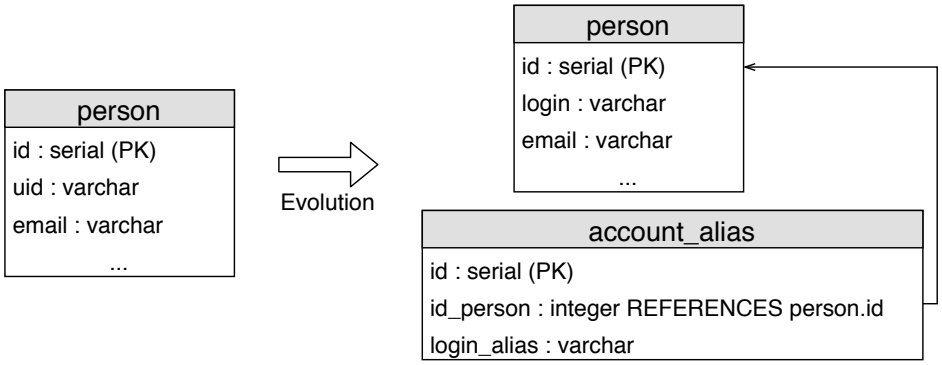

Figure 2.1: Summary of the evolution to be achieved by the database architect.

To give a bird's eye view of the whole evolution, we provide the roadmap here:

1. Copy database (to set up a realistic, up to date, development environment);
2. Create a dump of the schema (*i.e.* the structure of the database, tables, views, stored procedures, etc. are serialized as a SQL script from which the database can be rebuilt from scratch);
3. Rename `uid` column as `login`;
4. Search for occurrences of `uid` in stored procedures signatures and source code, rename foreign key constraints and indexes. This step is not only about replacing references to `uid` columns in the source code embedded in the database, it also consists of modifying entities for which the name contains `uid`. For example, the local variable `uidperson` of `login` stored procedure is renamed `loginperson`;
5. Add `account_alias` table;
6. Create a view managing main and secondary identifiers;
7. Create a stored procedure returning the main login according to the account given as parameter;
8. Modify `key_for_login(login)` stored procedure to return the primary key of a person whatever the identifier used;
9. Add entry in `configuration` table (this is a table containing configuration parameters for applications using the database: *e.g.*, IP addresses, ports, URLs, etc.);
10. Add `mail(account)` stored procedure to determine the email address based on the main login;

11. Compute email address in `logins` view;
12. Apply required changes on SQL patch (addition of queries written in previous steps into the patch);
13. Execute patch on a copy of the database;
14. Check generating reports on data in the database and, if necessary, replace `uid` column references by references to `login` columns in queries;
15. For each client application, look for `uid` references and replace them by `login` references.

The last two tasks concern the update of applications using AppSI database. They do not fall within the scope of our analysis.

2.2 Context

We were asked by the database architect of our university if we could propose solutions to help him evolve a PostgreSQL database named AppSI. The "2017-10-10" version of AppSI has some characteristics that make it difficult to evolve:

- It has many views (62). Views make the evolution of the database complicated because no inconsistency is allowed at any moment during the evolution (see Section 2.6.2). Because of that, views transitively using other views have to be removed and recreated in cascade.
- It has many stored procedures (64). Stored procedures make the evolution of the database complicated because of missing meta-data (see Section 2.6.1). The validity of the stored procedure must be verified by actually calling it and see if it produces a runtime error due to a broken reference.
- The same database schema is used in other laboratories and any update on one instance must be made in the form of a SQL script (a "patch") that can be applied to the other instances (as we discussed previously).

The database architect complained that evolutions are difficult because of these various cross-dependencies within the database. He found no tool that could help him in this task. To understand the problems he faced, we analyzed his actions on a real example of a schema evolution that he had to perform. We identify concrete problems he encountered and suggest tools that should be created to help him better handle these problems.

2.3 Conditions of the Case Study

For practical reasons, we could not be present when the database architect performed the evolution and it was not possible to postpone it. The architect agreed to record his screen during the whole task. The result consists of three recordings of about 1 hour each (total video time is 3.5 hours). Unfortunately, for confidentiality reasons (personal data of university employees appearing at different moments of the videos), the videos can not be made publicly available.

We analyze recordings of the architect screen to understand problems he faces during an evolution. Analyzing video data is not easy because of their unstructured nature. Furthermore, we can observe what the architect does but we need to infer why he does it. This information is not encoded in the video.

In our case, we want to understand:

- Which tools the architect use to perform the evolution?
- How tools are used during the evolution?
- How the architect achieves the evolution?

The first step to be able to understand data in the video is to extract some structure from it. We choose to split the video into entries to have a more structured version of data encoded in the recordings. As delimiter for these entries, we chose to use timestamps for which we observe changes in the screen display. For example when the database architect switched from one tool to another (*e.g.* text editor, shell terminal, or a database development tool), or from one tab of a tool to another tab (*e.g.* multiple files in a text editor). This delimiter is quite objective as we do not need to inject knowledge external to the video to split it. Because of that, our experiment is reproducible on another recording of another evolution for a different database conducted by a different architect.

We segmented the video into a list of "entries" containing a timestamp and a description of what happens on the screen between the timestamp of the entry and the next timestamp. The full transcript is available at https://github.com/juliendelplanque/icsme2018data. From that list of entries, we perform a qualitative and a quantitative analysis to understand the evolution. Figure 2.2 provides a visual representation of our methodology to analyze the video.

2.4 Qualitative Analysis

Starting from the list of entries we transcribed, we perform a qualitative analysis of the evolution. This analysis is twofold. First, we decompose the evolution into "actions" which are groups of entries that are related. Then, we group these

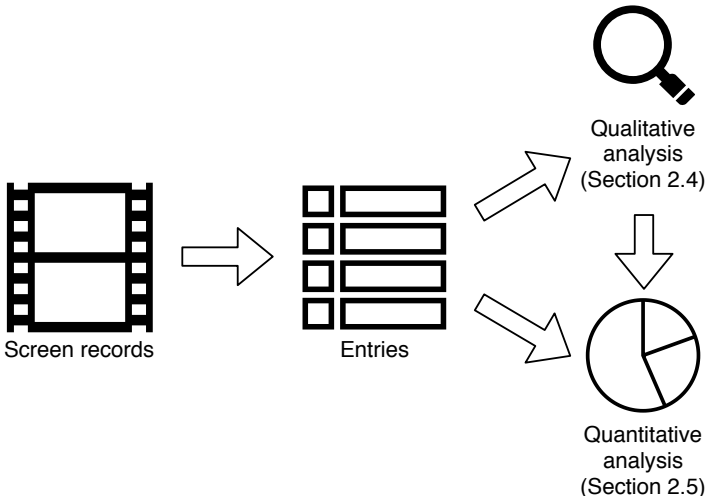

Figure 2.2: Methodology to analyze the video.

actions into activities which abstract the semantic of the architect's actions at a coarser grain. Activities allow us to identify the process followed by the architect to implement the evolution and to analyze it.

2.4.1 Decomposing the Evolution

We abstracted all entries in a list of 18 actions listed in Table 2.2. The evolution is first performed on a development version of the database and then made effective on the production version. It is thus important to synchronize the two versions, that is to say, make sure that the schema of the development database is even with the production database (action 1).

Actions 2, 3, and 14 correspond to moments where the document manipulated (*e.g.* the evolution roadmap, the database schema dump, the resulting SQL patch) is observed. That is to say, the corresponding windows are put in the forefront on the screen, but no interaction is visible in the video.

Searches can be performed in the database schema dump (via text-search utility of the text editor) either to identify an impacted entity (action 4) or to copy a fragment of SQL code like a `CREATE TABLE` query (action 5).

Action 13 occurs in case a less common query has to be created (for example an `ALTER TABLE` to change a constraint) and no example of the required syntax can be found in the database schema dump.

The schema is modified in the Navicat database management tool either using a dedicated UI (action 8), or by editing source code through the query builder tool (action 9). The query builder is a code editor with a button allowing one to let the RDBMS executes the SQL code it holds. Using the query builder tool allows the

database architect to just copy/paste the query in the database schema dump when it is considered valid.

To check the validity of queries modifying the schema, the architect checks that the modifications have been performed (action 11), executes `SELECT` queries to check the form of data stored in tables or returned by views and stored procedures (action 6), executes `INSERT` / `UPDATE` / `DELETE` queries to check constraints applied on the data (action 7), or runs unit tests written in an external language (action 16).

Action 15 occurs when a change is considered invalid, whatever the reason, and the architect modifies the related source code.

The patch is regularly modified (action 10) to integrate Data Description Language (DDL) queries (*i.e.* `CREATE`, `ALTER` and `DROP` commands) and the evolution roadmap is updated (action 12).

Finally, action 17 corresponds to moments where the database architect is not directly working on the evolution. These moments are different from actions 2, 3, or 14. To distinguish this "inactivity" from actions 2, 3 or 14, we used the two following criteria:

1. the screen shows absolutely no activity (*e.g.* no mouse movement, no scrolling, etc...); and

2. the duration of this absence of activity is greater than or equal to 20 seconds (we classified 1 entry that did not correspond to this criterion and last for 9 seconds but it looked clear to us that it was a short inactivity period because there was no activity on the screen and no tool was shown on the screen).

2.4.2 An Informal Process

The decomposition of the evolution (performed in sub-section 2.4.1) leads us to the list of entries with, for each entry, the number of the corresponding action. While analyzing the evolution decomposition, we observed repetitions and regularities in the actions taken. We formalized a small process from these regularities. This process is illustrated in Figure 2.3.

The process is applied for each schema evolution. The activities of the process are:

1. Read the next step to perform in the migration roadmap.

2. The queries required by the roadmap step are coded in the database development tool and/or the SQL patch. If queries are written in the development tool, the architect tends to work iteratively. That is to say, he starts with a small, simple query and modifies it iteratively to reach the desired result.

Table 2.2: List of actions during AppSI evolution

action #	Description
0	Other.
1	Synchronise development database with production database.
2	Observe patch.
3	Observe database entities.
4	Analyzes the schema dump by performing textual search.
5	Search for a fragment of SQL code in schema dump and eventually copy it.
6	Execute SELECT query from IDE.
7	Execute INSERT/UPDATE/DELETE query from IDE.
8	Execute DDL query from IDE's UI.
9	Execute DDL query in IDE by writing SQL code.
10	Modify patch.
11	Verify a change is correctly applicated.
12	Update evolution roadmap.
13	Check PostgreSQL documentation.
14	Check evolution roadmap.
15	Modify source code in query builder.
16	Run unit tests written in an external language.
17	Inactivity.

3. These queries are executed (on the development database) mostly wrapped in a transaction that will be rolled back (*i.e.* between `BEGIN` and `ROLLBACK` commands). Using a transaction enables the architect to test or validate an SQL syntax, check the result is correct and eventually undo the whole part of the patch under test if one of the queries fails during execution.

4. In case of errors, the queries are modified. In this case, the architects returns to activity (iii). This is the first interesting loop in the process (sub-loop A). It is executed until no more error can be detected in the queries execution. At this level, issues come from either syntax errors or, more often, nonexistent referred entities. Entities may be nonexistent if they have been renamed, removed, or not yet created.

5. Once there are no more errors in the queries, they are made effective through execution in a transaction that is committed (*i.e.* between `BEGIN` and `COMMIT` commands).

6. But even if the queries did not produce errors, their result might not be the expected one. To check whether this is the case, some observations or tests

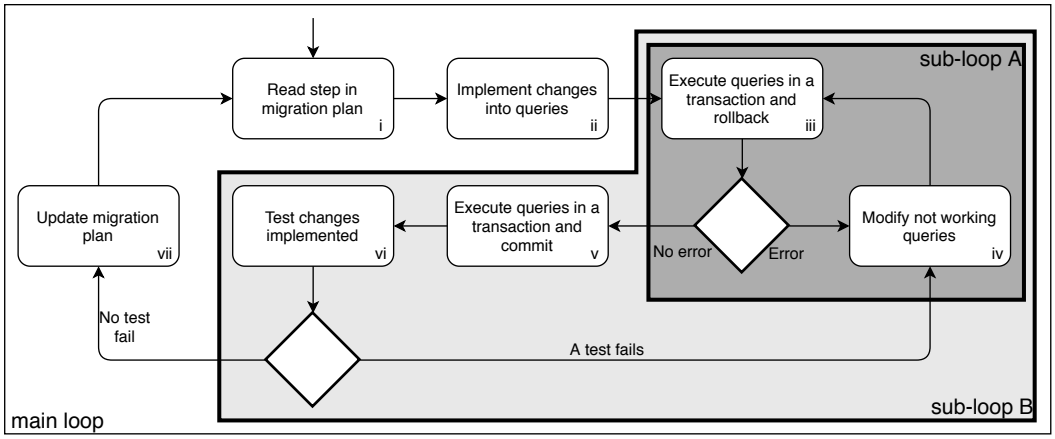

Figure 2.3: Process formalized from the database architect actions.

are manually performed on specific data familiar to the architect. These tests can be performed as `SELECT` queries, modifications on the data stored in the database and/or simply looking at the database structure from the development tool UI.

If the tests fail, the queries have to be corrected by returning to activity iv. This is the second interesting loop in the process (sub-loop B). At this level, issues come from deeper causes than syntax errors (semantic causes). In "traditional" software development (*e.g.*,, Java, Python, etc), they would be caught by tests.

7. Finally, if the tests succeed, the change is considered valid and the architect goes to the next step in the evolution roadmap, possibly updating it.

Each step can involve one or several actions. Table 2.3 summarizes the mapping between the process activities and the observed actions in the video.

Table 2.3: Relations between actions and steps in the observed process.

Activity	Actions
(i)	14
(ii)	2, 3, 4, 5, 10 or 13
(iii)	9
(iv)	2, 3, 4, 5, 8, 10, 13 or 15
(v)	9
(vi)	3, 6, 7, 11 or 16
(vii)	12

We note that in "traditional" software development, errors causing sub-loop A are usually detected by the IDE while the code is being written. Modern IDEs

highlight syntax errors and simple errors, like referencing a nonexistent entity in the code. This is not the case with the tools used by the database architect.

Similarly, in "traditional" software development, sub-loop B would be helped by testing platforms. Here, the tests were manually performed by the architect. They may not even be saved for later use because they may involve employees' data. The tests thus rely on the architect's knowledge of the database schema and its data. In addition, we noticed that sometimes tests are reduced to schema or data observations.

Although the architect confirmed that the process we observed and formalized is the objective he wants to reach, it is important to notice that it is not always possible for him to follow the process strictly. For example, when the architect is interrupted during the process he might not be able to restart from the right activity. Thus, we observe that some activities are missing in the video.

Findings 1: Qualitative Analysis

Qual.F. 1. The IDE is not sufficient to understand the schema as we observed the architect performing text searches on the dump to understand dependencies between entities of the schema.
Qual.F. 2. The schema is modified from the UI of the IDE and via custom DDL queries.
Qual.F. 3. The architect uses multiple strategies to test the database and some of them are not automated.
Qual.F. 4. The architect uses an implicit process to conduct the evolution.
Qual.F. 5. Queries are built incrementally, starting from a simple and working query and then complexifying it to satisfy requirements.
Qual.F. 6. Changes are simulated on the database before their real commitment (using a transaction that is rollbacked).
Qual.F. 7. Syntax errors and reference to non-existing entities are not caught by the IDE before executing a query.

These findings are specific to the experiment but some of them are used to infer generic problems in Section 2.6.

2.5 Quantitative Analysis

To get a better understanding of architect activities, we perform a quantitative analysis on the entries we created in Section 2.3 and the result of the qualitative analysis (Section 2.4). First of all, based on the actions identified in Table 2.2, we analyze the time spent on each of them. Then, we analyze the process followed by the database architect during the development and more specifically the time spent in

each sub-process loop (A and B in Figure 2.3). Finally, we look at the time spent using each tool during the development session recorded.

2.5.1 Time Spent per Action

Actions 0 (others) and *17* (inactivity), with cumulated durations of 40 and 30 minutes respectively, are removed from the dataset for the analysis because their semantic is not interesting for our purpose. The remaining ∼137 minutes of relevant actions are analyzed. The minimum duration measured for relevant actions is 9 seconds, the maximum is 6 minutes 14 seconds.

Action 10 (modify patch) takes the most of the time: 45 minutes of cumulated duration (around 18.5% of the relevant time). In terms of occurrences, it is also the most frequent action (63 occurrences). This action corresponds both to the design and the writing of queries modifying the schema. It is difficult to evaluate the reflection time.

The architect mostly writes the queries to modify the schema directly in the patch before executing them in the query builder of the database management tool. Three reasons explain this choice. First, the architect needs to keep track of the changes in a patch to apply them on the database of another laboratory later. Second, he had up to 16 queries to apply together since an entity referenced by another one can not be dropped. It is easier to manage them all together or to take care of their order in the text editor. Third, the architect considers the patch as the reference document in comparison to the query builder. Nevertheless, this action also includes when the architect modifies the patch after correcting an erroneous query in the query builder.

Action 9 (execute DDL query in IDE by writing SQL code) cumulated durations of 11.5 minutes and occurs 45 times. This action completes action 10 since it executes queries copied from the patch to the query builder. The corrected version is then copied back to the patch. The fact that this action has a lot of occurrences but lasts only 11.5 minutes is explained by the fact that executing SQL code is nearly immediate. Modifications of the queries are caught by other actions (10 or 15).

Action 6 (execute SELECT query from IDE) cumulated duration lasts for ∼13 minutes and occurs 28 times. The frequency of the action is due to the fact that it corresponds to an execution of a `SELECT` query. Such queries are used to check that the schema evolution has correctly been done, *i.e.* provides the expected result.

Action 3 (observe database entities) cumulated duration lasts for 10.5 minutes and occurs 21 times. The frequency of this action comes from the fact that even if the architect knows pretty well the database, he needs to observe some entities while writing queries.

If we take a look at the number of occurrences, *action 2* (observe patch) and *action 4* (analyzes the schema dump by performing textual search) occurred 22 times each while their cumulated durations last for 7.3 minutes and 8.3 minutes respectively. *Action 2* is comparable to action 3: it corresponds to an observation of the dump before or while writing queries to understand the structure of the database. *Action 4* is noticeable as it corresponds to the research of entities in the dump through a simple textual text search to identify dependencies between entities.

The other actions (1, 5, 7, 8, 11, 12, 13, 14, 15 and 16) are less relevant in terms of time and occurrences. For instance, they occur between 2 times (for actions 1, 7 and 16) and 13 times (for action 15). Because of little occurrences and cumulated time, we can not say much about these actions as we have little data about them.

2.5.2 Time Spent in Sub-Loops of the database architect's process

As introduced in Subsection 2.4.2, we identified 3 loops in the process followed by the architect:

- *main loop* concerns the complete implementation of a feature. It might include iterations on *sub-loop A* and/or *sub-loop B*.
- *sub-loop A* concerns the resolution of syntax errors and errors raised because of references to non-existing entities. The latest is interesting because it can be caused by a wrong order of the queries to execute (*e.g.* a table creation executed after the creation of a view referencing the table in its query).
- *sub-loop B* concerns the resolution of semantic errors. That is to say, the source code is correct and did not raise any error at execution but what it implements does not answer the requirements of the database architect.

We tagged entries of our dataset according to the loop they belong to when it was possible. Some entries that do not belong to one of these loops are not tagged (*e.g.,*, at the beginning of the first video, the architect does some actions that do not follow the process). The main loop occurred 9 times, sub-loop A occurred 9 times and sub-loop B occurred 6 times.

We observe that:

- Some main loops do not have iterations on sub-loops;
- Sub-loop B happens less (6 times) than sub-loop A (9 times);
- Sub-loop A appears up to 5 times in the same main loop. This observation can be explained by the fact that in this main loop, up to 16 queries are run at the same time. Thus, it seems normal to have a lot of syntax errors;

- Sometimes, sub-loop B can be short because what is done to "test" the changes is simply observing entities of the database from the IDE;
- One sub-loop B has an interesting property: a syntax error arise during an activity of this loop. It happened because, in order to fix a semantic problem (loop B), a modification on the structure of the schema had to be done. However, the queries written to implement this modification were incorrect syntactically.
- Another syntax error arise during one activity of a sub-loop of type B but for a different reason. A stored procedure was previously created with a syntax error in its source code. The RDBMS accepted to create this stored procedure without any warning. Nonetheless, when this stored procedure is tested by the database architect to ensure it works as expected, a syntax error is raised. The particularity here is that this error comes from the procedural language used in stored procedure body. This observation shows an interesting particularity of PostgreSQL stored functions: a stored procedure may hold syntactically invalid source code in a database. Such invalid stored procedures will only be discovered once they get executed.

2.5.3 Time Spent per Tool

For each entry in the data extracted from the video, we identified the tool used by the database architect. Four tools and their usage are identified below:

- Text editor: Used to browse the dump of the database, browse and modify the patch.
- Navicat: The database browser providing a set of tools to modify the database. Similar to an Integrated Development Environment (IDE) but for databases.
- Web browser: Mainly used to search PostgreSQL documentation.
- Terminal: Allowing one to interact with the operating system, manipulate files, interact with PostgreSQL, etc...

The text editor is the first tool used in terms of time (~ 79.3 minutes), the second one is Navicat (~ 52.5 minutes). Although it is a bit surprising to find the text editor used more often than the IDE, it is explained by the fact that the database architect has to store the changes in a patch file and also, as previously said, because up to 16 queries are run at the same time to perform a task of the evolution. The text editor is thus a good tool to develop the patch for the architect. However, an important drawback is that the content of the patch has to be copy/pasted into the

query builder to be tested. The process induced by the usage of the text editor to develop the patch (1. write SQL queries in the patch in the text editor; 2. copy/paste the queries; and 3. execute these queries in the query builder) is not always strictly followed during the video. When the part of the patch pasted in the query builder raises an error it happens that the architect modifies the problematic query directly from the query builder. Normally, he should come back to the text editor, modify the problematic query from there and then copy the fixed query in the query builder to test it. Because this process is sometimes not followed, the patch opened in the text editor has to be re-synchronized with the content of the query builder.

Another reason for patch development to happen in a text editor is because the IDE does not generate a patch after a full schema evolution to apply the changes on the production database or other instances.

The terminal is used during a significant amount of time (~ 28.76 minutes) to send administration command to PostgreSQL (*e.g.* create a dump of the database), to use grep[1] to perform a textual search in the dump or in the patch and to edit the configuration files on the operating system.

Findings 2: Quantitative Analysis

Quan.F. 1. Queries are mostly written in the text editor and then copied in the IDE to tests their execution.

Quan.F. 2. In the development process we identified, the sub-loop concerning fixes of syntax errors and broken references appears up to 5 times include a main loop.

Quan.F. 3. Because of poor testing, the sub-loop B concerning the test of changes made to the database has a short duration.

Quan.F. 4. It is possible for a stored procedure to hold syntactically invalid source code.

Quan.F. 5. The text editor is used more often than the IDE.

Quan.F. 6. The IDE does not support the generation of a patch to migrate a database.

These findings are specific to the experiment but some of them are used to infer generic problems in Section 2.6.

2.6 Problems

From our observations, we identified two main problems arising when it comes to evolve a relational database: relational databases are hard to understand and relational databases are hard to evolve.

[1] https://www.gnu.org/software/grep/

2.6.1 Relational databases are hard to understand

Modern RDBMS provide various features to describe data organization (tables, constraints, etc.) and to process data (stored procedures, views, etc.). We identified two reasons why understanding a relational database is complicated.

Complexity to query meta-data: The RDBMS provides metadata related to its structure, a meta-description of itself. It takes the form of a set of "system" tables and views. There is even a standard that emerged called the "information schema" providing information about entities in the database. However, all RDBMS implementation do not implement this standard.

Additionally to this standard, each RDBMS usually stores its meta-data in custom tables and views that are used internally. Eventually, a developer can query these tables to retrieve information about the structure of a database. However, one need a good understanding of the system tables and their relationships. This knowledge is not easy to gain as it requires to understand meta-data tables schema which boils down to the same problem.

Our observations suggest that the IDE used by the architect does not expose capabilities to query meta-data to the architect as he performs text search in the dump to understand dependencies between entities of the schema (Qual.F. 1.).

Missing meta-data: Furthermore, in the case of PostgreSQL, we realized that some dependencies between entities are missing in the meta-data tables (Qual.F. 7.). For instance, references to entities made in the source code of stored procedures are simply not available in the meta-data tables. The source code of stored procedures is not stored as an abstract syntax tree but as plain text. Because of that, the removal of an entity referenced in a stored procedure is possible without any complaint from the RDBMS. The bad effect of this behavior is that if the stored procedure gets executed, it will crash.

2.6.2 Relational databases are hard to evolve

The effects of a change on software are hard to predict for a human. This is true for relational databases as well but relational databases have two particularities that make their evolution even more challenging.

Implicit dependencies: Developers do not know dependencies of software using the part of the database under modification. One can, for example, remove a column used by a client and make the client crash. This problem has been addressed by Loup et al. [Meurice 2016a] in the context of software clients written in Java.

In the case of PostgreSQL, a similar problem arises with stored procedures. One can break stored procedures source code via a schema change. PostgreSQL will not notice this dependency break and when the stored procedure is invoked, it will crash (Qual.F. 7., Quan.F. 4.).

Client software usually build SQL queries as strings that are sent to the RDBMS to be executed. Because of that, SQL queries ar often built dynamically via string concatenation. In contrast, PostgreSQL's language allows SQL queries to be written directly in stored procedures source code (so they are usually not embedded in a string while it is possible to do it). Thus, stored procedures source code is rarely building SQL queries dynamically which makes source code analysis less complicated.

No inconsistency is allowed at any moment: The RDBMS does not allow one to put the database in an inconsistent state at any time during the evolution. For example, a state of the database where one of the entity references a non-existing entity is inconsistent. This particularity is the reason of the process used by the architect (Qual.F. 4.). Indeed, the change simulation before the real commitment (Qual.F. 6.) is used to discover potential inconsistencies arising from a part of the evolution script.

RDBMS does not allow inconsistencies because their execution is not stopped during the evolution. Changes to the schema of a database are made via SQL queries and these queries have the same constraints imposed by the RDBMS as other queries.

This is different from the development of other software written in Java, Python or C programming languages for example. With software written in those languages, while techniques for dynamic software update [Hicks 2005] exists, a developer usually:

1. Stops the execution of the software.
2. Apply a modification on the source code.
3. Compile the source code (optional, depends on the language).
4. Re-run the software.

In this process, as step 2 is done while the software is not running, the developer can temporarily put the source code in a non-executable or non-compilable state. Being able to temporarily reference non-existing entity is a valuable feature when it comes to modify software. Yet, a RDBMS forbids developers to do that.

2.7 Conclusion

In this chapter, we analyzed the screen record of an evolution performed on AppSI: a database used at our university. We identified 2 main problems (each with 2 sub-problems) occurring when it comes to maintain or evolve a relational database.

1. Relational databases are hard to understand.
   1.1. Complexity to query meta-data.
   1.2. Missing meta-data.
2. Relational databases are hard to evolve.
   2.1. Implicit dependencies.
   2.2. No inconsistency is allowed at any moment.

The validity of these problems are strengthened by the results of our survey related to issues arising during PostgreSQL database development [Delplanque 2017a]. Indeed, in the results of this survey, "maintenance complicated" issue was reported for stored procedures. By discussing with multiple database architects, we realized that it is not an isolated problem. It also arises with views and more generally, handling dependencies between entities in a relational database is a complex task.

The identification of these problems is a starting point to build the state of the art (Chapter 3) and to position our approach against previous works related to: *relational database reverse engineering* to address the *relational databases are hard to understand* problem and *relational database impact analysis* to address the *relational databases are hard to evolve* problem.

CHAPTER 3

State of the Art

Contents

In Chapter 2, we identified problems occurring when a relational database evolves: (1) *relational databases are hard to understand* and (2) *relational databases are hard to evolve*.

In this chapter, we review the scientific literature to assess the state of the art related to these problems. We steer towards *relational database reverse-engineering* to address the first problem and *relational database impact analysis* to address the second problem.

Before investigating the literature related to relational database reverse engineering and relational database impact analysis, let us give a short reminder about database design.

3.1 Database Design

Database design is a process consisting in the conception and implementation of a database that meets users requirements as described by Batini *et al.* [Batini 1992]. Figure 3.1 illustrates the 3 steps of this process.

1. *Conceptual design (Users requirements → Conceptual schema)*: From the user requirements, a high-level description of the structure of the database is produced: the *conceptual schema*. This description is independent of any particular data model. The conceptual schema can be expressed in various formalism: Entity-Relationship (ER) diagram, Unified Modeling Language (UML) diagram, etc.

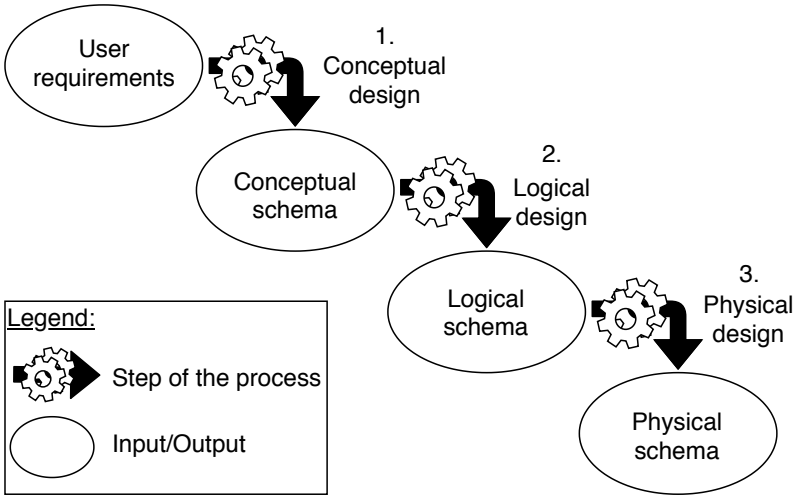

Figure 3.1: Database design process inspired from Figure 1.2 in [Batini 1992].

2. *Logical design (Conceptual schema → Logical schema)*: The conceptual schema is specialized for a particular logical model (*e.g.,* the relational model). However, it is not tied to a specific database management system.

3. *Physical design (Logical schema → Physical schema)*: The logical schema is then converted to a representation that can be directly translated into a sequence of DDL statements for a specific database management system: the *physical schema*. The DDL statements corresponding to the physical schema can be executed by a database management system to make the database operational.

The process of database design is often designated by the term "forward engineering" in opposition to "reverse-engineering". Indeed, as we will discuss in the next section, reverse-engineering a database is basically the opposite process: starting from the physical schema to retrieve the logical and conceptual schemas.

3.2 Relational Database Reverse-Engineering

When a relational database needs to evolve, one needs to have a good understanding of the relationships between entities of the database as well as their purpose (as we discussed in Chapter 2). This information is normally provided by the logical and the conceptual schemas. However, often, these schemas are not available and only DDL code or an access to RDBMS meta-data is available. To retrieve fragments of this information, a reverse-engineering process can be applied to relational databases.

3.2.1 Database Reverse-Engineering Process

Hainaut *et al.* [Hainaut 2009] describe a generic process for database reverse-engineering (applicable on non-relational databases as well). Figure 3.2 illustrates the reverse-engineering process.

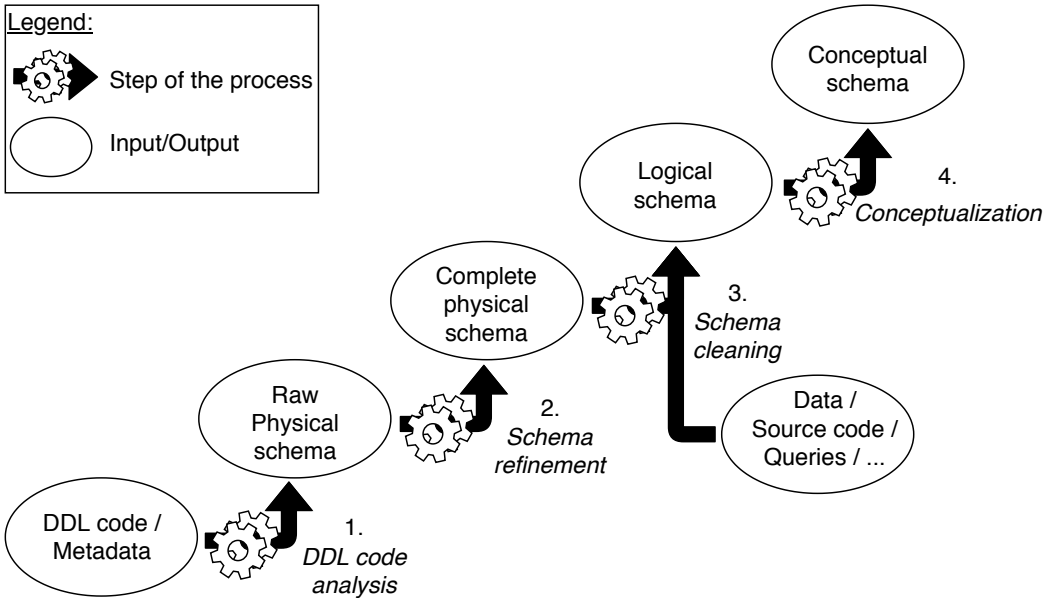

Figure 3.2: Database reverse engineering process adapted from Figure 4-3 in [Hainaut 2009].

This process is made of the following steps[1]:

1. *DDL code analysis (DDL code/Metadata→ Raw physical schema)*: This first step consists in analyzing the DDL statements or the meta-data provided by the RDBMS to provide a *raw physical schema* of the database. The DDL code analysis allows one to abstract the implementation details of the DDL code or the meta-data schema. For example, one can implement a primary key constraint for a table directly from inside the `CREATE TABLE` statement or one can execute a `CREATE TABLE` statement in which no primary key is specified and then execute an `ALTER TABLE` statement that specifies the primary key constraint. DDL code analysis step aims to provide an abstraction over this kind of implementation details.

[1] In this thesis, we scope the generic database reverse-engineering process describe by Hainaut *et al.* to relational databases. For example, the "Physical integration" step has been discarded from the explanations as it occurs when multiple DDL sources are available for the analysis which usually not the case for relational databases.

2. *Schema refinement (Raw physical schema → Complete physical schema)*: The physical schema of the database is enriched with additional sources of information to retrieve implicit or lost constructs. For example, in this step, if foreign key constraints are not explicitly specified in the schema, they could be retrieved by analyzing data present in the database. Various sources of information can be used depending on their availability: data stored in the database, source code of client application, SQL queries, etc. The result of this step is (in theory) the *complete physical schema* of the database. Of course, the completeness of the physical schema depends on the various sources of information available and on the accuracy of analyses applied on them.

3. *Schema cleaning (Complete physical schema + Artefacts → Logical schema)*: The complete physical schema gets specific technical constructs discarded (*e.g.,* indexes) to produce the *logical schema*. As stated by Hainaut *et al.* [Hainaut 2009], *"[...] this schema is the document the programmer must consult to fully understand all the properties of the physical data structures (s)he intends to work on."* This step might rely on additional artifacts such as data, queries, source code of external programs, etc.

4. *Conceptualization (Logical schema → Conceptual schema)*: The logical schema is analyzed and entities that are artifacts of the specialization for a particular data model or of technical optimization are removed.

Furthermore, Meurice [Meurice 2017] identified 5 techniques to perform this relational database reverse-engineering process according to the information source(s) exploited during the schema refinement step. As we find this taxonomy useful to have a bird's eye view on relational database reverse-engineering research, we reuse it as-is:

a) Database schema analysis: *"Analyzing the database schema structures may help identify hidden constructs such as relationships and hierarchies between data."*

b) Data analysis: *"Mining the database contents can be used in two ways. Firstly, to discover implicit properties, such as functional dependencies and foreign keys. Secondly, to check hypothetic constructs that have been suggested by other means."*

c) Graphical User Interface (GUI) analysis: *"Forms, reports and dialog boxes are user-oriented views on the database that exhibit spatial structures, meaningful names, explicit usage guidelines and, at runtime, data population and error messages that can provide information on data structures and constraints."*

d) Static program analysis: *"Analysis, such as data-flow graph exploration, can bring valuable information on field structure and meaningful names. More sophisticated techniques such as program slicing can be used to identify complex constraint checking."*

e) Dynamic program analysis: *"In the case of highly dynamic program-database interactions, the database queries may only exist at runtime. Hence recent techniques allow to capture and analyze SQL execution traces in order to retrieve structural information."*

In the context of this thesis, we rely on *database schema analysis* technique but we model (and thus extract information about) behavioral entities of the database as well (*e.g.,* stored procedures). For that purpose, we also apply *static program analysis* on behavioral entities to extract information related to the relationships these entities maintain with other entities of the database. We could have used *dynamic program analysis* technique for the same purpose but it was not necessary as we observed that queries performed by behavioral entities of the database are usually not built dynamically.

Because of that, we focus our state of the art on database reverse engineering to the three following techniques: *database schema analysis*, *static program analysis* and *dynamic program analysis*.

3.2.2 Criteria to Compare Relational Database Reverse-engineering Approaches

We identified 3 relevant criteria to compare relational database reverse-engineering approaches:

Crit. 1. For which kind of entities does the approach gather information to build a model of the database?

Possible values: *Structural*/*Behavioral*

Crit. 2. What formalism is used to model the database and programs using it?

Possible values: The name of the formalism used (*e.g.,* Extended Entity-Relationship, Unified Modeling Language, custom meta-model, etc).

Crit. 3. Is there a human intervention during the model importation or is the approach automatic?

Possible values: ✓/✗

3.2.3 Approaches to Reverse-Engineer Relational Databases

Database Schema Analysis Markowitz and Makowsky [Markowitz 1990] developed a procedure to translate relational schemas as extended entity-relationship diagram. The procedure support foreign keys and can be extended to support null constraints.

Castellanos [Castellanos 1993] created a methodology for importing relational databases schemas into a model with a richer semantic level: the BLOOM model. Their methodology involves extractions of implicit semantics (*e.g.,* if a RDBMS does not support the explicit declaration of foreign keys, a heuristic is used to retrieve them). The methodology aims to minimize as much as possible human intervention. The author used this methodology in the context of federated database systems [Sheth 1990] to gather higher knowledge on individual databases.

Shoval and Shreiber [Shoval 1993] proposed a process to transform a relational database schema to a Nested Binary Relationship (NBR) [Shoval 1985] model. They implemented this process in a tool that requires the source schemas definitions and semantical information from the user. The semantical information include data constraints (*e.g.,* foreign keys), functional dependencies, many-to-many constraints, etc and need to be manually specified by the user. This knowledge, injected by the user, helps the algorithm to generate a richer NBR model.

Premerlani and Blaha [Premerlani 1993] discussed implementation strategies of an object model in the relational model according to their experience. From this experience, the authors use a commercial reverse-engineering tool to reverse-engineer multiple case studies. An informal process for reverse-engineering a relational database and produce an object-oriented model is proposed.

Chiang *et al.* [Chiang 1994] present a methodology to extract Extended Entity-Relationship (EER) schemas from relational-databases based on the meta-data of the relational database and the data stored in the database. A heuristic is applied to determine the relationships between tables and the data are used to evaluate the results of the heuristic.

Polo *et al.* [Polo 2002, de Guzmán 2004, de Guzman 2005, Polo 2007] proposed a tool to generate applications interacting with a database directly from the relational databases schema. To do that, they perform reverse-engineering on the schema of the database to transform it into an instance of a meta-model they designed. This meta-model reifies the database, tables, columns (a column knows if a primary key constraint applies to it via a boolean attribute), foreign keys, and stored procedures. From an instance of their meta-model, the source code of the application is automatically generated.

Yeh *et al.* [Yeh 2008] developed a reverse-engineering approach that extracts EER diagrams from legacy databases. In these legacy databases, primary keys are not declared and the descriptions for some columns are poor. The approach uses

the database schema, data and display forms to generate an EER model.

Wang *et al.* [Wang 2009] propose a model driven approach for relational database reengineering. The authors designed a meta-model to represent the database at a logical level. Instances of this meta-model are transformed into conceptual models conform to an object-oriented system meta-model. The database meta-model reifies the model, the schema, tables and columns. Primary key and foreign key constraints are represented as boolean attributes on column. The authors propose a framework called RDBREF based on EMF which uses these meta-models to perform database reverse-engineering.

Static Program Analysis Petit *et al.* [Petit 1994] propose a method to extract extended entity-relationship schemas from a relational database using the metadata of the database schema and by analyzing SQL queries that an application performs on the database. The authors provide a list of patterns in the source code of SQL queries that allow them to deduce the relationships between tables of the database (*e.g.,* foreign key constraints).

Henrard *et al.* [Henrard 1998, Englebert 1995] present DB-MAIN, a general-purpose database engineering environment. It provides the ability to perform reverse-engineering of the database and programs using it. DB-MAIN can extract schema structure declared into DDL statements and provides a tool to find foreign keys of a schema. For program analysis, DB-MAIN provides utilities to perform static analysis: pattern matching on programs source, variable dependency graph visualization, program slicing and call graphs.

Cleve *et al.* [Cleve 2006] exploit database statements present in programs interacting with a database to compute program slices. Their analysis allows one to create a system dependency graph that includes tables and columns of the database and the system dependency graph of the program using this database. One of the case studies concern a COBOL program containing embedded SQL statements. With their analysis, the authors were able to determinate the set of tables used by the program, the set of columns of these tables used by the program and 32 implicit data dependencies.

Papastefanatos *et al.* [Papastefanatos 2008,Papastefanatos 2010,Manousis 2015] developed Hecataeus, a tool representing the database structural entities, queries and views as a uniform directed graph. This graph is used to simulate a change on the database and compute the impact of the change.

Meurice *et al.* [Meurice 2016a] presented a tool-supported approach able to analyze how the client source code and database schema co-evolved in the past and to simulate a database change to determine client source code locations that would be affected by the change. To extract the relationships between the database and programs using it, the authors used a static analysis approach [Meurice 2016b]. The authors use an entity-relationship model to store multiple versions of the database

and program.

Dynamic Program Analysis Cleve and Hainaut [Cleve 2008] analyze dynamically formed SQL queries occurring in programs using a relational database. The authors trace Java programs to determinate what are the SQL queries generated dynamically that get executed by the RDBMS. Analyzing these queries brings valuable information for database reverse engineering. For example, it can be used to reveal the existence of implicit foreign keys (*i.e.,* foreign key that are enforced by the RDBMS but only by external programs manipulating data).

Alafali *et al.* [Alalfi 2009] present an approach to analyze interactions between a web application and a database. They use a combination of static an dynamic analysis to find links between SQL query executed and the statement that generated the query.

Cleve *et al.* [Cleve 2011a, Cleve 2011b] introduces techniques to perform dynamic SQL queries capture at run time. The authors apply heuristics on these queries to detect implicit foreign keys in the database schema. An experiment has been conducted on a web application and its results suggest that the analysis of SQL traces if effective to achieve database reverse engineering.

3.2.4 Discussion

Table 3.1 lists the approaches related to database reverse engineering reviewed. If we take a look at Crit. 1., we observe that except for [Alalfi 2009], all the reviewed approaches consider structural entities of the database. Nevertheless, most of the approaches consider only tables, columns and foreign keys. This can be explained by the fact that a significant part of these approaches are dedicated to recover implicit foreign key constraints.

On the other hand, behavioral entities of the database are usually not considered. Except for Polo *et al.* [Polo 2002, de Guzmán 2004, de Guzman 2005, Polo 2007] and Papastefanatos *et al.* [Papastefanatos 2008, Papastefanatos 2010, Manousis 2015], all the other reviewed approaches do not handle behavioral entities in the reverse engineering process.

Our approach differs from Polo *et al.* for two main reasons. First, Polo *et al.* reverse engineer the schema of a database with the objective to generate an application that uses the database. Our approach reverse engineer the schema of the database to understand it and support forward engineering. Second, Polo *et al.* only consider stored procedures as behavioral entities in their model. They consider stored procedures to let the application they generate call them. Our approach aims to model all kinds of behavioral entities as we want to be able to evolve a database and adapt all entities impacted by a change.

Papastefanatos *et al.* model views but do not consider other behavioral entities such as stored procedures. Our approach consider all kind of behavioral and structural entities present in the database. We compare further our approach to the one of Papastefanatos *et al.* in Section 3.3.

Recent approaches tend to consider entities defined in programs using the database. This information is used to determinate where, in the source code, programs interact with the database. In this thesis, we focus on behavioral entities defined inside the database and we do not consider entities defined in external programs. Our meta-model represents the relationships between structural and behavioral entities of the database. For example, one can retrieve what tables are used by a stored procedure in a model of a database.

Multiple formalisms have been used to model the database and programs using it (Crit. 2.). Extended entity relationship (EER) model seems to be the most common formalism but many approaches use custom meta-models specifically designed to fit their needs. For our approach, we use a custom meta-model that we present in Chapter 4.

Finally, most of the approaches are fully automatic (Crit. 3.). When a human is involved in an approach, one validates results of heuristics to retrieve implicit metadata of the schema (*e.g.,* implicit foreign keys). The approach to reverse-engineer a relational database that we present in this thesis is fully automatic.

Table 3.1: Comparison of relational database reverse-engineering approaches found in litterature according to our criteria.

Approach	Crit. 1.	Crit. 2.	Crit. 3.
[Markowitz 1990]	Struct.	EER	Auto.
[Castellanos 1993]	Struct.	BLOOM	Human
[Shoval 1993]	Struct.	NBR	Human
[Premerlani 1993]	Struct.	OMT	Human
[Chiang 1994]	Struct.	EER	Auto.
[Petit 1994]	Struct.	EER	Auto.
[Henrard 1998]	Struct.	EER	Human
[Polo 2002]	Struct./Beha.	Custom meta-model	Auto.
[Cleve 2006]	Struct./Ext.	SDG	Auto.
[Cleve 2008]	Struct./Ext.	Not specified	Auto.
[Yeh 2008]	Struct.	EER	Auto.
[Wang 2009]	Struct.	Custom meta-model	Auto.
[Papastefanatos 2008]	Struct./Beha.	Directed graph	Auto.
[Alalfi 2009]	Ext.	Custom meta-model	Auto.
[Cleve 2011a]	Struct./Ext.	Not specified	Auto.
[Meurice 2016a]	Struct./Ext.	ER	Auto.

3.3 Relational Database Impact Analysis

In this section, we review the literature to assess the state of the art related to relational database impact analysis. To do that, we first present software change impact analysis in general and then we present related approaches.

3.3.1 Software Change Impact Analysis

Arnold and Bohnert [Arnold 1996] defined impact analysis as *"[...] the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change"*.

Below, one can find examples of impact analysis listed by Arnold and Bohnert [Arnold 1996]:

- using cross-reference listings to see what other parts of a program contain references to a given variable or procedure;
- using program slicing to determine the program subset that can affect the value of a given variable;
- browsing a program by opening and closing related files (to find out the impact of change "manually");
- using traceability relationships to identify changing artifacts;
- using configuration management systems to track and find changes; and
- consulting designs and specifications to determine the scope of a change.

Since Arnold and Bohnert paper defining impact analysis, the research field has been widely investigated by the scientific community. Meta-analyses on this topic exist, for example, Lehnert did a review of software change impact analysis in 2011 [Lehnert 2011]. In the context of this thesis, we focus on approaches adapting impact analysis techniques to relational databases.

3.3.2 Criteria to Compare Relational Database Impact Analysis Approaches

We identified 6 criteria to compare relational database impact analysis approaches:

Crit. 1. On which kind of entities can the approach specify an initial change?

Possible values: *Structural/Behavioral*

Crit. 2. On which kind of entities can the approach compute the impact of a change?

Possible values: *Structural*/*Behavioral*/*External*

Crit. 3. Does the approach propose changes to accommodate impacted entities with the initial change?

Possible values: ✓/✗

Crit. 4. Can the approach generate a script to evolve impacted entities?

Possible values: ✓/✗

Crit. 5. If a script is generated, does it handle database schema consistency?

Possible values: ✓/✗

3.3.3 Approaches for Relational Database Impact Analysis

Sjøberg [Sjøberg 1993] quantifies schema evolution. The author studied the evolution of a relational database and its application forming a health management system during 18 months. For this purpose, "the Thesaurus" tool analyses the number of screens, actions and queries that may be affected by a potential schema change. This tool shows code locations to be manually modified after a change on the database schema.

Haraty *et al.* [Haraty 2002, Haraty 2004] applied impact analysis to relational database and programs using them to perform regression testing. That is to say, the authors try to find the minimal set of tests that should be re-run to ensure that a change on the database did not break the program behavior. The authors implemented a database applications maintenance tool for ORACLE database application written in PL/SQL.

Gardikiotis and Malevris [Gardikiotis 2006] proposed an approach to estimate the impact of a database schema change on the operability of a web application. To achieve that, the authors proposed a tool named DaSIAn (Database Schema Impact Analyzer) based on their approach. This tool finds CRUD queries affected by a change on the database schema. The authors also presented an approach assessing impact on client applications from schema changes [Gardikiotis 2009]. They used this approach to assess both affected source code statements and affected test suites in the application using the database after a change in the database.

Maul *et al.* [Maule 2008] created a static analysis technique to assess the impact of changing a relational database on its object-oriented software clients. They implemented Schema Update Impact Tool Environment (SUITE) which takes the source code of the application using the database and a model of the database schema as input. Then, they queried this model to find out the part of the source code application impacted when modifying an entity of the database.

Curino *et al.* [Curino 2008,Curino 2009] proposed PRISM, a tool suite allowing one to predict and evaluate schema modification. PRISM also proposes database migration feature through rewriting queries and application to take into account the schema modification. To do so, they provide a language to express schema modification operators. From the change specification, their approach provide automatic data migration support and documentation of changes applied on the database. The authors evaluated their approach and tool on Wikimedia showing it is efficient.

Papastefanatos *et al.* [Papastefanatos 2008,Papastefanatos 2010,Manousis 2015] developed Hecataeus, a tool representing the database structural entities, queries and views as a uniform directed graph. Hecataeus allows users to create an arbitrary change and to simulate it to predict its impact. The simulation is performed by letting the user choose if a node of the graph should propagate the impact or not. It is possible to define rules to describe this impact propagation. A rule might apply for all nodes of the graph or be specific to a limited set of nodes.

Liu *et al.* [Liu 2011, Liu 2013] proposed a graph called the "attribute dependency graph" to identify dependencies between columns in a database and parts of client software source code using it. They evaluated their approach on 3 databases and their clients written in PHP. Their tool presents to the database architect an overview of a change impact as a graph.

Meurice *et al.* [Meurice 2016a] presented a tool-supported approach that is able to analyze how the client source code and database schema co-evolved in the past and to simulate a database change to determine client source code locations that would be affected by the change. Additionally, the authors provide strategies (recommendations and warnings) for facing database schema change. Their recommendations describe, in natural language, how to modify client program source code depending on the change performed on the database. The approach presented has been evaluated by comparing the historical evolution of a database and its client application with the recommendations provided by their approach. From the historical analysis the authors observed that the task of manually propagating database schema change to client software is not trivial. Some schema changes required multiple versions of the software application to be fully propagated. Others were never fully propagated.

3.3.4 Discussion

Table 3.2 lists the approaches related to relational database impact analysis reviewed. We can observe that all analyzed approaches allow one to specify a change on a structural entity of the database (Crit. 1.). This observation seems normal as the code using the database usually interact with tables and columns which are structural entities. Any approach for database impact analysis is thus expected to handle structural entities. Most approaches allow one to specify a change on a view

(behavioral entity) because the impact of such modification is most of the time similar to the impact of a change on a table, especially on external programs. Others behavioral entities are usually not considered.

One approach [Papastefanatos 2008] handles impact concerning structural and behavioral (Crit. 2.) entities of the database. All the other approaches focus on handling the impact on programs using the database under analysis.

Two approaches propose to accommodate impacted entities to the original change (Crit. 3.). Papastefanatos *et al.* [Papastefanatos 2008] approach allows one to modify the graph representing the database schema to fulfill policies set by the database architect. Meurice *et al.* [Meurice 2016a] provide a list of recommendations to accommodate entities depending on the initial change and warning in case there is no recommendation available. These recommendations are not supported by a tool. The other approaches allows one to inspect entities or location in the code that are impacted but the impact needs to be resolved manually.

Curino *et al.* [Curino 2008] approach is able to generate a script to evolve entities detected as impacted by a change (Crit. 4.). Note that this approach does not require to propose changes to accommodate other entities of the database as only tables and columns are considered. All the other approaches propose a description of the changes to apply or provide an impact report highlighting the code location to update.

We found no article that describe the problem of maintaining the schema consistency during the execution of the SQL script (Crit. 5.). Curino *et al.* [Curino 2008] generate a script to migrate the database but do not discuss this problem. However, as the authors handles tables and columns only, they might not need to handle this problem.

Table 3.2: Comparison of relational database impact analysis approaches.

Approach	Crit. 1. Initial change	Crit. 2. Impacted entities	Crit. 3. Change proposition	Crit. 4. Script generation	Crit. 5. Schema consistency
[Sjøberg 1993]	Struc.	Ext.	✗	✗	✗
[Haraty 2002]	Struc./Beha.	Ext.	✗	✗	✗
[Gardikiotis 2006]	Struc./Beha.	Ext.	✗	✗	✗
[Gardikiotis 2009]	Struc./Beha.	Ext.	✗	✗	✗
[Maule 2008]	Struc./Beha.	Ext.	✗	✗	✗
[Curino 2008]	Struc.	Ext.	✗	✓	✗
[Papastefanatos 2008]	Struc./Beha.	Struc./Beha.	✓	✗	✗
[Liu 2011]	Struc.	Ext.	✗	✗	✗
[Meurice 2016a]	Struc.	Ext.	✓	✗	✗

Via the literature review, we observe that approaches to perform impact analysis are usually focused on computing the impact on external programs using the

database. In this thesis, we develop an approach that focus on impact analysis inside the database concerning both structural and behavioral entities. Furthermore, we develop an approach that can generate a SQL script containing DDL statements to update the database according to data gathered during the impact analysis.

3.4 Conclusion

In this chapter, we reviewed the scientific literature concerning *relational database reverse engineering* and *relational database impact analysis*. As a conclusion of this literature review, we can say that approaches for relational database reverse engineering usually do not consider behavioral entities. Impact analysis approaches mainly focus on computing the impact of a change made to a structural entity of a database on external programs interacting with the database. Approaches that handle the impact on behavioral entities do not allow to generate a patch to accommodate the database with the initial change. Furthermore, the problem of maintaining schema consistency in the script generated by the approach is not handled by the approaches we reviewed.

In the next chapters, we develop an approach that aim to fill the gap we found in the literature. Chapter 4 presents a meta-model that represents both structural and behavioral entities of the database together with the relations between these entities. However, we do not handle relationships between a database and external programs using them. Chapter 5 presents our results using our meta-model to assess the quality of a relational database. Chapter 6 presents our approach involving impact analysis to evolve a relational database semi-automatically. This approach is able to generate a script to migrate the database and this script fulfill the schema consistency constraint.

CHAPTER 4

A Behavior-Aware Meta-Model for Relational Databases

Contents

In this chapter, we present a meta-model for relational databases and the approach we use to instantiate models. The meta-model takes into account both structural and behavioral entities of a database. Relationships between entities of the meta-model are reified to ease change impact computation. We present and compare two approaches to instantiate the meta-model. Then, we propose an approach that is a trade-off between the two previous approaches.

Furthermore, via 2 case studies, we illustrate that instances of the meta-model address the problem *"Relational databases are hard to understand"* presented in Chapter 2. This chapter is an extension of our publication at the International Conference on Advanced Information Systems Engineering (CAiSE) [Delplanque 2020].

4.1 Objectives

The design of our SQL meta-model is driven by one of the problem we identified in Chapter 2: *Relational databases are hard to understand*. As a reminder, this problem is caused by 2 main sub-problems:

1. *Complexity to query metadata.* One of the reasons of this complexity is that there is no uniform way to query entities in relation with a given entity. A good example of this problem is illustrated by one of Laurenz Albe's blog post [Albe 2019]. In this blogpost, the author illustrates how PostgreSQL meta-data tables can be used to retrieve the views that depend on a table `t1`:

```
SELECT v.oid::regclass AS view
FROM pg_depend AS d -- objects that depend on t1
   JOIN pg_rewrite AS r -- rules depending on t1
      ON r.oid = d.objid
   JOIN pg_class AS v -- views for the rules
      ON v.oid = r.ev_class
WHERE v.relkind = 'v' -- only interested in views
  -- dependency must be a rule
  -- depending on a relation
  AND d.classid = 'pg_rewrite'::regclass
  AND d.refclassid = 'pg_class'::regclass
  AND d.deptype = 'n' -- normal dependency
  AND d.refobjid = 't1'::regclass;
```

This query can be tweaked to retrieve the other kind of depending entities or to retrieve recursively all depending views (via a recursive `SELECT` query[1]).

However, the query need to be adapted to the kind of entity for which one wants to gather dependent entities. For example, to gather entities that depend on a column, the query is different. Similarly, to gather depending entities of a given entity, a specific query will need to be written for each kind of entity. For example, to gather tables that a view depends on, the query is different. A uniform way to query dependent and depending entities is needed.

2. *Missing metadata.* Some entities in meta-data tables have incomplete meta-data. In PostgreSQL, stored procedures source code is not meta-described in meta-data tables. Thus, the RDBMS is blind concerning the entities that are used by a stored procedure (*e.g.,* the other stored procedures it calls, the tables it references, etc.).

   Missing meta-data need to be retrieved to be able to get consistent results from queries on metadata.

In this chapter, we design a meta-model that aim to address this problem and its sub-problems. Thus, the development of the meta-model is driven by two objectives:

1. *Model the structure and behavior of the database.* Fulfilling this objective allows us to address problem 1 as an instance of the meta-model is a graph data structure. Representing the database schema as a graph (rather than using tables and views as done by the RDBMS) makes it easier to query

[1] https://www.postgresql.org/docs/current/queries-with.html

for properties related to dependency analysis (*e.g.,* computing a transitive closure to determinate entities impacted by a change recursively).

2. *Ease dependencies analysis.* This objective addresses problem 2 as to ease the dependencies analysis, having an explicit representation of the relationships between entities is needed. Being able to query a model to gather incoming (*e.g.,* views that reference a table) and outgoing (*e.g.,* tables that are referenced by a stored procedure) relationships of an entity is critical to perform impact analysis (which is needed in Chapter 6).

Objective 1 is fulfilled by modeling tables, columns and constraints as well as CRUD[2] queries, views, stored procedures, and triggers.

Objective 2 is fulfilled by reifying relationships entities. Because all dependency relationships are first-class citizens in the meta-model, no dependency is implicit anymore.

If we compare to the database reverse-engineering proposed by Hainaut et al. [Hainaut 2009] that we presented in Chapter 3, the meta-model corresponds to the logical schema of the database. Indeed, the meta-model is specialized to represent relational database schema but it is not tight to a particular RDBMS.

4.2 The Meta-Model

In this section, we present the meta-model and describe each kind of entity it represents. We distinguish three kinds of entities:

- *Structural entities*: entities defining the structure of data stored in the database or defining constraints applied on data (for example, tables).
- *Behavioral entities*: entities defining behavior (*i.e.,* source code that can be executed) that may interact with structural or behavioral entities (for example, stored procedures).
- *References*: entities representing links between entities (for example, the fact that a primary key constraint references a column in a table).

The following subsections present separately each kind of entities to ease readability. For the next UML diagrams, inheritance links have straight corners while other links are rounded; classes modeling structural entities are colored in red; classes modeling behavioral entities are colored in orange; and classes modeling references are colored in white. These conventions are not part of UML standard but are used to ease readability of diagrams.

[2]Create Read Update Delete queries in SQL: INSERT, SELECT, UPDATE, DELETE.

4.2.1 Structural Entities

Figure 4.1 shows the structural part of the meta-model. As a reminder, a structural entity is an entity defining the structure of data held by the database or defining constraints applied on these data (*e.g.,* table, column, referential integrity constraint, etc.). All entities falling under this definition inherit from StructuralEntity.

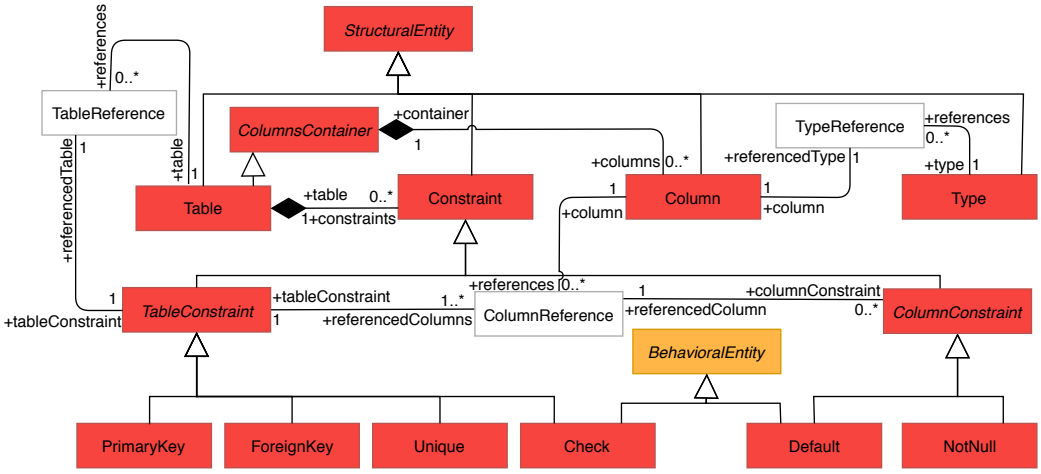

Figure 4.1: Structural entities of the meta-model.

Table models a relation in the relational database and Column models an attribute. The containment relation between Table and Column is modelled through ColumnsContainer which is an abstract entity modelling the fact that an entity contains Columns. This entity is also subclassed in the behavioral part of the meta-model (see Section 4.2.2).

A Column has a type. This relation is modelled by referencing a TypeReference entity linked to the corresponding Type.

A Column can also be subject to Constraints. Depending on whether a Constraint concerns a single or multiple columns (one or many ColumnReference), it is either a column or a table constraint and it inherits from, respectively, ColumnConstraint or TableConstraint.

Six concrete constraints inherit from Constraint.

PrimaryKey and ForeignKey model respectively the primary key and foreign key constraint as defined in the relational model [Codd 1970b].

Unique specifies that data contained in rows formed by the set of constrained Columns must be unique within the table. For example, if two integer columns `a` and `b` are constrainted by a unique constraint, the row `(1,2)` can not appear twice.

Check is a developer-defined constraint defined by a boolean expression: if the expression evaluates to false, the constraint is not respected.

Default is a default value assigned when no value is explicitly provided for a column when a row is inserted in a table. The "default value" can be a literal value or an expression to compute.

NotNull implies that values of a column can not be NULL.

Note that Check and Default constraints also inherit from BehavioralEntity because they have behavior: the expression to execute.

4.2.2 Behavioral Entities

A behavioral entity is an entity holding behavior that may interact with StructuralEntities or BehavioralEntity. All entities falling under this definition inherit from BehavioralEntity. Figure 4.2 shows the behavioral part of the meta-model.

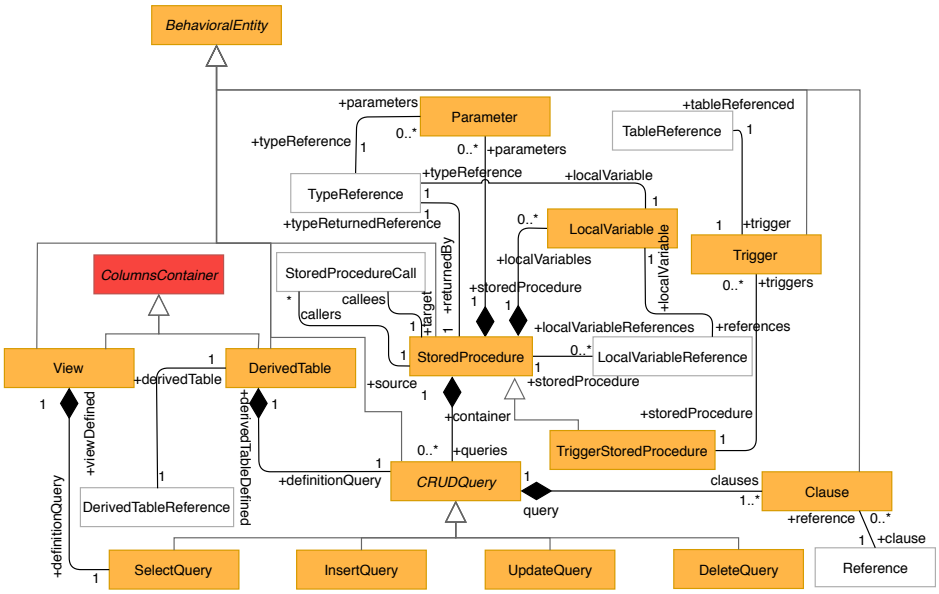

Figure 4.2: Behavioral entities of the meta-model.

A View is a named entity holding a SELECT query.

StoredProcedure is an entity holding developer-defined behavior which includes queries and reference to other entities. A StoredProcedure contains Parameter(s) and LocalVariable(s). Furthermore, a StoredProcedure can contain References to other entities in its statements as well as CRUDQueries. As explained later, CRUDQueries themselves can contain references to other entities through their clauses. However, for the analysis it is interesting to know if a reference is made from a CRUDQuery or from another statement of the stored procedure.

Trigger models actions performed in response to an event happening on a table (*e.g.,* row inserted, updated or deleted). Thus, it has a TableReference that reference

the table the trigger watches for events. A trigger also have a reference to a TriggerStoredProcedure. This is a special kind of stored procedure that, when called, can make reference to the old and new values of the row to be modified.

DerivedTable is an anonymous query usually used in another query but that can also appear in a cursor declaration of a StoredProcedure.

CRUDQuery and its subclasses model CRUD queries. Each subclass of CRUDQuery contains one or many clauses depending on the query. The possible clauses are: With, Select, From, Where, Join, Union, Intersect, Except, GroupBy, OrderBy, Having, Limit, Offset, Fetch, Insert, Into, Returning, Update, Set, Delete. These clause entities are not shown in Figure 4.2 to not clutter the diagram. However, Table 4.1 describes which clauses are contained by which kind of CRUDQuery. Each clause holds zero or many References to structural or behavioral entities. The references made to structural or behavioral entities from clauses are detailed in Section 4.2.3.

Table 4.1: Containment relations between CRUD queries and clauses.

CRUD query	Clauses
SelectQuery	With, Select, From, Where, Join, Union, Intersect, Except, GroupBy, OrderBy, Having, Limit, Offset, Fetch
InsertQuery	With, Insert, Into, Returning
UpdateQuery	With, Update, Set, From, Where, Returning
DeleteQuery	With, Delete, From, Where, Returning

4.2.3 References

The last part of the meta-model represents links between entities. It allows one to track relations between behavioral and structural entities. To simplify this task, all references have been reified. For example, a column is referenced through a ColumnReference, a local variable through a LocalVariableReference and a stored procedure through a StoredProcedureCall. Table 4.2 gathers the different entities that each clause of a query may refer to.

Let us explain some non-obvious references that can appear in some clauses. The first line of the table specifies that a reference to a derived table can appear in any clause. This comes from the fact that a `SELECT` query can occur in any SQL expression. Line 2 specifies that it is possible for a StoredProcedure to generate a table as a result, this is why it can appear in a From clause. Line 3 specifies that a reference to Table or View can appear in clauses that normally deal with column references. It comes from the fact that a developer can use a qualified reference to a column (*e.g.,* `table_name.column_name`). Finally, line 3 and 4 specify that a stored procedure local variable or parameter can appear in their respective

set of clauses. It occurs, for example, when the filter condition of a `WHERE` clause is parameterized by a local variable or a parameter. For example, `WHERE id = id_to_keep` where `id` would be a reference to one of the columns of a table appearing in the `FROM` clause of the query and `id_to_keep` would be a reference to a local variable declared in the stored procedure holding the query.

Table 4.2: References to entities from clauses. The other references to entities are shown in Figure 4.1 and 4.2.

	Clauses	Entities
1	Any clause	DerivedTable
2	From, Join, Into	StoredProcedure, Table, View.
3	GroupBy, OrderBy, Having, Set, Where, Returning, Select, Update	Table, View, Column, StoredProcedure, LocalVariable, Parameter
4	Limit, Fetch, Offset	StoredProcedure, LocalVariable, Parameter

4.3 Meta-model Instantiation

To analyze a database, we need to instantiate the meta-model. The instantiation of the meta-model corresponds to the process of reverse-engineering of a database schema as described by Hainaut *et al.* [Hainaut 2009] that we explained in Chapter 3. More precisely, in this section, we perform the steps of *DDL code analysis*, *Schema refinement* and *Schema cleaning* described by Hainaut *et al.*.

In this section, we explain how two different approaches achieve this task and how they compare together. The first one uses meta-data tables and the second analyses a dump of the database. To compare these approaches, we introduce some evaluation criteria. Then, we present our approach to instantiate the meta-model which is a hybrid between the two previous ones, taking the best of both approaches.

4.3.1 Evaluation Criteria for Meta-Model Instantiation

Before presenting the different approaches, we introduce criteria that allow one to compare them. We identified three relevant criteria to compare our approaches.

1. **Completeness**: Capability of the approach to import various types of entities populating the database into the model.

   Possible values for this criteria:

- *complete*: The approach can import all types of entities in a model.
- *incomplete*: The approach can not import all types of entities in a model.

2. **Sensitivity to RDBMS evolution**: Degree to which the approach implementation needs to be updated when evolution of the RDBMS occurs. This criterion measures how many hypotheses on a specific version of the RDBMS the approach makes.

   Possible values for this criteria:

   - *low*: An evolution of the RDBMS induces little and well identified update(s) to the approach implementation.
   - *limited*: An evolution of the RDBMS induces a limited update on the approach implementation. In particular, it does not require to update all steps of the approach.
   - *high*: An evolution of the RDBMS induces a large update to all steps of the approach implementation.

3. **Complexity to migrate approach to another RDBMS**: The degree to which the implementation must be modified in order to build a model for another RDBMS. This criterion is a measure of how many hypotheses on a specific RDBMS are made by the implementation of the approach.

   Possible values for this criteria:

   - *simple*: Porting the approach implementation to build a model for another RDBMS requires no or little modification(s).
   - *moderate*: Porting the approach implementation to build a model for another RDBMS requires a moderated amount of modifications. More specifically, it does not require to updated all steps of the approach.
   - *complex*: Porting the approach implementation to build a model for another RDBMS requires substantial modifications to all steps.

4.3.2 Meta-Data Analysis

The first and probably simplest approach to build a model (*i.e.,* instantiate the meta-model) of the database is to exploit meta-data provided by the RDBMS. This approach is used by database IDEs to display information related to the database schema to the user. These meta-data take the form of tables and views that one can interrogate via SQL queries.

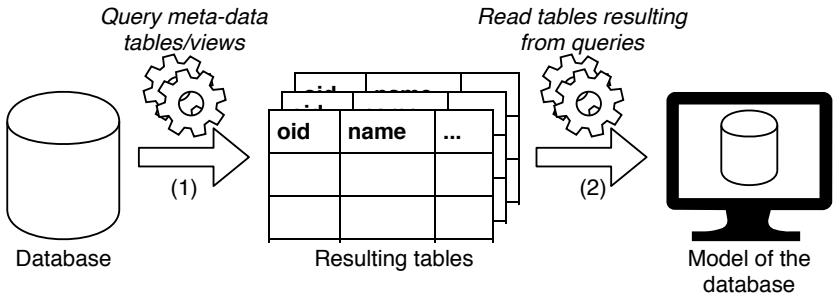

Figure 4.3: Instantiation of the model using meta-data tables only.

As illustrated in Figure 4.3, this approach consists in two steps. The first step (1) executes queries on the database to retrieve meta-data related to each kind of entity reified in the meta-model. For example, Figure 4.4 shows a SQL query to retrieve tables defined by a PostgreSQL database.

```
1  SELECT
2      -- Unique id of the entity.
3      oid,
4      -- Name of the table.
5      relname,
6      -- Id of the namespace.
7      relnamespace
8  FROM pg_class
9  -- Specify we want tables
10 -- ('r' is for relation).
11 WHERE
12     relkind = 'r';
```

(a) Query on meta-data table.

oid	relname	relnamespace
	...	
442	'person'	23
443	'phd'	23
444	'city'	25
	...	

(b) Possible result of the query.

Figure 4.4: Query gathering unique identifier, name and identifier of the namespace of tables stored in a database.

Similar queries can be used to retrieve other kinds of entities reified in the meta-model. For example, using results in Figure 4.4, we can query metadata tables to retrieve information of the namespace with oid 25 to know more about 'city' table namespace. In practice, we use more complex queries on meta-data tables that, for example, join columns with their tables with their namespace.

The second step (2) of the approach simply read tables resulting from queries and build entities of the model according to the data they hold.

Evaluation

1. **Completeness**: *incomplete*.

   The model built does not contain references made from behavioral entities source code to other entities. This incompleteness is directly linked to the missing meta-data problem described in Chapter 2.

2. **Sensitivity to RDBMS evolution**: *low*.

   System tables are subject to little breaking changes across RDBMS evolution. When a change occurs, it is usually to add new tables or new columns. Thus, existing queries should continue to work most of the time.

3. **Complexity to migrate to another RDBMS**: *simple*.

   We consider this approach simple to migrate compared to the other approaches because it requires a few changes to make it work on another RDBMS. Queries most probably need to be modified to work on meta-data tables of another RDBMS. However, only queries need to be updated, the code that reads results of queries to build the model can stay the same as long as tables resulting from these queries have the same schema.

Summary Extracting meta-data from the database provides a simple approach that can quickly be set up to create a model of the database. However, this approach has a major drawback: the model produced is incomplete. Indeed, as reminded at the beginning of this chapter, the RDBMS misses some meta-data concerning behavioral entities (*e.g.,* references made from the source code of a stored procedure to a table for example).

4.3.3 Dump Analysis

To build a model of a program, one can analyze its source code statically. Following this idea, we can consider doing the same thing to analyze relational databases.

A RDBMS is able to generate a "dump" of the database. A database dump is a text file containing SQL code allowing one to recreate the database in the exact same state as when it was exported. It is a way to serialize the database. One can request the RDBMS to produce a dump containing the database schema, the data or both.

Figure 4.5 illustrates the process to instantiate a model of the database from a dump file. This process consists in two steps. First, one needs a parser for the full SQL grammar. This parser reads the SQL source code contained in the file and produces an abstract syntax tree (i). Second, the abstract syntax tree (AST) needs to be analyzed to produce the model of the database (ii). A complex static analysis

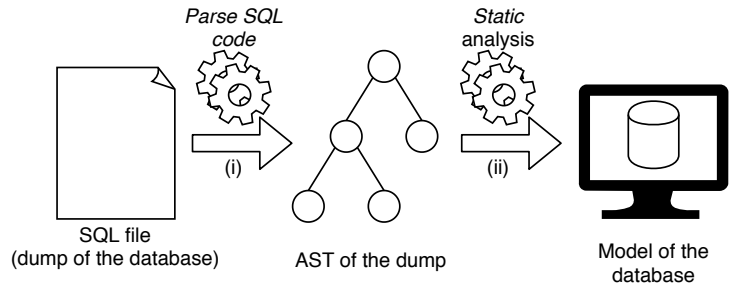

Figure 4.5: Instantiation of the model by parsing a dump of the database.

of the AST is required to create a model. Indeed, we tried this approach and we realized that often PostgreSQL generates the code to create an entity and later in the dump generates queries to modify this entity. For the sake of illustration, here is an example of what we observe in the dump generated for WikiMedia (the open source collaborative editing software project that runs Wikipedia) 1.27.1 database:

```
164 CREATE TABLE archive (
165     ar_id integer DEFAULT nextval('archive_ar_id_seq'
          ::regclass) NOT NULL,
166     ar_namespace smallint NOT NULL,
167     ar_title text NOT NULL,
168     ...
```

The table `archive` is created at line 164 in the dump file. Later in the dump file, at line 1142, it is modified by an `ALTER TABLE` DDL query.

```
1142 ALTER TABLE ONLY archive
1143     ADD CONSTRAINT archive_pkey PRIMARY KEY (ar_id);
```

Similar behavior occurs for other kinds of entities. Because of that, we have to set-up a complex static analysis mechanism. We have to mimic the SQL interpreter of the RDBMS so the model produced is consistent with the database built by the RDBMS once the dump has been executed.

Evaluation

1. **Completeness**: *complete*.

   Because the entire source code of the dump of a database is analyzed, the source code of all entities is considered. Thus, the approach creates a model that represent all kind of entities used by the database.

2. **Sensitivity to RDBMS evolution**: *high*.

While as discussed in the previous approach, system tables are rarely subject to a breaking change, the grammar of the SQL dialect used by RDBMS evolves more frequently. For example, if we compare the grammar of the `CREATE TABLE` command between PostgreSQL version 10[3] and version 11[4], we can observe changes in the command syntax. For instance, the apparition of `HASH` keyword in `PARTITION BY` clause of the command. Knowing that a little change in the grammar requires to modify the parser, the AST and the static analyzer of the AST, we consider that this approach is highly sensitive to RDBMS evolution.

3. **Complexity to migrate to another RDBMS**: *complex*.

   This approach is specific to a particular RDBMS. Indeed, each has a slightly different version of SQL language. For example, PostgreSQL support inheritance between tables. It is thus possible to specify the super-table of a table from the SQL syntax of `CREATE TABLE` statement. MySQL does not support this feature, thus it is not supported by the syntax. Both the parser and the static analysis of ASTs make a lot of assumptions on the SQL flavor used by the RDBMS and how it is interpreted (to mimic the SQL interpreter of the RDBMS and get a consistent model). Because of that, we consider this approach complex to migrate.

Summary Analyzing a dump of the database to build a model has the advantage that the source code of all kind of entities of the database is taken into account. In particular, the source code of behavioral entities is analyzed and references to other entities made from the source code are taken into account. Thus, the model built by this approach is more complete than the one built from meta-data analysis. However, two major drawbacks arise. First, a parser covering the entire SQL grammar is required. Considering the expressiveness of SQL language, building this parser is a complex and time-consuming task. Second, if a parser reading SQL source code from a dump and producing an AST is available, one need to mimic the execution of some of the available SQL commands to build the model as we saw previously in our example. These two tasks are engineering challenges and can not be easily ported from one RDBMS to another because they use a lot of properties of the RDBMS for which they were designed.

4.3.4 Hybrid Approach

The approach we use to instantiate the meta-model takes the best parts of the two previous approaches while minimizing their downsides. The idea is to extract as

[3] https://www.postgresql.org/docs/10/sql-createtable.html
[4] https://www.postgresql.org/docs/11/sql-createtable.html

much information as possible from the RDBMS meta-data tables and views to get a partial model of the database and then complete this model by analyzing the source code of behavioral entities. Using this approach allows one to reduce the complexity of the source code analyzer as only a subset of the grammar needs to be covered (*i.e.,* being able to parse views and stored procedures bodies). Reducing this complexity is interesting as the implementation of the source code analyzer is costly in engineering time.

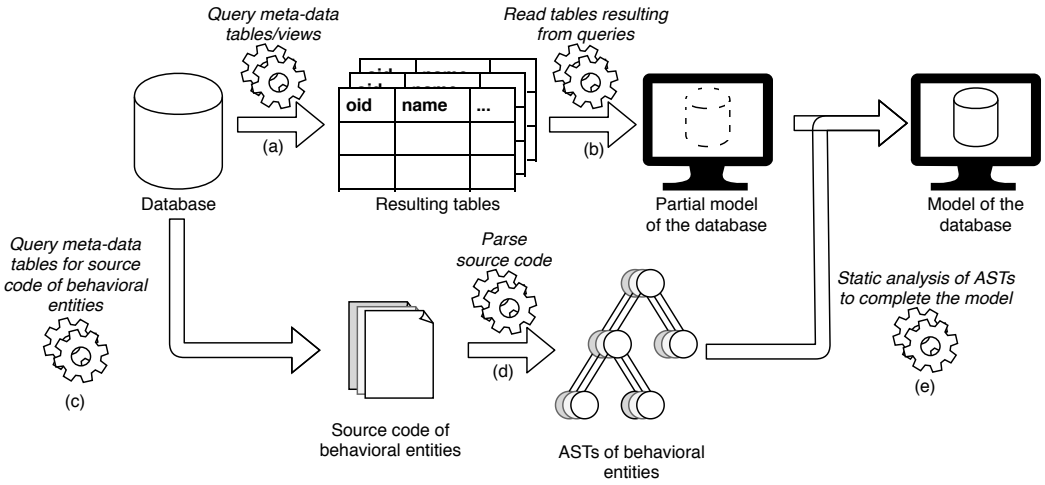

Figure 4.6: Instantiation of the model via an hybrid approach: query meta-data tables and parse source code of behavioral entities.

Figure 4.6 illustrates how the hybrid approach works at coarse grain. Let us provide more details on each step:

(a) The first step consists in extracting as much information as possible from the RDBMS meta-data (similarly to what is done in Subsection 4.3.2).

(b) We read the tables resulting from step (a) and instantiate a partial model.

The implementation of steps (a) and (b) relies on PgMetadata[5] v1.0.0, a library that creates objects modelling the database schema. These objects are not instances of our meta-model classes, thus, we convert them in this step. The result is a partial model that misses references made from behavioral entities.

(c) We interrogate metadata of the RDBMS to gather the source code of each behavioral entity instantiated in step (b). We get a collection of strings containing the source code of those behavioral entities.

[5] https://github.com/olivierauverlot/PgMetadata

(d) We apply the parser on each string of source code produced by step (c). To achieve this step, we developed a parser for *a subset of PostgreSQL language*[6]. This parser accepts the PlpgSQL language and CRUD queries. Data Definition Language is not supported by the parser but it is not required as it can be retrieved from the meta-data (step (a)). Data Control Language is not supported either by the parser but it is not required as we do not need this information. A parser accepting only PlpgSQL language and CRUD queries is simpler than a parser that accepts the whole PostgreSQL language and reduce both the development cost of the parser and the sensibility of the approach to RDBMS evolution. The output of this step is a collection of abstract syntax trees (ASTs).

(e) Finally, we perform a static analysis of the ASTs produced in step (d). This analysis consists in two tasks: we instantiate queries and clauses entities so they are represented in the model and we analyze identifiers appearing in the ASTs to create reference entities in the model.

Evaluation

1. **Completeness**: *complete*.

   The approach combines meta-data analysis and static analysis of behavioral entities source code. Thus, it can create models that represents all types of entities in a database together with references they make to other entities.

2. **Sensitivity to RDBMS evolution**: *limited*.

   RDBMS evolutions will not impact all steps of the approach. Indeed, only steps related to the analysis of static analysis of source code of behavioral entities (steps (d) and (e)) are subject to complex modifications. Steps (a), (b) and (c) have a low sensitivity to RDBMS evolution for the reasons developed in subsection 4.3.2.

3. **Complexity to migrate to another RDBMS**: *moderate*.

   We qualify this approach as moderately complex to migrate. As it uses both meta-data queries and a parser for behavioral entities source code, the meta-data part of the approach is simple to migrate and the parser part is complex. However, as the parser handles a smaller grammar than the dump analysis approach, it requires less work to migrate to another RDBMS.

[6] https://github.com/juliendelplanque/PostgreSQLParser

Summary The hybrid approach combines meta-data analysis and source code analysis of behavioral entities of the database. It provides a tread-off between the two previous approaches by relying as much as possible on meta-data analysis and, where it is needed, by deploying a more complex static analysis of the source code. The model built is complete and evolution of the RDBMS has a limited impact on the approach. However, a part of the approach is specific to a particular RDBMS as the parser and static analysis need to be designed with high assumptions on the SQL code to be analyzed. Table 4.3 compares the three approaches according to the criteria we defined previously. The hybrid approach takes the best from meta-data and source code analysis approaches. The implementation of the hybrid approach is open-source and available at https://github.com/juliendelplanque/FAMIXNGSQL.

Table 4.3: Comparison of approaches to build a model of a database.

Approach	**Completeness**	**Sensitivity to RDBMS evolution**	**Complexity to migrate to another RDBMS**
Meta-data Analysis	*incomplete*	*low*	*simple*
Dump Analysis	*complete*	*high*	*complex*
Hybrid Approach	*complete*	*limited*	*moderate*

4.4 Case studies

In this section, we present two case studies illustrating that our meta-model addresses the two problems discussed in Section 4.1: *complexity to query metadata* and *missing metadata*. For both case study, we built a model of the "2019-12-12" version of AppSI database schema (AppSI database was presented in Chapter 2). For the two case studies, we participated to the experiment with AppSI database architect. However, we have fewer knowledge of AppSI than its architect (who have a deep knowledge of the database).

4.4.1 Queries to support (re)modularization

For this case study, we query a model to understand the schema of AppSI. This understanding helped the architect of the database taking decisions concerning AppSI evolution. In particular, it helped in the extraction of a set of views of the database into a new namespace.

This case study is an illustration of how the meta-model solves sub-problem 1 discussed in Section 4.1. Indeed, to extract the set of views, each view needs to be

taken separately and the model is queried to retrieve all entities that depend on the view or that the view depends on.

Context At the University of Lille, two instances of AppSI database schema are running in two different laboratories. We will refer to those laboratories as "Laboratory 1" and "Laboratory 2".

An instance of AppSI database schema can be used by multiple applications. In the context of Laboratory 1, one of them is a web application that uses a set of views of the database. These views are used by no other application. The names of these views start with "`web_`" prefix. In the rest of this section, we refer to these views as "*web views*".

Figure 4.7 illustrates the situation. Laboratory 1 has one web application that uses web views. Laboratory 2 has no application that uses web views.

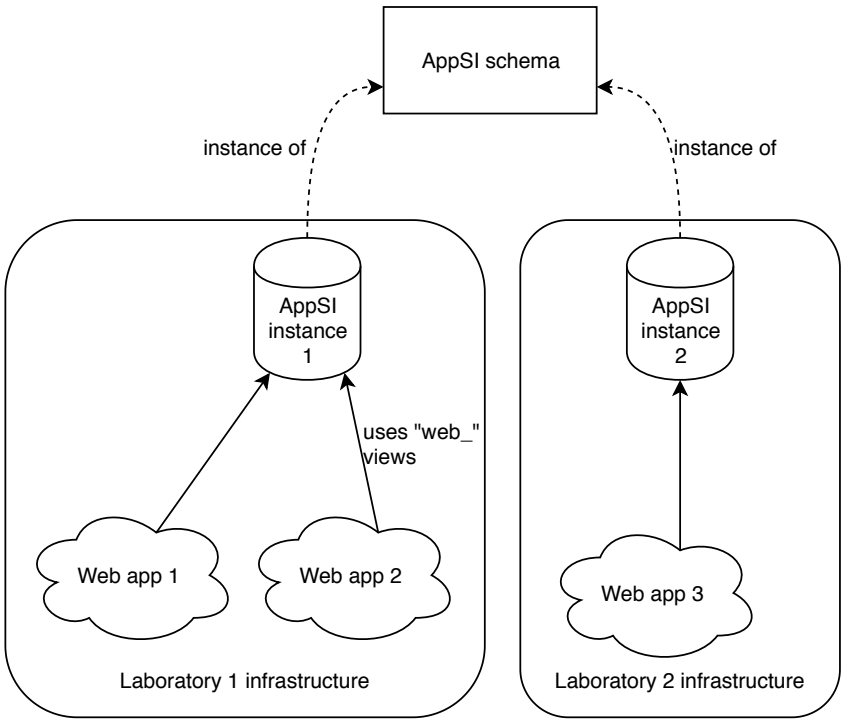

Figure 4.7: Context of the case study.

Problem The database architect would like to extract web views in a separated namespace. It will make it easier to create instances of AppSI without these views. Thus, Laboratory 2 database (that does not require web views) can get them removed by removing the related namespace.

To move web views in a separated namespace, one needs to know what are the dependencies between these views and the rest of the database. The current

database architect of AppSI does not have this information as he is not the person who implemented the web views.

4.4.2 Methods

To tackle this problem we:

1. Extract dependency relationships between web views and the rest of the database.
2. Analyze dependencies to take a decision on how to move web views to a separated namespace.

The first step is performed by querying a model of the database that we created using the approach described in Chapter 4. A query is executed on the model to gather web views. Then, two queries interrogate the model for entities *depending* on web views and for entities the views *depends on*. The result of these queries is visualized to understand the relationship between the web views and the rest of the database.

The second step consists in deciding, according to data collected during first step, how to create a new namespace and move web views in it.

Analysis Web views of the database are retrieved by executing the following query on the model:

```
1 webViews := (model allWithType: FmxSQLView)
2   select: [ :v | v name beginsWith: 'web_' ].
```

The above query is a Smalltalk script gathering views available in a model (line 1) and selecting those with a name prefixed by `'web'` (line 2).

Entities depending on web views From this set of web views we can retrieve, for each view, the set of entities of the database that *depend on the view* (*i.e.,* those using the view). The query to extract this information from the model is implemented by the following Smalltalk script:

```
1 scopes := {
2   FmxSQLTable. FmxSQLView.
3   FmxSQLStoredProcedure. FmxSQLTrigger
4 }.
5 webViews collect: [ :v |
6   v -> ((v queryAllIncoming allAtAnyScope: scopes)\{v})
7 ] as: Array.
```

In the code above, at line 6, `queryAllIncoming` method is interrogating the model for all incoming references pointing to the current view `v`. This set of incoming references is then scoped to only get tables, views, stored procedures and triggers as depending entities (`allAtAnyScopes:`). This scoping is needed because `queryAllIncoming` returns all kinds of entities referencing the view. For example, consider the database in Figure 4.8. If we call `queryAllIncoming` method on view `v1`, we get the `SELECT` clause and `FROM` clause entities from `v2` as result. To ease readability, we scope these entities to the main entities of the meta-model: tables, views, stored procedures and triggers. By scoping the result of the query, we get as result the view `v2` in our example database of Figure 4.8.

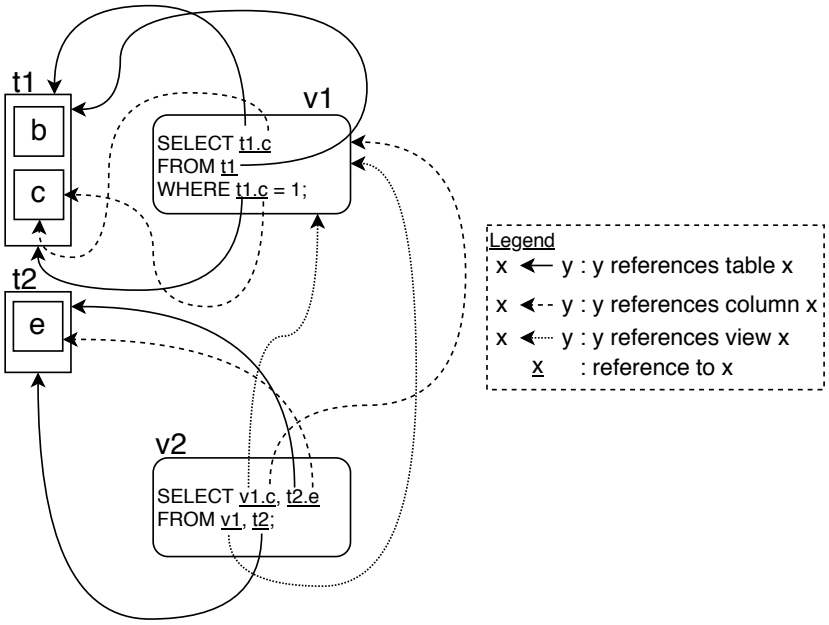

Figure 4.8: Example of database model with references between entities.

It can happen that the `SELECT` query of a view defines a derived table (or sub-query) in its `FROM` clause and refers to it in its `SELECT` clause. Figure 4.9 illustrates this case.

In the case of a view defined by a `SELECT` query with a sub-query, the expression `(v queryAllIncoming allAtAnyScope: scopes)` will result in a set of entities containing the view `v` itself because of the scoping. To address this issue, the views for which we query the model for incoming references are removed from the results (`\{v}`).

The result of the query retrieving entities depending on web views of AppSI revealed that no entity of the database depends on a web view. It means that the database can work without this set of views as they are only used by external programs.

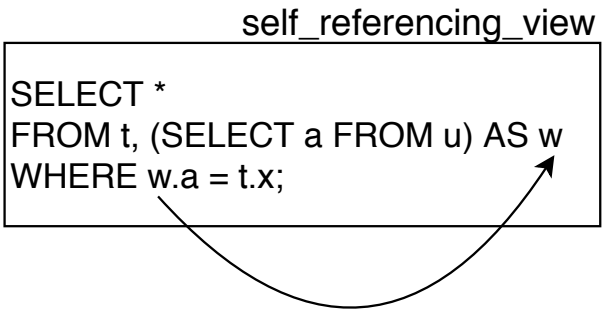

Figure 4.9: Example of self referencing view. With scoping applied to `queryAllIncoming` method result, it cause the `selfReferencingView` to reference itself.

Entities web views depend on We now investigate outgoing dependencies from web views to the rest of the database. We write a query on the model that retrieve, for each web view, the set of entities of the database *the view depends on*. The query to extract this information from the model is implemented by the following Smalltalk script:

```
1 webViews collect: [ :v |
2   v -> ((v queryAllOutgoing allAtAnyScope: scopes)\{v})
3 ] as: Array.
```

The above query is similar to the one querying entities depending on web views in the previous section. The difference is that the method `queryAllOutgoing` (line 2) is called instead of the method `queryAllIncoming`.

The result of this query revealed which entities of AppSI are required by web views.

This result is shown in Figure 4.10. Exploring the different dependencies between entities provides valuable information to the database architect when it comes to understand how web views use the rest of the database.

Conclusion With the knowledge we gained by exploring the database schema via these queries on the model (for instance, the fact that no entity in the database depends on a web view), we were able to generate a simple SQL script that creates a new namespace and moves the web views in it. We provided this script to the database architect and he executed it on AppSI.

The script was run without error and AppSI evolved successfully. The database architect reported that he updated `SELECT` queries of the web application using web views to prefix reference to web views with the name of the namespace created. After this change, the web application was able to run correctly.

Additionally, this case study illustrates that the meta-model provides a uniform

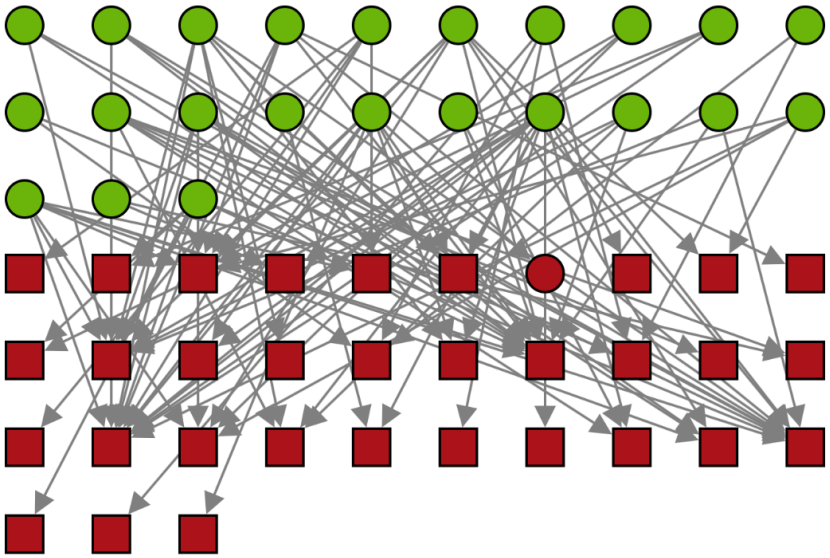

Figure 4.10: Visualisation of query extracting entities that web views depends on. Squares represent tables, circles represent views. Green color represent web views and red color represent other entities. An arrow between a web view (green entity) and another entity (red entity) means that the web view depends on this entity.

and simple way to query dependent and depending entities. This observation suggests that the meta-model address *"complexity to query metadata" problem*.

4.4.3 Retrieving Features Implemented by a Database

In this section, we report our experiment in reverse-engineering the whole AppSI database for documentation purposes.

The goal of the experiment is to tag entities of the database with the name of the feature they implement. With that additional meta-information available for all entities of the database, future maintainers can more easily understand how the database is organized. For that purpose, we developed a heuristic that, starting from tables that are manually tagged, propagate the tag to depending entities. We evaluated this heuristic with the help of AppSI architect.

This case study illustrates how the meta-model addresses sub-problems 1 and 2. Indeed, to retrieve the features implemented by AppSI, dependency analysis was required involving stored procedures. Because of that, the sub-problem 2 is addressed. Indeed, to include stored procedures in the dependency analysis, their missing metadata need to be retrieved and represented in the model.

Context AppSI, as any information system, evolves to fulfill new requirements of the laboratory. The needs of the laboratory are quite various: store data related

to human resources, manage mailing lists, manage PhD students, manage buildings keys, etc.

Problem Because the needs of the laboratory are various, AppSI implement various features. The problem is that these features are not clearly defined because they emerged from successive evolutions of the database. As AppSI gets bigger and bigger, it becomes complicated to have an overview of what are the different parts of AppSI and how they work together. It would be good to analyze AppSI database schema to document its different parts. More specifically, it would be useful to know for what feature an entity of AppSI was implemented.

Methods To tackle this problem, we created a model of the database and used the following four-steps approach:

1. *Identify features from tables in the database.*
   Input = The schema of the database
   Output = A list of features and for each table, the candidate feature.

   We create a model of the database and explore its tables to create groups of semantically related tables. We call these groups of semantically related tables "feature groups". To infer the semantic of a table, we rely on its name and the name of its columns. Once we have the list of features implemented by AppSI, we tag tables with the feature they belong to (each table should be tagged with exactly one feature).

2. *Validate feature identification with the database architect.*
   Input = A list of features and for each table, its corresponding feature.
   Output = A validation of the input and adjustment(s) to the input if needed.

   We provide the names of the features we identified to the AppSI architect. He can discuss the features we identified and eventually fine-tune them if needed.

3. *Infer the feature implemented by behavioral entities depending on the feature of the entity(ies) they use.*
   Input = A validated list of features and for each table, its corresponding feature.
   Output = For each behavioral entities, a candidate feature to tag it.

   We have the hypothesis that if a behavioral entity references entity(ies) from the same group of features, it probably belongs to this group. From that hypothesis, we designed a heuristic to categorize a behavioral entity in a feature group. Three cases can happen in the dependency relationships between a behavioral entity and the entities it uses.

(a) All the entities used by a behavioral entity belong to a single feature group, then the behavioral entity automatically belongs to this group (Behavioral entity 1 and Features group 1 on Figure 4.11).

(b) The entities used by a behavioral entity belongs to 2 feature groups and one of them is the core feature of the database. In that case, the entity belongs to the other group (the non-core feature group).

(c) The entities used by a behavioral entity belongs to multiple feature groups (other than the core feature). Behavioral entity 2 in figure 4.11 illustrates this case. In that case, the entity belongs to one of these groups but this needs to be decided by the architect.

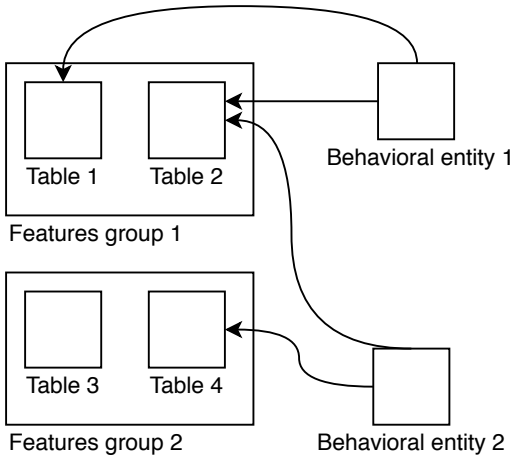

Figure 4.11: Two cases of dependency relationship between a behavioral entity and one or many group(s) of tables.

At the beginning of the process, the only entities that have a group are the tables that we categorized previously in step 1. During the process, the categorizations will spread to the other entities. At the end of the process, all entities will be in one of the following states:

(i) The entity has a feature group assigned.

(ii) The entity has multiple candidates feature groups and a decision needs to be taken.

(iii) The entity has no category because it references entities that need a decision to be taken.

(iv) The entity has no category because it references no entity defined by AppSI. This case can occur for utilities features (*e.g.,* a stored procedure implementing a string transformation).

4. *Validate the categorization of behavioral entities with the database architect and let him categorize those that could not be automatically categorized.*
   Input = For each behavioral entity, the candidate feature.
   Output = A validation of the input and adjustment(s) to the input if needed.

   Finally, we provide the list of entities categorized automatically to AppSI architect to evaluate the accuracy of our heuristic. If needed, the architect updates the group of some behavioral entities to fit his knowledge of the database. If such an update is performed, it means our approach made an error. Additionally, we provide the list of uncategorized entities to let the architect manually categorize these entities.

Figure 4.12 summarize the approach as an activity diagram. The result of our approach will be better documentation of the current state of AppSI. The groups of entities combined with the dependencies between entities provide a big picture of the database and allow one to understand more easily how the database is organized.

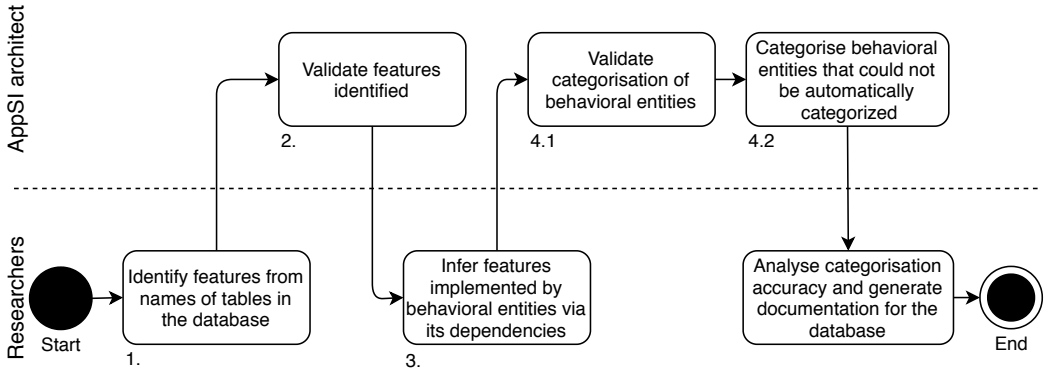

Figure 4.12: Activity diagram of the six-steps approach we use to document AppSI database.

Results Let us present our results for each step of our approach. We identified 11 feature groups (step 1 of the previous process) that were validated by the architect of AppSI (step 2):

- *Members management*: Deals with the management of members of the laboratory. Among its tables, it contains the central table of the database: `person` table. It stores data of people working at the laboratory such as name, address, email-address and so on. Because of that, this feature group can be considered as the core of AppSI.

- *Buildings management*: Deals with the management of buildings of the laboratory. It stores information such as where are members offices located and manages who has the keys of offices/buildings.
- *Mailing list*: Deals with mailing lists management.
- *PhD and HDR management*: Deals with PhD and HDR (a diploma one can get after the PhD in France) management. This include tracking PhD and HDR statuses of people concerned.
- *Public website*: Stores various data to show on the website of the laboratory.
- *Structure of laboratory*: Stores various data related to how the laboratory is organized from a human resources point of view. For example, it stores the different research team and people belonging to them, it stores the fact that some people are part of the administration, etc.
- *Access management*: Deals with access rights to buildings or zones for people working at the laboratory. For example, people working in a building A may not be authorized to access another building B. This group is separated from "Buildings management" because some access rights do not concern only buildings but also areas.
- *Profile*: The laboratory has an intranet proposing a directory that allows one to gather information on a person working at the laboratory. For example, one can search for the office of a person or get their phone number.
- *Scientific production*: Manages publications of researchers.
- *System tables*: Tables that contain data and meta-data required for some application. Example of such data are: URLs to useful endpoints, value for configuring some algorithms, etc.
- *Utilities*: A collection of stored procedures that perform generic tasks (*e.g.,* string manipulation) useful in various other feature groups. This feature group has the specificity that it was identified later in the experiment (step 4) as it does not contain any table.

Table 4.4 shows the number of tables per feature group after validation by the database architect (step 2). We can see that "Members management" and "Structure of laboratory" are the groups with the greatest number of tables in the database. It seems reasonable as managing members of the laboratory and managing the relationship between members is a central day to day task. The next feature group is "System tables", this observation is explained because of the policy of applications using AppSI to store their parameters directly inside the database and not in

configuration files. PhD and HDR management is the next one in terms of number of tables. Indeed, managing PhD and HDR is also one of the main tasks of the laboratory. The other feature groups have less tables.

Table 4.4: Number of tables for each feature group.

Feature group	# Tables
Members management	31
Structure of laboratory	24
System tables	16
PhD and HDR management	12
Buildings management	8
Public website	6
Mailing list	4
Scientific production	4
Profile	2
Access management	1
Utilities	0

Table 4.5 shows the results of the heuristic (step 3) and the evaluation of these results by the architect (step 4). 119 entities were automatically classified in a feature group by our heuristic and 83 were not. Out of the 119 entities automatically classified, only 3 were incorrectly classified.

Table 4.5: Results of the classification heuristic and architect validation.

	# View	# Stored Proc.	# Trigger	# Trigger Stored Proc.	Total
Automatically classified	45	26	31	17	119
Manually classified	30	33	0	20	83
Correctly classified	44	24	31	17	116
Incorrectly classified	1	2	0	0	3

Table 4.6 shows the final categorisations of views, stored procedures, triggers and trigger stored procedures of the database. We exported these data as a CSV file containing for each entity, its name and the category it belongs to. This CSV file together with feature groups descriptions form documentation for the database that is useful for future database architects. Furthermore, this dataset can be used

together with the heuristic to semi-automatically classify new entities that will be added in a future evolution of the database.

Table 4.6: Number of entities of a particular type in a particular category.

Feature group	# Tables	# View	# Stored Proc.	# Trigger	# Trigger Stored Proc.
Members management	31	24	30	13	18
Buildings management	8	0	1	0	0
Mailing list	4	1	0	0	0
PhD and HDR management	12	17	6	11	9
Public website	6	8	0	2	3
Structure of laboratory	24	24	10	4	7
Access management	1	0	0	1	0
System tables	16	0	1	0	0
Profile	2	0	0	0	0
Scientific production	4	1	0	0	0
Utilities	0	0	11	0	0

Conclusion Our approach to identify features implemented by AppSI and tagging entities of the database allowed us to reverse-engineer the database. The reverse-engineering of the database provides documentation that will be useful for future maintainers of AppSI.

Furthermore, we were able to do a dependency analysis for all kind of entities of the database. This shows that the meta-model addresses missing meta-data problems because stored procedure are handled by the approach while their body is not meta-described by the RDBMS.

4.5 Conclusion

In this chapter, we presented a meta-model for relational database taking into account both structural and behavioral entities and reifying references between entities. Then, we analyzed two different approaches to instantiate this meta-model: via meta-data extraction and by analyzing the dump of a database. We evaluated them according to three criteria: the completeness, the sensitivity to RDBMS evolution and the complexity to migrate to another RDBMS. This evaluation leads us to an hybrid approach to instantiate the meta-model which aims to satisfy as much criteria as possible. Finally, we presented two case studies illustrating that our meta-model addresses the problem “Relational databases are hard to understand” we identified in Chapter 2.

In the next chapter, we will show how instances of the meta-model can be used to evaluate the quality of a relational database.

CHAPTER 5

Identifying Quality Issues in Relational Databases

Contents

In this chapter, we show how querying an instance of the meta-model we presented in Chapter 4 can serve for detecting bad patterns in the schema of the database. Such a query can be seen as a rule that the database must fulfill to be considered of good quality. By checking the quality of relational databases, we adopt an approach that is a well known Software Engineering practice: code quality checking.

This section is an extension of our article "CodeCritics Applied to Database Schema: Challenges and First Results" published in the industrial track of the International Conference on Software Analysis, Evolution and Reengineering (SANER) [Delplanque 2017b].

5.1 Scenarios

In this chapter we present DBCritics, a tool to assess the quality of a relational database based on queries run on a model of the database. We identified three scenarios in which such a tool can be used on a DB schema.

5.1.1 Detecting Smells in Database

Database administrators need tools to find smells, anti-patterns and violations of business rules. The spaghetti query antipattern [Karwin 2010] aims to detect queries that are too complex to understand, maintain or debug. Some naming convention could need to be checked like prefixing all key columns names with `k_`. This first

example is a *generic* smell applicable to any database and even to any RDBMS. In contrast, *company-specific* or *database-specific* smells depend on their domain and their use. A DB code critics checking tool can provide a snapshot of the database schema quality, and could be used before and/or after each commit to detect possible deterioration of the database quality.

5.1.2 Migrating from a RDBMS Version to Another One

RDBMS are constantly evolving and new versions are regularly released to introduce new features or to fix bugs. In theory, the new version of a RDBMS should be backward compatible with older versions. In practice, it may happen that the behavior or name of an instruction has been changed from one version to the next. Upgrade migration patches are rarely provided due to the risk of breaking a database and all the applications that rely on it. In the case of open-source RDBMS like PostgreSQL, new releases only come with a textual change log. The task of database schema migration is left to the database administrator. One must identify what impact the changes will have on his schema, and correct them accordingly. For example, PostgreSQL documents that for versions after 9.0, PL/PgSQL variables will take preference over a table or view column with the same name[1]. If the behavior of unchanged code is modified after migration, this may lead to errors. In this case, detecting occurrences of variables with the same name that columns used in requests in the same stored procedure may prevent future errors.

5.1.3 Maintaining Consistency between Different Forks of a Database Schema

A database schema may be used as a basis for multiple software projects, each one adapting the schema to its needs. For example, AppSI database schema is shared with another laboratory, together with all the applications using the database. Each laboratory has its own database based on the initial schema. However, some modifications have been performed on the initial schema to adapt the database to the needs of new users. The laboratory that adopted the AppSI also benefits from maintenance to accommodate new features and/or bug fixes. Each change in the master database needs to be ported to the slave database with the risk that both databases continue to evolve separately, thus drifting further apart. Ensuring the naming convention between forks reduces maintenance efforts.

[1] https://wiki.postgresql.org/wiki/What's_new_in_PostgreSQL_9.0

5.2 DBCritics

This section presents our approach to analyze database schema quality based on a model of the database. It is implemented in a tool named DBCritics.

5.2.1 Overview

Figure 5.1 provides an overview of how DBCritics works. DBCritics takes as input a model of the database schema and a set of rules to check the model. It evaluates the rules on the model and generates a report that contains the result of each rule evaluation (violation or success).

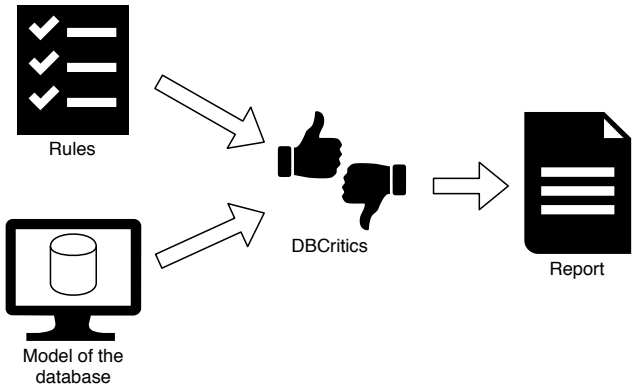

Figure 5.1: High-level view of DBCritics that evaluates rules on a model of the DB and provides a report.

5.2.2 Rules

DBCritics relies on the notion of *rule*. A rule describes a property the database schema should satisfy. Some rules are mandatory because they identify plain errors. Some rules are optional as they identify quality problems that may hamper future maintenance or evolution. A severity criterion allows to classify the rules, helping the user to concentrate his correction efforts. Three severity levels are provided by default: information, warning and error. Others may be added by the user if needed.

Each rule holds the list of entities violating it. This list can be reviewed by the user to mark false positives that are also stored to ignore future detections of these violations. A method specifying the entity types that the rule checks is also defined as well as a method specifying the violation constraints. The notion of rule has been extended to be able to define rules using a threshold. These rules can be parametrized by the users to fit their needs. For example, a `Table` is considered too big if it has $\geq n$ columns, where n is specified by the user.

5.2.3 Examples of Rules

Next to the severity criterion, other means exist to classify rules, such as the entity type on which the rule is applied. Some rules are generic (*i.e.,* applicable to any database) or specific (*i.e.,* only applicable to a particular database or RDBMS). Some rules concern the code, while others focus on the entity structures and the relationships between entities.

A first set of rules has been defined with a focus on diversity (*e.g.* focus on code, entity, relationship, or severity, ...). They have been defined by extending either `Rule` or `RuleWithThreshold` and overriding the methods presented in the previous section. These rules have been created based on a combination of the experience of database architects, related works and adaptation of existing rules for code smells detection.

- **Rule 1:** *Detect use of $\star$ in `SELECT` request.* Using $\star$ in the request selects all the columns of the used tables. The structure of the request result thus would change after column addition or removal in one of these tables. This can cause problems to the programs using it.

- **Rule 2:** *Foreign key referencing a non primary key*. The uniqueness of the reference is not guaranteed and leads to semantic errors.

- **Rule 3:** *Too many columns in SELECT request*. This rule identifies queries that may be complex to maintain (spaghetti query anti-pattern).

- **Rule 4:** *Table without primary key*. A table should always have a primary key.

- **Rule 5:** *Column not key (PK/FK) using the name convention for key (*e.g.*"k_" in name)*. If a naming convention exists, it is as important to use it for key columns as not to use it for non-key columns.

- **Rule 6:** *Stub entities* are used but not defined in the database schema. This rule detects for example the call of stored procedures not defined in the schema either intentionally if they correspond to system entity like `pg_class` table or `count()` function or involuntarily if the name of the function is misspelled or a removed function is still called.

- **Rule 7:** *Isolated table*. A table that is referenced by no entity and does not use any table cannot be accessed through natural join (based on foreign keys). It is certainly not used, or with other criteria that foreign keys which can lead to semantic errors.

- **Rule 8:** *Unused stored procedures* could be removed (if it is not used by an external program). This rule does not detect issues but may help in the cleaning of the schema.
- **Rule 9:** *Views using other views* is a poor design for the future evolution of the database. For example, if a view `vA` uses a view `vB` and `vB` needs to be modified, `vB` has to be deleted as well as `vA` and both views need to be recreated in an order satisfying usage dependencies between the views.
- **Rule 10:** *Views using only one table* might not be needed as a simple `SELECT` request can do the job.
- **Rule 11:** *Too many columns in a table* may be an illustration that the table has several concerns and should be decomposed.

Rule 5 is specific to our concrete example (the convention could be different depending on the database). The other rules are generic. Rule 3 and 11 have a threshold that must be defined by the user. Rule 4 represents an error because it means that the DB schema is not in the first normal form. Rule 6 can correspond to an error if the undefined entity is not a system one. All other presented rules correspond to warnings or information. Rules 1 and 3 focus on the code whereas all others focus on entities and their relationships.

5.2.4 False Positives

Sometimes, rules can be too strict: some issues can be acceptable in certain contexts. For example, referencing the `pg_class` while using inheritance between tables is normal. Yet, it will appear as a violation of rule 6. It may also happen that a database architect voluntarily leaves known smells in the schema because of lack of time or too high level of risk to fix them. For DBCritics, these bad smells can be tagged as false positives.

5.3 Case Studies

This section evaluates the usefulness of DBCritics on two case studies: the PostgreSQL version of the WikiMedia database [med 2016b] and AppSI. Table 5.1 summarises the sizes of these databases with the minimum and maximum number of entities of each kind over all analyzed versions.

WikiMedia is an open-source collaborative editing software project that runs Wikipedia. It currently has three versions of its database schema for three DBMS: MySQL/MariaDB, PostgreSQL and SQLite. We analyzed 25 different versions of

the PostgreSQL database schema representing the different versions available on Github [med 2016a] modified by 23 contributors.

We analyzed 12 consecutive versions of AppSI database schema.

Table 5.1: Min/Max number of entities per type and Min/Max lines of code for each database.

	Tables	Columns	Views	Functions	Triggers	LOC
WikiMedia	30/51	196/353	0/1	3/5	2/3	1,435/2,453
AppSI	71/91	583/974	30/52	46/67	12/16	4,910/7,006

DBCritics has been used to analyze the quality of the different versions of these two DBs. Three aspects were analyzed: the number of rule violations per version; the proportion of "violating entities" (*i.e.,* entities that violate at least one rule); and the "time-to-fix" of a rule violation (only for violations that actually got fixed). For this experiment, we applied the rules described in the previous section. WikiMedia had 67 violations on average over all considered versions, while AppSI had 76 violations on average.

5.3.1 Violation Count Per Version

Figure 5.2 shows the evolution of the number of violations (*i.e.,* unique pairs (entity, rule)) over time. We observe that the number of violations tends to increase (from 37 to 66 for WikiMedia and from 54 to 87 for AppSI). A possible explanation would be that contributors are unaware of these violations because of the lack of tools similar to DBCritics.

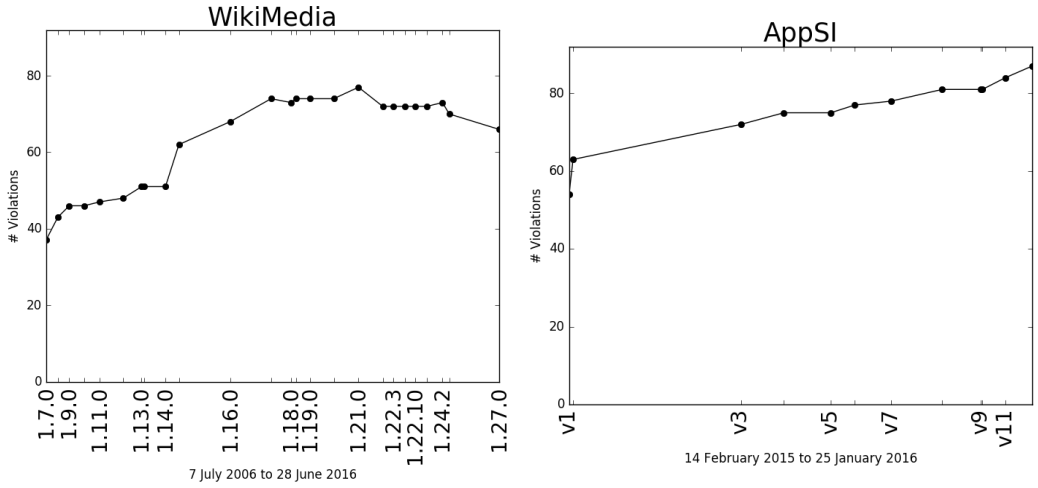

Figure 5.2: Violation count per version for WikiMedia and AppSI.

5.3.2 Violating Entities Proportion

Figure 5.3 shows the proportion of violating entities (grey cross) against the total number of entities (black dots) in each DB schema. On average, WikiMedia has more rule violations than AppSI (respectively 14% and 6%). The proportion of violating entities is globally stable over time (whereas the total number of entities is increasing).

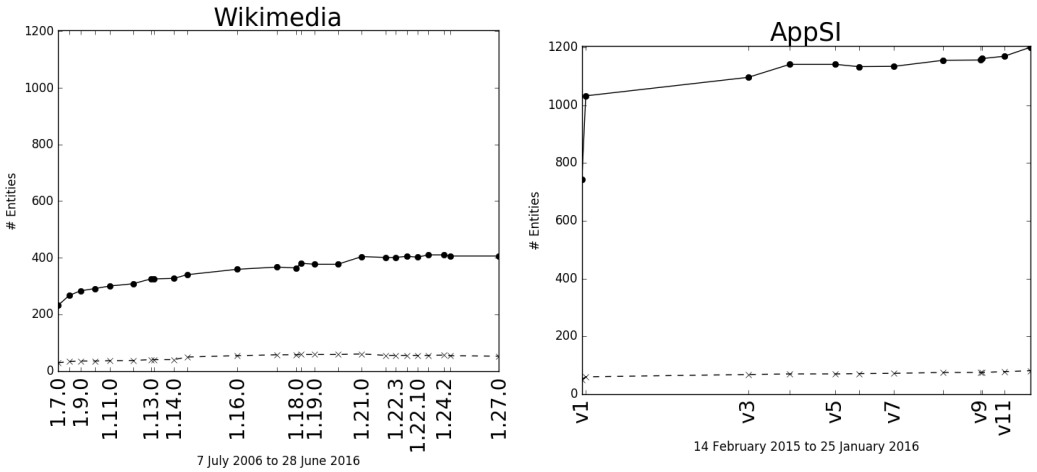

Figure 5.3: Violating entities (dashed) against entities count (solid) per version for WikiMedia and AppSI.

5.3.3 Time-to-fix of a Rule Violation

Only 3 of the 85 rule violations were corrected in AppSI on the 12 versions analyzed, and 21 of the 87 rule violations for WikiMedia. Table 5.2 summarises the "time-to-fix" in number of days for those rule violations that get resolved. The table reveals that corrections occur faster for AppSI than for WikiMedia. An interpretation would be that when the unique database architect of AppSI is aware of a violation, he corrects it very quickly because he knows his database very well. When several contributors work on the same database, knowledge is shared and more diffused. However, we can not make strong statements about this observation because we have not enough data.

These three analyses show that (1) there are rule violations in real life databases; (2) their number increases over time with the number of entities; and (3) only a few violations are fixed. A tool like DBCritics is thus needed to help in the violation correction. A deeper comparison between open-source and closed DB would be relevant. Moreover, it would be interesting to check whether the number of contributors really impacts the speed at which a rule is corrected.

Table 5.2: Minimum, first quantile, median, third quantile and maximum of the "time-to-fix" of resolved rule violations in days.

	Min	First quantile	Median	Third quantile	Max
WikiMedia	95	1227	1833	2403	3644
AppSI	3	/	125	/	278

5.3.4 Discussion About False Positives

Three categories of violations can be distinguished:

(i) real design issues that require modifications of the schema,

(ii) issues that the database architect considers correct despite the rule violation and;

(iii) issues due to limitations of DBCritics.

Concerning the second category, in AppSI, table `person` contains 26 columns and violates rule 11 for which the threshold is 25. However, this table contains 4 columns corresponding to computed values that are saved in the DB for performance reasons. Concerning the third category, at the time of the experiment, DBCritics was not able to manage views appearing in the schema as a table with an associated rule[2]; they are considered as tables by the tool although they are views. Some false violations are caused by these restrictions.

We have discussed with the database architect of AppSI to examine in detail the violations on one arbitrarily chosen version of its schema to get an idea of the proportion of each of these three categories. In version v10, there were 81 rule violations, 51 fell in the first category, 8 in the second and 22 in the last one. Therefore, the database architect judged that 63% of the detected violations did point to quality problems. Even if this proportion cannot be extrapolated to all versions of AppSI or any database, it gives a first idea. Deeper analysis should be done to generalize this result.

5.4 Conclusion

In this chapter, we adopt a software engineering approach by checking the quality of relational database schemas. Through 2 case studies, we investigated how querying a model of the database can be useful to find quality issues in a proprietary database (AppSI) and an open-source (WikiMedia) database.

[2] https://www.postgresql.org/docs/9.5/static/rules-views.html

As a summary of the analysis of AppSI and Wikimedia rules violation evolution, we observed that:

1. Rule violations can be found in open source as well as in proprietary database schemas.
2. The number of violations evolves with the number of entities.
3. With time, on both databases, some violations are fixed but not all of them.

In Chapter 6, we go further and we propose an approach to semi-automatically evolve a relational database based on the description of a change to apply on an entity of the database. Such an approach can be used to fix problems detected by DBCritics.

CHAPTER 6

Recommendations for Evolving Relational Databases

Contents

In Chapter 5, we show that querying the model of a database brings valuable information for database reverse-engineering. In this chapter, we go one step further in that direction by exploiting a model of a database to make it evolve via a semi-automatic process.

As explained in Chapter 2, relational databases are hard to evolve. This problem is caused by two main reasons: *some dependencies are implicit* and *no schema inconsistency is allowed at any moment*. Because of that, evaluating the impact of an evolution of the database schema is cumbersome. To address these problems, we present a semi-automatic approach based on recommendations that can be compiled into a SQL script handling implicit dependencies and fulfilling RDBMS constraints. Our approach is supported by the meta-model we presented in Chapter 4.

We performed an experiment to validate the approach by reproducing a real evolution on a database. The results of our experiment show that our approach can set the database in the same state as the one produced by the manual evolution in 75% less time. This chapter is an extension of our publication at the International Conference on Advanced Information Systems Engineering (CAiSE) [Delplanque 2020].

6.1 Setting the context

Before getting into the approach, let us set some definitions.

Impact of a change: Changing a database will probably affect its structure and/or its behavior. The impact of such a change is defined as the set of database entities that potentially need to be adapted for the change to be applied. For example, `RemoveColumn`'s impact includes constraints applied to the column.

Recommendation: Once the *impact of a change* has been computed, decisions might need to be taken to handle impacted entities. In the context of this thesis, we call each of these potential decisions a *recommendation*. For example, if one wants to remove a column, we recommend to remove the `NOT NULL` constraint concerning this column.

Note that Bohnert and Arnold definition of impact [Arnold 1996] mixes the set of impacted entities and the actions to be done to fix such entities: *"Identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change"*. To avoid confusion, we decided to use a specific term for each part of the definition.

Database schema consistency: The changes to apply on a relational database must comply with its constraints. We identified two kinds of constraints involved in a relational database:

(1) *data constraints* are responsible for data consistency. Five types of such constraints are available: "primary key", "foreign key", "unique", "not-null" and "check".

(2) *schema constraints* are responsible for schema consistency. These constraints ensure that the entities of the database fulfill the meta-model implemented by the RDBMS. Example of such constraints: a table can have only one primary key, a column can not have the same constraints applied twice on it, a foreign key can not reference a column that has no primary key or unique constraint, two entities of the same kind with the same name can not co-exist in the same namespace, etc...

The RDBMS ensures the consistency of the database schema. This notion of consistency is characterized by the fact that schema constraints are respected and no dangling reference is allowed (except in stored procedure).

Our approach works on a model of the database schema. Using a model allows one to temporarily relax schema constraints and dangling references constraints during evolution. It allows the developer to focus on changes to be made and not on how to fulfill schema consistency constraints and avoid dangling references at any time.

Operator: An operator represents a change to the database schema. It may impact several entities and require further changes to restore the schema in a consistent state after its application. The recommendations made by our approach take the form of operators.

More precisely, an operator is defined as follows:

- *A signature*: Defines the unique name and input parameters of an operator.
- *A description*: The description briefly explains the purpose of the operator.
- *One or multiple categories of entities*: A change may impact entities (instances of the meta-model concepts) of different kinds (*e.g.,* column, table, stored procedure, column reference, table reference, ...) and each kind may be handled differently. Moreover, within a given kind, entities with different properties may also be handled differently. For example, columns may carry different constraints (foreign key, *default*, or *not null*) requiring different treatments. Each entity kind and property subgroup defines a *category of entities*. Each *category of entities* has an identifier C_n.
- *One or multiple recommendations for each category*: For each C_n, at least one recommendation should be provided. The recommendation consists in the application of one or multiple operator(s) on entities of C_n and a textual description of its meaning in natural language. This description provides some context to justify why an operator is needed. If multiple recommendations are provided, the database architect needs to choose the operator fitting their needs.

In Appendix A, we provide the list of 11 operators we defined.

Reference-oriented operator: A reference-oriented operator responds to the previous definition. In addition, it applies on an element of the model representing a reference. RDBMSs do not reify references. Thus, such concepts are implicit and only exist in the source code of database entities. Because of that, they can not be directly translated as SQL queries. On the other hand, our meta-model reifies references. Consequently, our approach converts reference-oriented operators into entity-oriented operators. The details of this conversion are explained later in the chapter (Sub-section 6.2.3). An example of reference-oriented operator is `ChangeReferenceTarget` (see Sub-section A.1.6).

Entity-oriented operator: An entity-oriented also responds to the previous definition but applies on an element of the model that does not represent a reference. This kind of operator has the particularity to be translatable directly into one or

many SQL queries. An example of entity-oriented operator is `RenameColumn` (see Sub-section A.1.5).

6.2 Description of the Approach

To evolve a database, the database architect specifies operators on some of its entities. These operators impact other entities that, in turn, need to evolve to maintain the database in a consistent state. To handle evolutions induced by the initial operators, we developed a 3-step approach (see Figure 6.1). The implementation of this approach is available on github[1].

For each operator expressed by the architect:

A. *Impact computation*: The set of impacted entities is computed from the operator. The next step treats impacted entities one by one.

B. *Recommendations selection*: Depending on the operator, and one impacted entity, our approach computes a set of recommendations. These recommendations are presented to the database architect that chooses one when several are proposed. This introduces new operators that will have new impacts. Steps A. and B. are recursively applied until all the impacts have been managed.

C. *Compiling operators as a valid SQL patch*: All operators (original one plus the recommendations chosen by the architect) are converted into a set of SQL queries that can be run by the RDBMS. The set of SQL queries is used to migrate the database to a state in which the initial architect's operators have been applied.

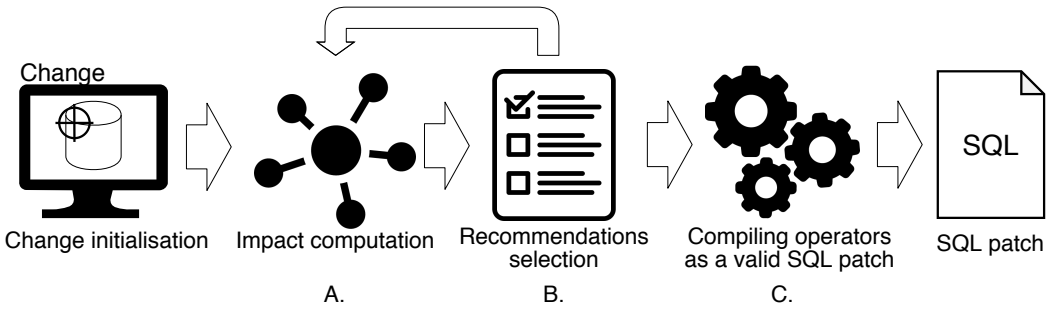

Figure 6.1: Coarse-grain illustration of the approach.

We now detail each step.

[1] https://github.com/juliendelplanque/DBEvolution

6.2.1 Impact computation

To compute the entities potentially affected by an operator, one needs to collect all the entities referencing the entity targetted by the operator. For example, if a Column is subject to modification, our approach identifies the impacted entities by gathering all the ColumnReferences targeting this column. The impact of the operator corresponds to the sources of all ColumnReferences since they can potentially be affected by the modification.

6.2.2 Recommendations selection

For each operator, the set of impacted entities is split into disjoint sub-sets called *categories*. For each of these categories, one or several recommendations are available. We determinate those recommendations by studying how to make evolve the impacted entities while respecting the database schema constraints.

The output of this step is a tree of reference-oriented operators where the root is the operator initiated by the architect and, each other node corresponds to an operator chosen among recommendations.

6.2.3 Compiling operators into a SQL patch

Once all the recommendations are chosen, our approach generates a SQL patch. This patch contains queries in SQL data definition language (DDL). For this purpose, the tree of operators resulting from the previous step has to be transformed into a sequence of SQL queries. These queries enable migrating the database from its original state to a state where the initial operator and all induced operators have been applied.

We stress that, during the execution of any operator of the patch, *the RDBMS cannot be in an inconsistent state*. This constraint is fundamentally different from source code refactoring where the state of the program can be temporarily inconsistent. Therefore, each operator must lead the database to a state complying with schema consistency constraints. Else the RDBMS will forbid the execution of the SQL patch.

Converting reference-oriented operators into entity-oriented operators The tree resulting from the step described in Section 6.2.2 is composed of operators on references. However, DDL queries only deal with entities. Thus, reference-oriented operators are transformed into entity-oriented operators. This is performed in two steps:

1. All reference-oriented operators are grouped according to their source entity, *i.e.,* the entity to which belongs the source code in which the reference ap-

pears. This step enables the identification of the entity on which the entity-oriented operator is applied. It has to be noticed that several referenced-oriented operators may be convert in a single entity-oriented operator in order to satisfy schema constraints.

2. For each entity identified in the previous step, we create an operator modifying its source code. To do so, we iterate on the list of reference-oriented operators. For each reference, we update the part of the source code corresponding to the reference to reflect the change implemented by the operator.

Once the iteration is complete, a new version of the source code has been built with no more dangling reference.

Converting entity-oriented operators into DDL statements As the RDBMS does not allow schema inconsistencies at any moment, even during DDL statements execution, DDL statements need to be ordered to fulfill this constraint. Solving this issue implies two steps:

1. the sourced entity affected by an operator modifying its source code needs to be deleted and created again.

2. entities depending on this sourced entity that are not part of the architect's decisions also have to be deleted before and (re-)created again exactly as they were after the deletion/creation of the sourced entity.

Thus, these entities are deleted in an order that satisfies dependencies between entities of the database.

Finding the right order of deletion is important because otherwise the RDBMS will fail while interpreting the SQL script. To find this order, we create a graph containing entities of the database concerned by the operator as vertices and dependencies between these entities as edges. Then, we apply a topological sort algorithm on this graph.

The re-creation of entities is made in the opposite order of deletions. The newly created entities are potentially different from deleted ones because operators applied by the architect were taken into account. The operator initiated by the architect occurs in the middle of these deletions and re-creations.

6.2.4 Example

To explain the proposed process, let us take a small example. Consider the simple database model shown in Figure 6.2. In this database, there are two tables, `t1` with two columns `t1.b`, `t1.c` and `t2` with column `t2.e`. Additionally, one stored procedure

s() and three views v1, v2 and v3 are present. On this figure, dependencies between entities are represented by arrows. These dependencies arrows are a generalization over the various kinds of reference entities of the meta-model. For example, the arrow between s() and t1 is an instance of TableReference and the arrow between s() and b is an instance of ColumnReference. Views and stored procedures have source code displayed inside their box. In this source code, any reference to another entity of the database is underlined.

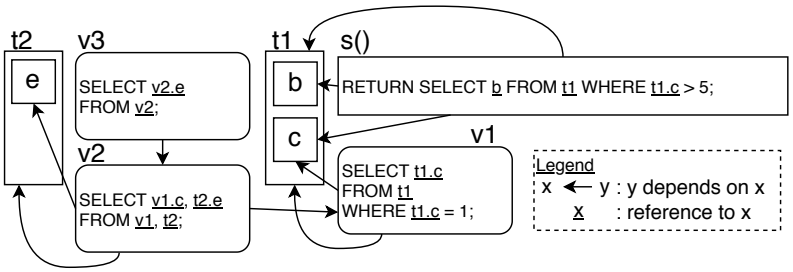

Figure 6.2: Example database.

The architect wants to rename the column c of table t1 into d.

Impact computation. First, we compute the impact of this operator. Column c of table t1 is referenced three times. Thus, the three entities below that make a reference to the column are part of the impact set of the operator:

- The WHERE clause of the SELECT query of the stored procedure s().
- The SELECT clause of the query defining view v1.
- The WHERE clause of the query defining view v1.

Recommendations selection. For each of the three impacted entities, recommendations are produced. For the WHERE clause of the stored procedure s(), the recommendation is to replace the reference to column t1.c with a new one t1.d. The result of replacing this reference will be the following source code: RETURN SELECT b FROM t1 WHERE t1.d $>$ 5;. The process continues recursively on this operator, the impact is computed but is empty.

The recommendation concerning the WHERE clause of v1 is the same: replacing the reference to t1.c by a reference to t1.d. Again, there is no further impact for this operator.

For the reference to t1.c in the SELECT clause of view v1, two recommendations are proposed to the architect: replacing the reference with aliasing (*i.e.,* replacing SELECT t1.c by SELECT t1.d AS c) or without aliasing (*i.e.,* replacing SELECT t1.c

by SELECT t1.d). In the latter case, the column c in view v1 becomes d; it is no longer possible to refer to v1.c. Consequently, the second recommendation leads to renaming column v1.c. If the architect chooses this option, the recursive process continues: new impacts need to be computed and new operators to be performed. The SELECT clause of view v2 is impacted. Two recommendations are again provided: replacing the reference with or without aliasing. In this case, the architect chooses to alias the column and replace the reference. Thus, the rest of the database can continue to refer to column c of view v2. Figure 6.3 illustrates this step.

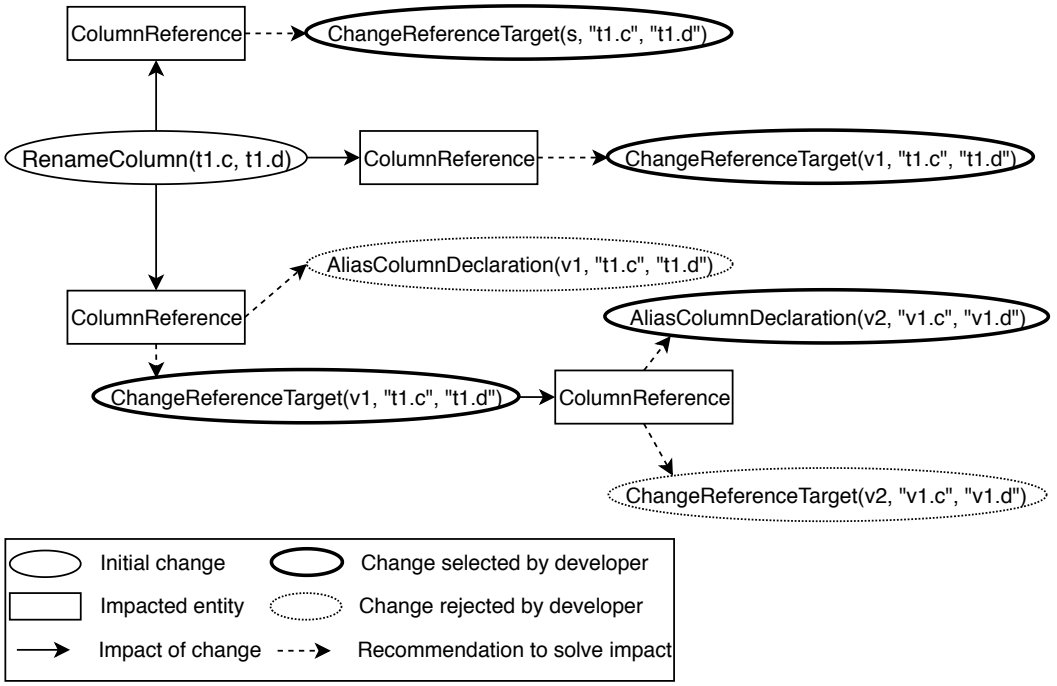

Figure 6.3: Recommendations selection.

Convert reference-oriented operators as entity-oriented operators Figure 6.4 illustrates the step of conversion of reference-oriented operators resulting from the recommendations into entity-oriented operators. For this purpose, operators concerning the same sourced entity are aggregated. Operators (3) and (4) concern the same sourced entity, v1. They are thus aggregated into ModifyViewQuery(v1). At the end, there is a single operator per entity to be modified.

Convert entity-oriented operators into DDL statements The resulting list of entity-oriented operators needs to be converted to DDL statements that the RDBMS can evaluate and ordered to fulfill schema consistency constraints. Figure 6.5 illustrates this step of the approach. For example, ModifyStoredProcedureBody(s) is converted into DeleteStoredProcedure(s) that remove the current version of the stored

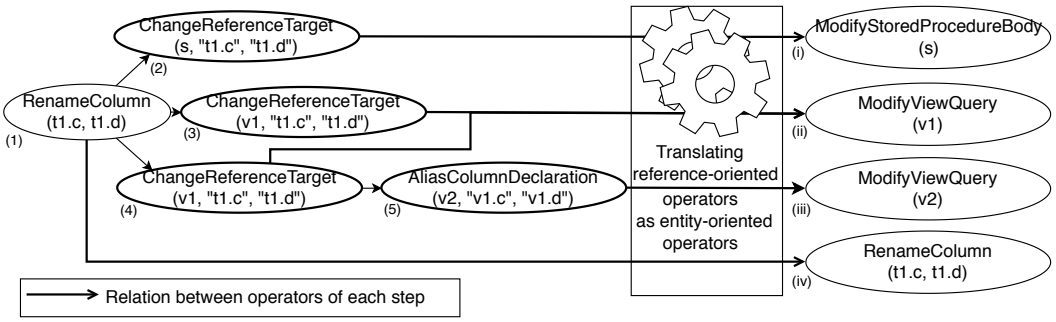

Figure 6.4: Convert reference oriented operators as entity-oriented operators.

procedure from the database and CreateStoredProcedure(s) that re-create a version of the stored procedure with its source code updated. As reminder, the new source code of the stored procedure makes a reference to t1.d instead of t1.c because column c of table t1 is renamed to d.

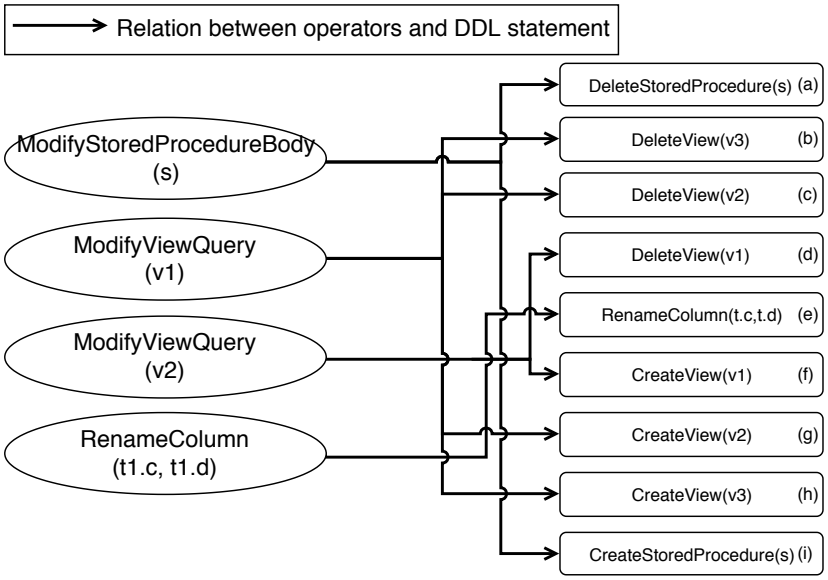

Figure 6.5: Convert entity-oriented operators as a list of DDL statements that fulfill RDBMS schema consistency.

One can observe that the view v3 is deleted and recreated while no entity-oriented operator applies to it. The occurences of DeleteView(v3) and CreateView(v3) are induced by the schema consistency constraint. Indeed, v3 references v2 and no dangling reference is authorized at any moment during the execution of the SQL patch. Thus, v3 needs to be removed to be able to remove v2 and then recreated after v2 has been recreated with an updated version of its source code. Nevertheless, v3 source is not altered during this process.

Figure 6.6 illustrates the graph formed by t1.c and all the depending sourced entities gathered recursively. Entities that are not subject to an entity-oriented operator are gathered as well to build this graph which explains why v3 gets deleted and recreated in the process. A topological sort algorithm is applied on this graph to find the proper order of entities deletion. A possible order is s(), v3, v2, v1, c. This order can be used for ordering the deletion DDL statements. To recreate entities with potentially a new version of their source code, one simply need to inverse the order of deletion. In our case, it leads to the following order: c, v1, v2, v3, s().

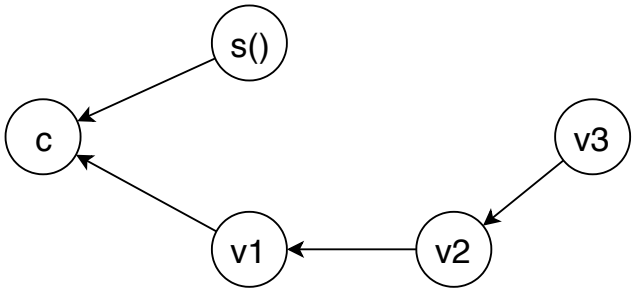

Figure 6.6: Graph on which topological sort is applied to find the order of DDL statements ensuring schema consistency.

6.3 Experiment

We performed an experiment on AppSI database introduced in Chapter 2. Before each migration of AppSI the database architect prepares a road map in natural language containing the list of operators initially planned for the migration. We observed that this road map is not complete or accurate [Delplanque 2018]. Following a long manual process, the architect writes a SQL patch to migrate from one version of the database to the next one.

The architect gave us access to the SQL patch used to perform the evolution to do a post-mortem analysis of the database evolutions. One of the patches implements the renaming of a column belonging to a table that is central to the database. This is interesting because it is a non-trivial evolution.

We had the opportunity to record the architect's screen during this migration [Delplanque 2018]. We observed that the architect used multiple tools to perform a trial-and-error process to find dependencies between entities of the database. He implements part of the patch and runs it in a transaction that is always rolled back. When the patch fails during its execution, the architect uses the gained knowledge to correct the SQL patch. Using this methodology, the architect built incrementally the SQL patch during approximately 1 hour. The patch is $\sim$ 200 LOC and is composed of 19 SQL statements. To validate our approach, we regenerate this

SQL patch with our tool but without the architect's expertise. Then, we compare our resulting database with the one obtained by the architect.

6.3.1 Experimental Protocol

The goals of the experiment are multiple:

(i) Illustrate, on a concrete case, the generation of a SQL patch.

(ii) Compare the database resulting from our approach with the database resulting from the patch originally written by the architect.

(iii) Estimate the time required to generate a SQL patch as compared to the manual generation.

Based on the road map and the comments in the patch we extracted the operators initiated by the architect during this migration. A discussion with the architect allowed us to validate the list of initial operators:

1. RenameColumn(person.uid, login)
2. RemoveFunction(key_for_uid(varchar))
3. RemoveFunction(is_responsible_of(int4)),
4. RemoveFunction(is_responsible_of(int4,int4))
5. RenameFunction(uid(integer), login(integer))
6. RenameLocalVariable(login.uidperson, login.loginperson)
7. RemoveView(test_member_view)

Details on these operators can be found in Appendix A.

We implemented a graphical user interface for our approach. Figure 6.7 shows a screen capture of DBEvolution graphical user interface. It guides the user through the steps of choosing recommendations.

Panel 1 shows the list of operators selected by the user and the tree of impacts resulting from the user's choices. When an operator is inserted or clicked in panel 1, panel 2 shows the entities potentially impacted by the operator. The UI allows one to unfold the set of impacted entities to show one or many recommendations the user needs to make on them. The "gear and spanner" icon means that the user still needs to choose a recommendation. The green check icon means that the user already chose a recommendation. When one of the entities in panel 2 is clicked, two things happen: first, panel 3 shows the different recommendations the user can

choose and second, panel 4 shows the source code of the entity that is concerned (if it holds source code) with the reference to the entity that created the impact highlighted. In panel 3, the "Use this operator" button allows one to choose a recommendation.

Once all the choices were made (all operators in panel 1 have the green check icon), one can click the "Generate patch" button (top left of the tool) to generate the SQL script.

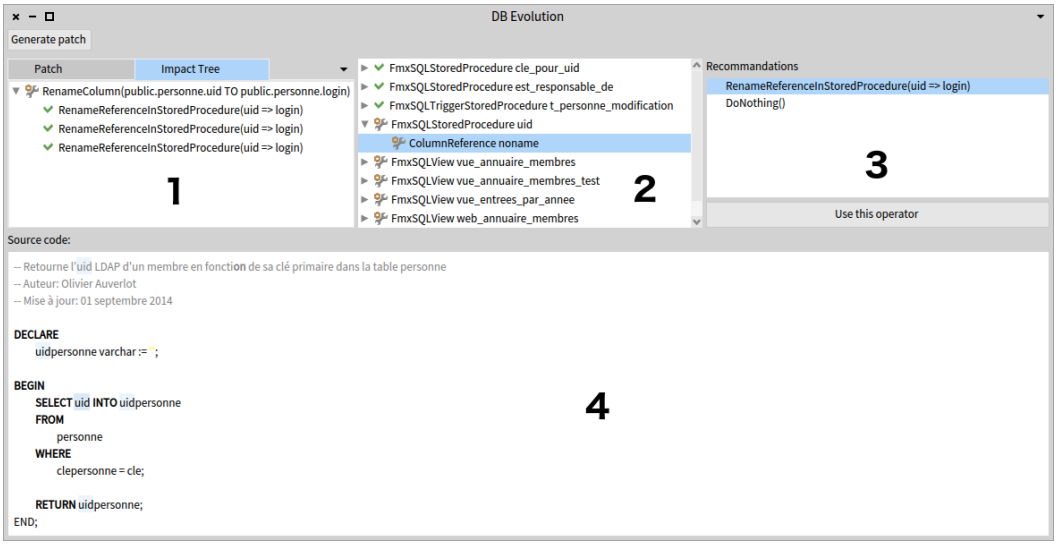

Figure 6.7: Screenshot of DBEvolution, the implementation of our approach that we used to perform the experiment.

The experiment consists in inserting the list of operators extracted from the roadmap in our tool and following the recommendations proposed. Potentially several recommendations might be proposed, particularly as whether to create aliases in some referencing queries or to rename various columns in cascade. The architect told us that, as a rule, he preferred to avoid using aliases and renamed the columns. These were the only decision we had to do during the experiment.

We finished the experiment by executing the SQL patch generated by our tool on an empty (no data) copy of the database. Note that having no data in the database to test the patch might be a problem for operators modifying data (*e.g.,* changing the type of a column implies converting data to the new type). However, in the case of our experiment no operator modifies data stored in the database. First, we checked whether the generated patch ran without errors. Second, we compared the state of the database after the architect's migration and ours. For this, we generated a dump of the SQL schema of both databases and compared these two dumps using a textual diff tool. Third, we also considered the time we spent on our migration and the one used by the architect when he did his.

6.3.2 Results

We entered the seven operators listed previously in our tool and let it guide us through the decision process to generate the SQL migration patch.

Fifteen decisions were taken to choose among the proposed recommendations. They all concerned the renaming or aliasing of column references. From this process, the tool generated a SQL patch of ~ 270 LOC and 27 SQL statements.

To answer the goals of the experiment listed previously:

(i) The generated SQL patch was successfully applied on the database.

(ii) The diff of the two databases (one being the result of the hand-written patch and the other being the result of the application of the generated patch) showed a single difference: a comment in one stored procedure is modified in the hand-written version. Such changes are not taken into account by our approach.

(iii) Encoding the list of operators and taking decisions took approximately 15 minutes. This corresponds to about 25% of the time necessary to the architect who has a very good knowledge of his database to obtain the same result.

6.4 Conclusion

In this chapter, we presented an approach to manage relational databases evolution. This approach allows one to specify an operator to apply on an entity of a model (instance of the meta-model presented in Chapter 4) and to guide the database architect in the decisions one needs to take to apply the initial operator. The decisions taken by the architect together with the initial operator are then converted into a SQL script that can be executed on the database to make it evolve. We set up an experiment to assess that our approach can reproduce a change that happened on a database used by a real project. The experiment shows that our approach can reproduce the change but allows a faster implementation with a gain of 75% of the time. These results show that our approach address "database are hard to evolve" problem and more specifically, it addresses the implicit dependencies and the schema consistency issues.

CHAPTER 7

Conclusion

Contents

7.1 Summary

Relational databases have been at the core of many information systems for decades. Many of these databases reflect human or societal activities. For example, processes related to human resources, insurances, banks, etc. Thus, these relational databases evolve regularly and tools are needed to support this evolution.

The purpose of relational databases is not only to store data. Indeed, it is possible to have behavioral entities taking the form of views, stored procedures, triggers, etc. These behavioral entities, while allowing one to implement complex computations on the database side, make the understanding and evolution of relational databases challenging.

In this thesis, we addressed relational database evolution from a software engineering perspective. More specifically, we addressed the two following problems that we observed during an evolution performed by a database architect (see Chapter 2):

1. *Relational databases are hard to understand*: This problem has two main causes: RDBMS meta-data are complex to query and, for some types of entities in the database, meta-data are missing.

2. *Relational databases are hard to evolve*: This problem comes from the fact that: the dependencies between some kinds of entities are implicit and the database schema can not be in an inconsistent state at any moment.

To address these problems, we developed a behavior-aware meta-model that helps to understand the database. We exploited this meta-model to find quality

issues and developed an approach to evolve the database while addressing implicit dependencies and schema consistency problems.

Below, we summarize the work we achieved in each chapter.

Chapter 2 reports our observations of the evolution of AppSI, the relational database used by the information system of our laboratory. We did a qualitative and quantitative analysis of screen records of AppSI database architect performing an evolution. Our observations led us to the identification of the two problems addressed in this thesis: *relational databases are hard to understand* and *relational databases are hard to evolve*.

Chapter 3 reviews the scientific literature to assess the state of the art related to problems we identified in Chapter 2. In particular, we review approaches related to relational database reverse engineering and relational database impact analysis. From the literature review, we concluded that approaches for relational database reverse engineering usually do not consider behavioral entities. Furthermore, impact analysis approaches focus on computing the impact of a change made to a structural entity of a database on external programs interacting with the database.

Chapter 4 presents a meta-model representing structural entities, behavioral entities and relations between them. Then, we compared two approaches to instantiate the meta-model via a set of criteria and designed an hybrid approach to satisfy as much criteria as possible. Finally, we presented two case studies illustrating that the meta-model addresses the problem *relational databases are hard to understand*.

Chapter 5 presents a tool exploiting the meta-model to detect quality issues in the schema of a relational database. This tool was used to analyze quality issues in a proprietary database (AppSI) and an open-source database (WikiMedia). This study reveals that rule violations can be found in open source as well as in proprietary database schemas, the number of violations evolves with the number of entities and with time some violations are fixed but not all of them.

Chapter 6 present a semi-automatic approach to evolve a relational database. The approach is based on recommendations that can be compiled into a SQL script handling implicit dependencies and fulfilling RDBMS constraints. We set up an experiment to assess that our approach can reproduce a change that happened on a database used by AppSI. The experiment shows that our approach can reproduce the change but allows a faster implementation with a gain of 75% of the time. These results suggest that our approach addresses the problem *relational databases are*

hard to evolve and more specifically, it addresses the implicit dependencies and the schema consistency issues.

7.2 Contributions

The main contributions of this thesis are:

- The identification of concrete problems encountered during relational database evolution via the observation of a developer.
- A meta-model designed to help in dependencies analysis related to entities defined inside a database.
- A tool to find quality issues in a database schema and the study of quality issues evolution on 2 case studies.
- A semi-automatic approach based on recommendations that can be compiled into a SQL script fulfilling RDBMS constraints.

7.3 Future Work

In this section, we present open issues that are not addressed in the thesis. These open issues provide opportunities to continue our research concerning the behavior-aware meta-model and the semi-automatic approach to evolve relational databases.

Abstract Syntax Tree Reification In this thesis, we developed a meta-model that models structural entities, behavioral entities and references between these entities (Chapter 2). In this meta-model, the concept of "reference" is used to represent the fact that an entity references another entity.

In its current state, the meta-model reifies queries and their clauses. Each clause contain a collection of references allowing one to know which clauses are referencing an entity in a model. For example, the model can be queried to retrieve the clauses that are referencing the column of a table.

However, the meta-model does not represent the full query abstract syntax tree. For example, in the following `SELECT` query, in the `WHERE` clause the boolean expression is not represented in the model. Instead, the `WHERE` clause in the model stores two table references: pointing respectively to `t1` and `t2` and stores two column references: `c3` and `c4`.

```
1 CREATE VIEW v AS
2     SELECT t1.c1, t2.c2
```

```
3     FROM t1, t2
4     WHERE t1.c3 = t2.c4 ;
```

This representation of references in the meta-model allows one to perform various analyses as illustrated in Chapter 4, Chapter 5 and Chapter 6. However, to provide more possibilities to evolve a database with the approach presented in Chapter 6, a finer-grain representation of queries and source code is needed. For example, with the current representation of queries, it is not possible to transform a comparison operator <= into <. Such kind of changes could be needed for some operators. One can address this problem by representing the abstract syntax tree of stored procedures and CRUD queries in the meta-model.

With an AST representation of the previous query, its `WHERE` clause would not be a simple collection of references but a tree with as root the equality operator (=) and as leaves the references to the tables and columns.

This enhancement of the meta-model would allow one to perform more detailed dependency analysis and provide recommendations for our approach presented in Chapter 6.

Support Data Transformation Some operators might require to transform or move data stored in the database. A simple example would be an operator that moves a column from a table to another table. For example, if we have a database storing blogposts for a blog with the following schema:

```
1 CREATE TABLE blog_post (
2     id INTEGER PRIMARY KEY,
3     title TEXT,
4     content TEXT
5 );
```

One can implement an operator modifying the schema of the database in order to historize the values of a column. In the example, above, let us historize the values of `content` column in `blog_post` table. One way to historize these values is to create a second table that stores the content of a blog post. This new table named `blog_post_history` has a column `blog_post_id` that references the blogpost for which the `content` is stored and has a `timestamp` column that stores the time of modification of the content. With such schema, one can get the content of a blogpost by selecting the related row in `blog_post_content` with the greater timestamp. The schema of the database after the historization is shown below.

```
1 CREATE TABLE blog_post (
2     id INTEGER PRIMARY KEY,
3     title TEXT
```

```
);

CREATE TABLE blog_post_content (
    id INTEGER PRIMARY KEY,
    content TEXT,
    timestamp TIME,
    blog_post_id INTEGER REFERENCES blog_post(id)
);
```

To implement this historization operator, one need to move the column `content` from `blog_post` to `blog_post_content`. Additionally, the current data in the column `content` of `blog_post` table need to be moved.

This new constraint raises the following challenge for the approaches we presented in Chapter 6: *How to to handle data?*

We designed the approach in a way that, in the previous example, it would first remove `content` column of `blog_post` table and then create `content` column in `blog_post_content` table. To implement the historization operator, the approach needs to let the two columns co-exist to be able to move the data between these two columns. Thus, the algorithm that orders changes to apply on the database needs to be adapted to fulfill this requirement.

More generally, other operators could require other kinds of transformation on data. Further work on our approach to evolve database is required to support data transformation.

Simulate Operators on the Model In its current state, the approach presented in Chapter 6 allows one to specify a single operator at a time. For example, one can ask the approach to provide recommendations for the renaming of a column but one can not ask the approach to provide recommendations for two operators that would rename a column and then rename the table that contains this column. This limitation is due to the fact that once all recommendations have been selected by the user, an analysis of the operators and the model is performed and a patch to migrate the database is generated. In this process, the model of the database is not updated.

Figure 7.1 illustrates how the current approach applies a sequence of 2 operators on the database. Basically, one needs to handle the operators of the sequence one by one. On the figure, a model of the database in version 0 is created, then DBEvolution is used to compute recommendations for operator 1 which leads to the generation of patch 1. Then, the patch is applied on the database with version 0. The database evolves to version 1 and the process repeats for operator 1.

The current way to treat a sequence of operators works but the approach does not have a global view on the effect of the sequence of operators on the database.

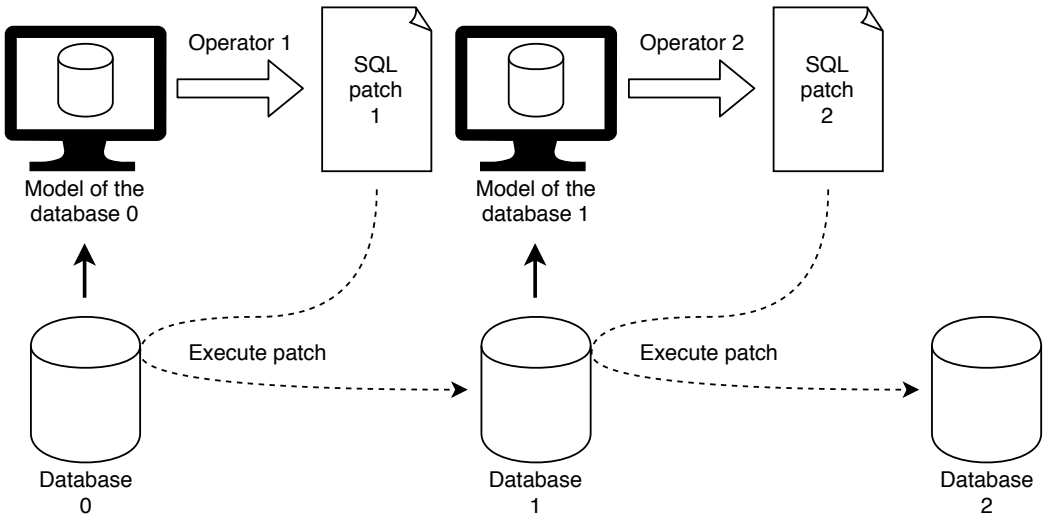

Figure 7.1: Current strategy to apply a sequence of operators.

It considers the local effect of each operator. This behavior prevents us to optimize the patch. For example, if an operator requires a view to be deleted and recreated but this view gets deleted by the next operator in the sequence, we could avoid recreating the view as it is not needed.

If one wants to generate a patch containing a sequence of operators and their induced operators without having to modify the database operator per operator, we need to simulate the execution of each operator on the model as illustrated by Figure 7.2. On that figure, a model of the database in version 0 is created. Then, operators are applied directly on the model and only once all the operators of the sequence have been applied, the SQL patch is generated to evolve the database.

For that purpose, more work is required on the approach to let it simulate the effect of an operator directly on the model. This future work is challenging because one needs to make sure that the simulated operators transform the model into a state consistent with the state of the database when executing the patch. For example, the model of the database 1 in Figure 7.2 must be exactly the same as the model of the database 1 in Figure 7.1 that was built by executing the patch and recreating the model. If the simulation of operators on the model creates corrupted models, our semi-automatic approach to evolve database might generates wrong recommendations.

Conflict Management This future work occurs once it is possible to simulate operators on a model. If one provides a sequence of operators to apply on the database, it is possible that some of them are conflicting. For example, if one creates the following sequence of operators:

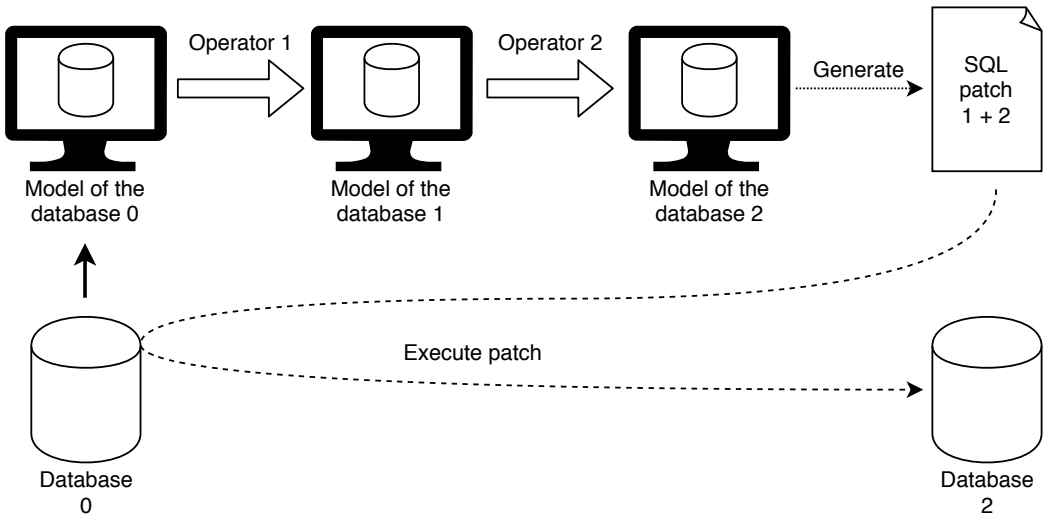

Figure 7.2: Desired strategy to apply a sequence of operators.

1. `DeleteColumn(t.c)`
2. `RenameColumn(t.c)`

Obviously, these operators are conflicting because the first one will remove the target entity of the second one.

The goal of this future work is to explore what are the combination of simple operators (Add, Remove, Rename, Move) that are conflicting. This information could be part of the description of these simple operators.

In a second step, the goal would be to determinate if it is possible to decide if two complex operators (operators defined in terms of a sequence of simple or complex operators) are conflicting or not. The challenge is to design an algorithm that relies on the conflict description of operators that define the two complex operators provided as input.

APPENDIX A

Operators Catalog

Contents

This section provides additional information related to operators used by DBEvolution in Chapter 6 experiment.

A.1 Catalog

This section provides specifications only for operators used in the context of the experiment described in the paper.

A.1.1 Rename Local Variable

RenameLocalVariable renames a local variable of a stored procedure.
Signature: RenameLocalVariable(localVariable, newName)
Description: Renames a local variable of a stored procedure.
Impact: Let I be the set of entities impacted by the change;

- $C_1(I) = I$ Deals with references to the local variable from inside the stored procedure holding it. This category is the only one available for this operator. Thus, it is equal to the impact set I.

Recommendations:

- $C_1 \rightarrow$ ChangeReferenceTarget: Since the local variable is renamed, the references need to be adapted to target the reference to the local variable via its new name.

A.1.2 Rename Function

RenameFunction renames a stored procedure in the database. This stored procedure might be called by behavioral entities in the database.
Signature: RenameFunction(function, newName)
Description: Renames a stored procedure in the database.
Impact: Let I be the set of entities impacted by the change;

- $C_1(I) = \{e \in I | has\_type(source(e), SelectClause)\}$ Deals with references to the stored procedure from select clause in SELECT query.
- $C_2(I) = \{e \in I | \neg has\_type(source(e), SelectClause)\}$ Deals with other references to the stored procedure.

Recommendations:

- $C_1 \rightarrow$ Choice(ChangeReferenceTargetInSelectClause, AliasColumnDeclaration): If a stored procedure is called from a select clause and not aliasing is specified, the name of the associated column is the name of the stored procedure. Thus, the architect needs to choose between changing call site and aliasing the column with the old stored procedure name to stop the impact propagation or only changing call site and thus propagating further the impact.
- $C_2 \rightarrow$ ChangeReferenceTarget: Calls to the stored procedure in other contexts need to be adapted to use the new name of the procedure. This recommendation is similar to what is done when renaming a function in procedural programming languages.

A.1.3 Remove Function

RemoveFunction removes a stored procedure from the database. This stored procedure might be used by behavioral entities of the database.
Signature: RemoveFunction(function)
Description: Removes a stored procedure from the database.
Impact: Let I be the set of entities impacted by the change;

- $C_1(I) = I$ Deals with calls to the stored procedure.

Recommendations:

- $C_1 \rightarrow$ HumanIntervention: The removal of the method induces complex changes in the source code which need to be evaluated and performed locally by the architect on each entity that was calling the stored procedure. To assist the architect in this task, DBEvolution highlights these call sites.

A.1.4 Remove View

RemoveView removes a view from the database. This view might be used by behavioral entities of the database.
Signature: RemoveView(view)
Description: Removes a view from the database.
Impact: Let I be the set of entities impacted by the change;

- $C_1(I) = I$ Deals with all references to the view.

Recommendations:

- $C_1 \rightarrow$ HumanIntervention: The removal of the view induces complex changes in the source code which need to be evaluated and performed locally by the architect on each entity that was referencing the view.

A.1.5 Rename Column

RenameColumn renames a column in a table of the database. This column might be referenced by other entities in the system.
Signature: RenameColumn(view)
Description: Renames a column in a table of the database
Impact: Let I be the set of entities impacted by the change;

- $C_1(I) = \{e \in I | is\_wildcard\_reference(e)\}$ Deals with references to the column through a wildcard ($\star$).
- $C_2(I) = \{e \in I | has\_type(source(e), SelectClause)\}$ Deals with references to the column from a select clause in a SELECT query.
- $C_3(I) = \{e \in I | \neg has\_type(source(e), SelectClause)\}$ Deals with other references to the column in another clause.

Recommendations:

- $C_1 \rightarrow$ DoNothing: If the column to be renamed is referenced through SQL wildcard ($\star$), nothing needs to be done. Indeed, as the wildcard does not refer to the column via its name but indirectly, the reference is not impacted by the renaming.
- $C_2 \rightarrow$ Choice(ChangeReferenceTargetInSelectClause, AliasColumnDeclaration): References to columns made in a `SELECT` clause define the names and types of the resulting table. Thus, when updating the reference made to a column in the `SELECT` clause, the architect needs to decide if the impact should be propagated to the users of the table resulting from the query execution (by only changing the reference target) or not (by changing the reference target and aliasing the column declaration with the old referenced entity name).
- $C_3 \rightarrow$ ChangeReferenceTarget: Other references need to be updated to refer to the new name of the column.

A.1.6 ChangeReferenceTarget

ChangeReferenceTarget changes the entity targetted by a reference by rewritting the corresponding source code. This operator must not receive a reference that occurs in a `SELECT` clause.
Signature: ChangeReferenceTarget(reference, newTarget)
Description: Changes the entity targetted by a reference by rewriting the corresponding source code.
Impact: None
Recommendations: None

A.1.7 ChangeReferenceTargetInSelectClause

ChangeReferenceTargetInSelectClause changes the target of a reference that occurs in a `SELECT` clause. This operator propagates the impact to users of the derived table resulting from the `SELECT` query execution.
Signature: ChangeReferenceTargetInSelectClause(reference, newTarget)
Description: Changes the target of a reference that occurs in a `SELECT` clause.
Impact: Let I be the set of entities impacted by the change;

- $C_1(I) = I$ Deals with the column of the derived table defined by the reference.

Recommendations:

- $C_1 \rightarrow$ RenameColumn: As the reference made in the `SELECT` clause is updated without aliasing, the change is equivalent to renaming the column of the table resulting from the `SELECT` query.

A.1.8 AliasColumnDeclaration

AliasColumnDeclaration changes the target of the reference in the `SELECT` clause and creates an alias with the old reference name. Because of the aliasing, this operator stops the propagation of impact.
Signature: AliasColumnDeclaration((view, reference, newTarget))
Description: Changes the target of the reference in the SELECT clause and create an alias with the old reference name.
Impact: None
Recommendations: None

A.1.9 DoNothing

DoNothing Does nothing, it is the null-operator.
Signature: DoNothing()
Description: Does nothing.
Impact: None
Recommendations: None

A.1.10 HumanIntervention

HumanIntervention Asks the database architect to perform the intervention on the source code of an entity. This operator generates no impact as we consider that the architect modifies the source code of the entity provided as argument in a way that there is no additional impact.
Signature: HumanIntervention(entity)
Description: Let the architect perform a manual editition of the source code of an

entity.
Impact: None
Recommendations: None

Bibliography

[Alalfi 2009] Manar H Alalfi, James R Cordy and Thomas R Dean. *WAFA: Fine-grained dynamic analysis of web applications*. In 2009 11th IEEE International Symposium on Web Systems Evolution, pages 141–150. IEEE, 2009. 36, 37

[Albe 2019] Laurenz Albe. *Tracking View Dependencies in PostgreSQL*. https://www.cybertec-postgresql.com/en/tracking-view-dependencies-in-postgresql/, 2019. Accessed 2020-06-23. 43

[Arnold 1996] Robert S Arnold and S.A Bohnert. Software change impact analysis. IEEE Computer Society Press, 1996. 38, 80

[Batini 1992] Carlo Batini, Stefano Ceri and Navathe Sham. Conceptual database design: an entity-relationship approach. Benjamin/Cummings, 1992. 29, 30

[Castellanos 1993] Malú Castellanos. *A methodology for semantically enriching interoperable databases*. In British National Conference on Databases, pages 58–75. Springer, 1993. 34, 37

[Chiang 1994] Roger HL Chiang, Terence M Barron and Veda C Storey. *Reverse engineering of relational databases: Extraction of an EER model from a relational database*. Data & knowledge engineering, vol. 12, no. 2, pages 107–142, 1994. 34, 37

[Cleve 2006] Anthony Cleve, Jean Henrard and Jean-Luc Hainaut. *Data reverse engineering using system dependency graphs*. In 2006 13th Working Conference on Reverse Engineering, pages 157–166. IEEE, 2006. 35, 37

[Cleve 2008] Anthony Cleve and Jean-Luc Hainaut. *Dynamic analysis of SQL statements for data-intensive applications reverse engineering*. In 2008 15th Working Conference on Reverse Engineering, pages 192–196. IEEE, 2008. 36, 37

[Cleve 2011a] Anthony Cleve, Jean-Roch Meurisse and Jean-Luc Hainaut. *Database semantics recovery through analysis of dynamic SQL statements*. In Journal on data semantics XV, pages 130–157. Springer, 2011. 36, 37

[Cleve 2011b] Anthony Cleve, Nesrine Noughi and Jean-Luc Hainaut. *Dynamic program analysis for database reverse engineering*. In International Summer School on Generative and Transformational Techniques in Software Engineering, pages 297–321. Springer, 2011. 36

[Codd 1970a] Edgar F Codd. *A relational model of data for large shared data banks*. Communications of the ACM, vol. 13, no. 6, pages 377–387, 1970. 2

[Codd 1970b] Edgar F Codd. *A relational model of data for large shared data banks*. Communications of the ACM, vol. 13, no. 6, pages 377–387, 1970. 46

[Curino 2008] Carlo A Curino, Hyun J Moon and Carlo Zaniolo. *Graceful database schema evolution: the prism workbench*. Proceedings of the VLDB Endowment, vol. 1, no. 1, pages 761–772, 2008. 40, 41

[Curino 2009] Carlo Curino, Hyun J Moon and Carlo Zaniolo. *Automating database schema evolution in information system upgrades*. In Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades, page 5. ACM, 2009. 40

[de Guzmán 2004] Ignacio García-Rodríguez de Guzmán, Macario Polo, Mario Piattini and S K Koskimies de Agosto. *Metamodels and architecture of an automatic code generator*. NWUML'2004, page 115, 2004. 34, 36

[de Guzman 2005] I Garcia-Rodriguez de Guzman, Macario Polo and Mario Piattini. *An integrated environment for reengineering*. In 21st IEEE International Conference on Software Maintenance (ICSM'05), pages 165–174. IEEE, 2005. 34, 36

[Delplanque 2017a] Julien Delplanque. *Software Engineering Issues in RDBMS, a Preliminary Survey*. In 16th edition of the BElgian-NEtherlands software eVOLution symposium (BENEVOL 2017), 2017. 28

[Delplanque 2017b] Julien Delplanque, Anne Etien, Olivier Auverlot, Tom Mens, Nicolas Anquetil and Stéphane Ducasse. *CodeCritics Applied to Database Schema: Challenges and First Results*. In 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2017. 69

[Delplanque 2018] Julien Delplanque, Anne Etien, Nicolas Anquetil and Olivier Auverlot. *Relational Database Schema Evolution: An Industrial Case Study*. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018. 11, 88

[Delplanque 2020] Julien Delplanque, Anne Etien, Nicolas Anquetil and Stéphane Ducasse. *Recommendations for Evolving Relational Databases*. In International Conference on Advanced Information Systems Engineering (CAiSE), 2020. 43, 79

[Englebert 1995] V Englebert, J Henrard, JM Hick, D Roland and JL Hainaut. *DB-MAIN: un atelier d'ingénierie de bases de données1, 2*. 1995. 35

[Gardikiotis 2006] Spyridon K Gardikiotis and Nicos Malevris. *DaSIAn: A Tool for Estimating the Impact of Database Schema Modifications on WEB Applications*. In Computer Systems and Applications, 2006. IEEE International Conference on., pages 188–195. IEEE, 2006. 39, 41

[Gardikiotis 2009] Spyridon K Gardikiotis and Nicos Malevris. *A two-folded impact analysis of schema changes on database applications*. International Journal of Automation and Computing, vol. 6, no. 2, pages 109–123, 2009. 39, 41

[Hainaut 2009] Jean-Luc Hainaut, Jean Henrard, Didier Roland, Jean-Marc Hick and Vincent Englebert. *Database reverse engineering*. In Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends, pages 181–189. IGI Global, 2009. 31, 32, 45, 49

[Haraty 2002] Ramzi A Haraty, Nashat Mansour and Bassel A Daou. *Regression testing of database applications*. Journal of Database Management, vol. 13, no. 2, pages 31–42, 2002. 39, 41

[Haraty 2004] Ramzi A Haraty, Nashat Mansour and Bassel A Daou. *Regression test selection for database applications*. In Advanced Topics in Database Research, Volume 3, pages 141–165. IGI Global, 2004. 39

[Henrard 1998] Jean Henrard, Vincent Englebert, Jean-Marc Hick, Didier Roland and Jean-Luc Hainaut. *Program understanding in databases reverse engineering*. In International Conference on Database and Expert Systems Applications, pages 70–79. Springer, 1998. 35, 37

[Hicks 2005] Michael Hicks and Scott Nettles. *Dynamic software updating*. ACM Transactions on Programming Languages and Systems, vol. 27, no. 6, pages 1049–1096, nov 2005. 27

[Karwin 2010] Bill Karwin. Sql antipatterns: Avoiding the pitfalls of database programming. Pragmatic Bookshelf, 2010. 69

[Kazman 1998] R. Kazman, S.G. Woods and S.J. Carriére. *Requirements for Integrating Software Architecture and Reengineering Models: CORUM II*. In Proceedings of WCRE '98, pages 154–163. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6. 2

[Lehman 1980] Manny Lehman. *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE, vol. 68, no. 9, pages 1060–1076, September 1980. 2

[Lehman 2000] Meir M Lehman. *Evolution as a noun and evolution as a verb*. Proc. Work. Software and Organisation Co-evolution: SOCE'00, 2000. 2

[Lehnert 2011] Steffen Lehnert. *A review of software change impact analysis*. Ilmenau University of Technology, page 39, 2011. 38

[Liu 2011] Kaiping Liu, Hee Beng Kuan Tan and Xu Chen. *Extraction of attribute dependency graph from database applications*. In Software Engineering Conference (APSEC), 2011 18th Asia Pacific, pages 138–145. IEEE, 2011. 40, 41

[Liu 2013] Kaiping Liu, Hee Beng Kuan Tan and Xu Chen. *Aiding maintenance of database applications through extracting attribute dependency graph*. Journal of Database Management, vol. 24, no. 1, pages 20–35, 2013. 40

[Manousis 2015] Petros Manousis, Panos Vassiliadis and George Papastefanatos. *Impact analysis and policy-conforming rewriting of evolving data-intensive ecosystems*. Journal on Data Semantics, vol. 4, no. 4, pages 231–267, 2015. 35, 36, 40

[Markowitz 1990] Victor M. Markowitz and Johann A. Makowsky. *Identifying extended entity-relationship object structures in relational schemas*. IEEE transactions on Software Engineering, vol. 16, no. 8, pages 777–790, 1990. 34, 37

[Maule 2008] Andy Maule, Wolfgang Emmerich and David Rosenblum. *Impact analysis of database schema changes*. In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, pages 451–460. IEEE, 2008. 39, 41

[med 2016a] *MediaWiki github repository*. https://github.com/wikimedia/mediawiki, 2016. Accessed 2016-11-26. 74

[med 2016b] *MediaWiki website*. https://www.mediawiki.org, 2016. Accessed 2016-11-26. 73

[Mens 2008] Tom Mens. *Introduction and roadmap: History and challenges of software evolution*. In Software evolution, pages 1–11. Springer, 2008. 2, 3

[Meurice 2016a] Loup Meurice, Csaba Nagy and Anthony Cleve. *Detecting and preventing program inconsistencies under database schema evolution*. In Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on, pages 262–273. IEEE, 2016. 26, 35, 37, 40, 41

[Meurice 2016b] Loup Meurice, Csaba Nagy and Anthony Cleve. *Static analysis of dynamic database usage in java systems*. In International Conference on Advanced Information Systems Engineering, pages 491–506. Springer, 2016. 35

[Meurice 2017] Loup Meurice. *Analyzing, understanding and supporting the evolution of dynamic and heterogeneous data-intensive software systems*. PhD thesis, Ph. D. dissertation, University of Namur, Namur, 2017. 32

[Naur 1969] Peter Naur and Brian Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968, Brussels, Scientific Affairs Division, NATO*. 1969. 1

[Papastefanatos 2008] George Papastefanatos, Fotini Anagnostou, Yannis Vassiliou and Panos Vassiliadis. *Hecataeus: A what-if analysis tool for database schema evolution*. In Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, pages 326–328. IEEE, 2008. 35, 36, 37, 40, 41

[Papastefanatos 2010] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis and Yannis Vassiliou. *Hecataeus: Regulating schema evolution*. In Data Engineering (ICDE), 2010 IEEE 26th International Conference on, pages 1181–1184. IEEE, 2010. 35, 36, 40

[Petit 1994] J-M Petit, Jacques Kouloumdjian, J-F Boulicaut and Farouk Toumani. *Using queries to improve database reverse engineering*. In International Conference on Conceptual Modeling, pages 369–386. Springer, 1994. 35, 37

[Polo 2002] Macario Polo, Juan Ángel Gómez, Mario Piattini and Francisco Ruiz. *Generating three-tier applications from relational databases: a formal and practical approach*. Information and Software Technology, vol. 44, no. 15, pages 923–941, 2002. 34, 36, 37

[Polo 2007] Macario Polo, Ignacio García-Rodríguez and Mario Piattini. *An MDA-based approach for database re-engineering*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, no. 6, pages 383–417, 2007. 34, 36

[Premerlani 1993] William J Premerlani and Michael R Blaha. *An approach for reverse engineering of relational databases*. In [1993] Proceedings Working Conference on Reverse Engineering, pages 151–160. IEEE, 1993. 34, 37

[Royce 1987] Winston W Royce. *Managing the development of large software systems: concepts and techniques*. In Proceedings of the 9th international conference on Software Engineering, pages 328–338, 1987. 1

[Sheth 1990] Amit P Sheth and James A Larson. *Federated database systems for managing distributed, heterogeneous, and autonomous databases*. ACM Computing Surveys (CSUR), vol. 22, no. 3, pages 183–236, 1990. 34

[Shoval 1985] Peretz Shoval. *Essential information structure diagrams and database schema design*. Information Systems, vol. 10, no. 4, pages 417–423, 1985. 34

[Shoval 1993] Peretz Shoval and Nili Shreiber. *Database reverse engineering: from the relational to the binary relationship model*. Data & knowledge engineering, vol. 10, no. 3, pages 293–315, 1993. 34, 37

[Sjøberg 1993] Dag Sjøberg. *Quantifying schema evolution*. Information and Software Technology, vol. 35, no. 1, pages 35–44, 1993. 39, 41

[Wang 2009] Hanzhe Wang, Beijun Shen and Cheng Chen. *Model-driven reengineering of database*. In 2009 WRI World Congress on Software Engineering, volume 3, pages 113–117. IEEE, 2009. 35, 37

[Yeh 2008] Dowming Yeh, Yuwen Li and William Chu. *Extracting entity-relationship diagram from a table-based legacy database*. Journal of Systems and Software, vol. 81, no. 5, pages 764–771, 2008. 34, 37