LESSQL: AN APPROACH TO DEAL WITH DATABASE SCHEMA EVOLUTION IN CONTINUOUS DEPLOYMENT

ARIEL ANTONY AFONSO

LESSQL: AN APPROACH TO DEAL WITH DATABASE SCHEMA EVOLUTION IN CONTINUOUS DEPLOYMENT

Dissertation presented to the Graduate Program in Informatics of the Universidade Federal do Amazonas in partial fulfillment of the requirements for the degree of Master in Informatics.

Advisor: Altigran Soares da Silva

Manaus

April 2020

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

A257l

Afonso, Ariel Antony
LESSQL: An approach to deal with Database Schema Changes in Continuous Deployment / Ariel Antony Afonso . 2020
55 f.: il. color; 31 cm.

Orientador: Altigran Soares da Silva
Dissertação (Ciência da Computação) - Universidade Federal do Amazonas.

1. schema changes. 2. continuous deployment. 3. database decay. 4. query language. I. Silva, Altigran Soares da. II. Universidade Federal do Amazonas III. Título

FOLHA DE APROVAÇÃO

"LESSQL: An approach to deal with Database Schema Changes in Continuous Deployment"

ARIEL ANTONY AFONSO

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

[illegible]

Prof. Altigran Soares da Silva - PRESIDENTE

[illegible]

Prof.ª Tayana Uchôa Conte - MEMBRO INTERNO

[illegible]

Prof. João Marcos Bastos Cavalcanti - MEMBRO EXTERNO

[illegible]

Prof. Alessandro Fabrício Garcia - MEMBRO EXTERNO

Manaus, 24 de Abril de 2020

“Embora ninguém possa voltar atrás e fazer um novo começo,
qualquer um pode começar agora e fazer um novo fim.”
(Chico Xavier)

Abstract

The adoption of Continuous Deployment (CD) aims at allowing software systems to quickly evolve to accommodate new features. However, structural changes to the database schema are frequent and may incur in systems' services downtime. This encompasses the proper maintenance of both schema and source code, including rewrites of all outdated queries that use the same database. Previous solutions try to mitigate the burdening task of manually rewriting outdated queries. Unfortunately, a software team must still interact with some tools to properly fix the affected queries. Moreover the team still has to locate and modify all the impacted code, which are often error-prone tasks. Thus, a project may not experience CD benefits when changes impact various code regions. In this thesis, we present an alternative approach, called *LESSQL*, whose goal is to improve queries' stability in the presence of structural schema changes over time. LESSQL supports queries that are less dependent on the database schema since they do not include the `FROM` clause. An underlying framework intercepts each LESSQL query and generates a corresponding SQL query for the current schema. It also locates the query attributes in the current schema and generate proper expressions to join required tables. LESSQL supports unsupervised, supervised and hybrid configurations to process mappings of attributes to a newer schema version. We conducted experiments in two open-source applications: Wikipedia, an online and popular information system, and WebERP, a web-based accounting and business management system. Experiments outcomes indicate that our approach is effective in significantly reducing the modifications required for applying schema changes, allowing to better reap the benefits of CD. While supervised and hybrid configurations achieved a success rate higher than 95% with a minor query generation overhead, the unsupervised configuration was also successful for certain types of structural schema changes. These results show that LESSQL effectively favors CD and keeps queries running after database schema changes without services interruption.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Continuous Deployment (CD) is a development process that automates software building, testing, releasing and deployment (Laukkanen et al., 2017). Adopting CD enables developers to keep up with fast-changing markets and allows applications to quickly evolve to accommodate new features (Rossi et al., 2016; Savor et al., 2016). However, developing new features may require structural changes to the database schema and incur in application service downtime due to the time it takes to perform tasks such as: (i) predicting and estimating effects before each structural schema change is performed (Qiu et al., 2013), (ii) locating and rewriting the outdated queries in the source code (De Jong et al., 2017; Curino et al., 2013; Qiu et al., 2013), (iii) evaluating the cost of reconciling the existing code *w.r.t.* the new schema before any schema changes (Qiu et al., 2013), and (iv) locating and modifying all impacted code regions after applying the schema changes (Qiu et al., 2013). The cost of performing these tasks is not in concert with CD goals of reducing changes along each cycle and quickly producing correct builds. Moreover, the high commit throughput due to frequent updates can hugely impact the application runtime depending on the number of changes that affect the schema. In particular, changes that directly affect attributes names or locations in the schema are harmful.

In enterprises, it is often the case that database administrators (DBAs) try to circumvent the required changes by adding spurious schema elements, resulting in a superset of what is needed (Savor et al., 2016). In the long run, this implies in a large gap between the current conceptual database schema and the logical database schema, which may lead to serious maintenance and performance problems. This phenomenon is known as *database decay* (Stonebraker et al., 2016). In order to tame the database decay at some point, DBAs will have to perform complex changes to the schema. Meanwhile, developers will also have to rewrite affected queries and perform major refactoring in the application source code.

Studies in the literature indicate that database schema changes occur as often as once per week (Qiu et al., 2013). Thus, enterprises cannot afford to interrupt the service that often, neither to stop the development of new features for providing support to existing services during the evolution. An empirical study about the co-evolution of source code and database schema in ten open-source applications found out that for every schema change, approximately 10 to 100 lines of code were modified to keep the functionality running (Qiu et al., 2013). Another research based on two enterprises, Facebook and OANDA, found out that every developer pushed around 2 to 3 application updates per week (Savor et al., 2016). Considering that every developer commit is independent, a change on the schema can hugely impact the overall development environment since it may force developers to search and fix for affected functionalities and queries in the application source code.

To better illustrate the problems caused by schema changes, consider that in an initial schema version, attributes `uname` and `urights` both belong to relation `USER`. In a newer schema version, `uname` belongs to relation `USER` and attribute `urights` belongs to relation `USER_RIGHTS`. Figure 1.1(a) presents an SQL query S that retrieves attributes `uname` and `urights` for a particular user, considering the initial schema version. Query S is outdated by the newer schema version, so that, another SQL query S', illustrated in Figure 1.1(b), must be specified to retrieve the same attributes. This implies in source code maintenance and eventually in service interruption.

(a) Query S

```
SELECT uname, urights
FROM USER
WHERE uid = 'X'
```

(b) Query S'

```
SELECT uname, urights
FROM USER UD
JOIN USER_RIGHTS UR
  ON UD.uid = UR.uid
WHERE uid = 'X'
```

Figure 1.1: Equivalent queries in different schema versions

In the literature, there are several proposals that address aspects related to the co-evolution of software systems and database schemas. Recently, Caruccio et al. (2016) presented comprehensive survey on the problem of query rewriting upon schema evolutions. They discuss and compare several techniques in the literature for assisting developers in the tasks of adapting query and views embedded in the application code when the underlying databases schema changes. One of the most representative techniques based on such an approach is PRISM++ (Curino et al., 2013), a framework for assisting database schema migrations and automating the process of rewriting queries when there are changes in the schema. In all of the solutions that follow this approach, the DBA must interact with some tool or assistant to rewrite queries affected by schema changes. Moreover, the operation of the application services must be interrupted so that the the new version of the schema and the adapted queries are deployed. This represents a significant burden if changes are frequent, thereby harming CD goals.

To avoid the problem of application services interruption, an approach adopted in the industry is to allow different schema versions to co-exist in the database. Some systems developed in the industry based on such an approach are Gh-ost (Berquist and Gunson, 2018) by Github, TableMigrator (Saur et al., 2016) by Twitter and Online Schema Change by Facebook (FOS, 2019). A representative solution based on such an approach is QuantumDB (De Jong et al., 2017). The main idea of this tool is that the old and new schemas are active in the database while the process of migrating database instances happens in the background. Then, the tool is responsible for correctly assigning each query to the schema version it refers to. However, developers must still search and fix all the queries to reflect the changes applied to the schema.

In our work, we present a novel approach whose goal is to allow queries written for an initial schema version to be executed on a newer version of the same schema, without rewriting these queries. As a consequence, service interruption and slow CD cycles due to source code maintenance are avoided. Our approach, called *LESSQL* (**Les**s **S**tructured **Q**uery **L**anguage), is based on queries that are less dependent from the database schema since they do not include the `FROM` clause, providing more flexibility than regular SQL queries. When the application issues a LESSQL query, our framework intercepts this query, generates an actual SQL query suitable for the newer schema and sends it to be processed by the Relational Database Management System (RDBMS).

To this intent, the LESSQL framework locates the query attributes in the newer schema version and uses a graph-based algorithm we designed, called *JNG*, to generate a proper expression to join the required relations. To locate the query attributes, as some of them may have been renamed, relocated or removed in the newer schema version, LESSQL relies on a set of attribute mappings. Specifically, these mappings describe changes to attributes regarding the newer schema version. That is, they indicate where missing attributes can be found. Also, LESSQL was designed to operate in a supervised configuration, where the DBA supplies attribute mappings or in an unsupervised configuration, where a schema matching algorithm we propose, called *SchemaDiff*, automatically generates these mappings. Additionally, we consider a hybrid configuration that combines the supervised and unsupervised configurations, where the DBA supplies a small and non-trivial set of attribute mappings.

We conducted experiments in two open-source applications: Wikipedia, an online and popular information system, and WebERP, a web-based accounting and business management system. Both projects were used in previous well-known research on schema and application source code evolution (Curino et al., 2013; Qiu et al., 2013; Vassiliadis, 2016). In both applications, we observed that only a small fraction of the queries originally written for a given schema can be executed when this schema changes. This fraction, we call Success Rate (SR), can be as low as 0.23 in the case of Wikipedia and 0.13 in the case of WebERP. Our experiments show that LESSQL can largely improve this metric to 0.96, in the case of the supervised and hybrid configurations, and 0.91 in the unsupervised configuration. These results indicate that our approach is effective in reducing the manual effort required for dealing with frequent schema changes that affect application code, allowing our method to better reap the benefits of CD.

In summary, we present the following contributions: a novel approach for reducing source code maintenance while handling structural changes to schemas; an algorithm called JNG that allows queries written for an initial schema version to be executed on newer schema versions; an algorithm called SchemaDiff, for automatically finding mappings between attributes of two schema versions; and an empirical evaluation of our approach on the maintenance of an open-source project.

This proposal is organized as follows. Chapter 2 provides a background for the problem of co-evolution of database and application source code as well as database decay and database migrations, along with researches that aim at studying and solving these problems. Chapter 3 illustrates an overview of the LESSQL framework and its processing stages. In Chapter 4, we present the concepts necessary to understand the generation of the joining network expression, which is used to compose the SQL query that will be issued to the DBMS. Chapter 5 details how LESSQL is able to automatically generate a set of attribute mappings through our SchemaDiff algorithm. In Chapter 6, we present the experimental results of LESSQL when validating in two open-source applications. Finally, Chapter 7 details the next step towards improving the LESSQL framework.

Chapter 2

Background and Related Work

In this chapter, we present studies in the literature that propose to assist developers when the database schema changes. Section 2.1 presents the co-evolution of software systems and database schemas approach. Section 2.2 presents a new paradigm where developers specify database operations differently, for instance, without SQL queries. Section 2.3 presents an alternative approach that is widely adopted in the industry to prevent application downtime when the database schema changes. Finally, Section 2.4 presents studies on the problem of co-evolution of software systems and database schemas.

2.1 Co-evolution of Software Systems and Database Schemas

Through the years, the most successful approach for handling evolution of database schemas in software development is the so-called *co-evolution of software systems and database schemas*(Qiu et al., 2013). This approach consists in providing tools to support code maintenance and evolution to keep it up to date with the schema changes. Issues related to co-evolution have been addressed by several proposals in recent literature.

Recently, Caruccio et al. (2016) presented a comprehensive study that surveys several methods and techniques based on this approach. This study covers three different strategies for handling the problem of co-evolution. The first strategy, called *operation-based evolution*, requires the database administrator to describe the evolution of the schema through a set of predefined schema change operations, such as relation decomposition or merging. The second strategy, called *mapping-based evolu-*

tion, expects the systems to describe the schema evolution through a set of mappings between schema elements from two database schema versions. The last strategy, called *hybrid* merges the two previous strategies.

One research that represents the mapping-based strategy is the *Generic Model Management* (GMM) tool (Bernstein and Melnik, 2007). This tool focuses on reducing the programmatic effort in handling mappings between schemas. To this intent, the system provides operators that describe the evolution of a database schema through mappings. One important operator described is the *Match* operator, which takes as input two schema versions, namely, the source schema and the target schema, and creates mappings between schemas elements. These mappings aid the developer in updating outdated queries by deriving from the new schema what happened to the old schema elements that are referred to in these queries.

Another research that is classified as a mapping-based strategy was proposed by Polese and Vacca (2009b). This work introduced the concept of *Default Mappings* to achieve a higher query rewriting ratio than that of GMM. The authors provide an approach to recover information lost during an evolution where the DBA provides a formula based on default logic that states whether an information can be recovered or not. The information to be recovered is programatically predefined. Upon a database schema change that removes an attribute, the DBAs must provide a formula that "outputs" a result that otherwise would be retrieved from the removed attribute. According to the authors, there should be effort towards retrieving information that would be lost after a database schema changes. Thus, interrupting application functionalities.

Alternatively, Polese and Vacca (2009a) provide a cooperative approach based on Hintikka interrogative logic (Hintikka and Bachman, 1991). This cooperative process is handled as a dialogue between the DBA and the system where the main goal is to find adequate mappings between a query and the target schema. The user queries for information on the target schema and, if not found, the system derives deductions to where the information from the query could possibly be found. Although the interaction between DBA and system is friendly, handling a dialogue can be a long drag depending on the operations executed over the database schema.

A representative solution based on the co-evolution approach is the PRISM workbench (Curino et al., 2008), later updated to PRISM++ (Curino et al., 2013). The workbench interface allows the DBA to define a set of atomic change operations to be applied on the database schema, named Schema Modification Operators (SMOs). SMOs are used to describe step by step instructions to transform the current schema in a new one. Using SMOs, PRISM/PRISM++ can support two main tasks: migration of the database instance from the current schema to a new one and rewriting queries embedded in the application code, thus reducing the burden of manually rewriting them.

Wang et al. (2019) recently proposed a new co-evolution approach. Their tool, called *Migrator*, aims at automatically synthesizing a new version of the database program given its original version and the source and target schemas. The authors argue that they avoid the developers or DBAs having to specify modification operators (e.g., SMOs). However, their approach still requires code to be refactored, albeit automatically, to accommodate schema changes.

In fact, all the work based on the co-evolution approach, including PRISM++, try to avoid the burdening task of manually rewriting queries when schema structure changes. However, the developers or DBAs must still interact with some tool or assistant to rewrite queries affected by schema changes besides searching for the affected queries. Notice that they neither prevent, nor reduce, source code maintenance. Nonetheless, fixing the code represents a significant burden if changes are frequent and time-consuming, hampering the achievement of CD goals.

In our work, we propose an alternative to the co-evolution approach. Our goal is to allow queries written for an early schema version to be executed on a newer version of the same schema, so that queries written once do not need to be modified when database schema changes. This avoids rewriting queries and the consequences of service interruptions due to source code maintenance. Additionally, LESSQL relies on a simple way to provide mappings between two schema versions when necessary, reducing even more the burdening task of providing support to application functionalities affected by schema changes. This is a major distinction of our work with respect to the co-evolution approach.

2.2 Database Decay

Recently, Stonebraker et al. (2016) found out that, in practice, DBAs avoid making changes, that are considered as "good practices", to a database schema. This is due to an attempt on minimizing application source code maintenance since these changes may span across multiple teams. In the long run, this implies in a large gap between the current conceptual database schema and the logical database schema, which may lead to software maintenance and performance problems. This phenomenon is known as *database decay*.

The authors present two solutions for the software maintenance problem: (i) coding "defensively", trading performance for resiliency and (ii) adopting a new application development paradigm, in which developers specify database operations differently. In the first solution, developers would avoid application maintenance and services interruption at the cost of database decay. In the second solution, developers would use an alternative paradigm for developing applications. In this paradigm, developers specify database operations using *messages* instead of SQL queries. Each message is associated with an SQL query template previously built during application development. During operation, each message is sent over to a middleware which selects the corresponding query template, builds an actual SQL query based on this template and executes this query. Their goal is to centralize SQL query templates in the middleware, so that, when database schema changes, all affected queries can be easily fixed. The authors indicate that adapting templates to a new schema can be done automatically and that there are cases in which human intervention is required to validate this process. This manual validation slows CD cycles. Moreover, no algorithmic details on this process have been published so far.

Our work in in-line with the second solution. Although, in our case, developers would still use SQL-like logical-level queries instead of conceptual-level abstract messages, thus, easing the adoption of our approach by developers. This makes it viable to automatically generate the actual SQL query from any LESSQL query using our JNG algorithm, without having developers to define a query template in advance as it is done in Data Civilizer. As a consequence, there are no templates to be fixed either automatically or manually, when the schema changes. Furthermore, to the best of our knowledge, our work is the first that implements such a paradigm with concrete algorithm description and empirical evaluation.

2.3 Database Instance Migration

A more critical scenario for schema evolution takes place in environments which adopt continuous deployment as a software development process. CD aims at frequent feature releases and more developer freedom in the choice of when deploying these features. If a change affects the database schema, the application can be interrupted and effort must be put in finding the defective commit. To this intent, the industry applies a technique that keeps more than one database schema active at the same time. That is, multiple application versions are operational but, still, the developers must fix the queries in the source code. Also, this state that keeps different database schema actives runs a database instance migration on the background, transferring information from the old schema to the new schema.

The industry shared proprietary tools that are used to aid in the database migrations and schema evolutions. They are: Gh-ost (Berquist and Gunson, 2018) by Github, TableMigrator (Saur et al., 2016) by Twitter, Online Schema Change by Facebook (FOS, 2019) and Large Hadron Migrator by Soundcloud (SLH, 2019). Other solution used in the industry is Liquibase, a database refactoring and migration tool. However, Liquibase needs a system shutdown to execute schema migrations. Notice that queries must still be rewritten by developers, not minimizing the effort to search and fix all the queries affected.

These strategies adopted by the industry were used as baseline in a solution that aims at a zero-downtime database schema evolution. De Jong et al. (2017) keeps multiple database schema versions active at the same time and focuses on enterprise environments that use continuous deployment as development process. The MySQL[1] and PostgreSQL[2] RDBMSs were used to indicate which operations used for changing a database schema blocked read and write operations. Then, the authors developed a tool, called *QuantumDB*, which acts as a middleware to all the applications that use the database. During schema evolution, *ghost* tables are created to mirror the database tables affected by a set of schema changes, namely, a *changeset*. This state where the database has duplicate tables, each for a different application version, is called *mixed-state*. In this state, each application version is fully operational and data is transferred in the background. For this, QuantumDB must delegate each application query to the correct schema version.

Although circumventing the locking nature of some database operations is an ideal scenario, the applications evolve and the source code that is affected by a database schema change still must be fixed since different application versions are active. This solution can be used complementary to ours, as we focus on minimizing source code maintenance and not database instance migrations.

2.4 Experimental Studies

Besides providing solutions to the problem, many studies in the literature focus on exploring the richness of open-source database applications to study how frequently and significantly software and databases co-evolve and how they impact in application source code or procedures that are stored inside the database.

The first large-scale empirical study that attempts to study the co-evolution was presented by Qiu et al. (2013). This study was based on ten open-source databases to

[1]https://www.mysql.com/

[2]https://www.postgresql.org/

understand how database changes affected the applications source code. The authors found out that, for every atomic schema change (e.g. ADD COLUMN, ADD TABLE) approximately 10 to 100 lines of code (LoC) were co-changed. This situation gets even worse since applications presented 15 to 300 atomic schema changes in a one year period. Thus, some schema changes are harmful to the application operational state since they may deprecate some queries written for a previous version of the schema. After applying these changes, the developers must rewrite the affected queries and refactor the surrounding code to reflect the changes on the schema. These maintenance implications affect CD cycles and are harmful to the development environment, since there will be an interruption to properly fix the affected functionalities.

Another study was based in a complex university database (Delplanque et al., 2018). Roughly, this database presented 62 views and 64 stored procedures. It is considered complex not only for the stored views and procedures but because several laboratories fork the "main" database and evolve each fork database schema independently, according to its own requirements. This introduces another problem since changes to the "main" database schema must be applied to the other database schema instances. To ease the maintenance of the different application accessing the database instances, the database architect implemented stored procedures to maintain consistency across all applications. The authors detected problems that prevent easy transition between schemas, such as: analysing dependencies between database entities, evaluating the impact of a modification in the database, managing the co-evolution of multiple instances of the database and testing the database functionalities, that is, if the results returned from the database were unmodified by the schema change.

Overall, previous studies present real case scenarios that are hugely affected by database schema changes. Besides dealing with database schema changes, developers also need to provide maintenance on applications that are affected by these changes. Considering a continuous deployment scenario, these changes imply in a burdensome search and fix for affected code regions. Thus, this maintenance problem requires a solution which reduces the human effort needed to keep applications up to date with the database schema and can better reap the benefits of CD.

Chapter 3

LESSQL Framework

In this chapter, we present an overview of the LESSQL framework and of how it addresses issues in software development caused by database schema changes. Also, we introduce the execution flow for all different configurations of LESSQL: the supervised configuration, the unsupervised configuration and the hybrid configuration.

3.1 Framework Overview

In order to keep applications functional when the database schema changes and accelerate CD cycles, LESSQL relies on the use of database queries that are less dependent on the database schema than regular SQL queries. To this intent, queries do not include the `FROM` clause. Therefore it is not necessary to specify join conditions to define where the attributes are located or how they are associated.

(a) Query L	**`SELECT uname, urights`**
	`WHERE uid = "X"`

Figure 3.1: A LESSQL query example

For instance, considering Figure 1.1 (a), we propose that instead of query S, the developer would write the query L shown in Figure 3.1 (a), which is an example of a LESSQL query. Instead of being issued directly to the DBMS, query L would be sent to the LESSQL framework. Depending on the schema version, the framework would automatically generate query S or S' from Figure 1.1. Thus, no query rewriting would be necessary when the schema changes.

We highlight that query L does not include the `FROM` clause. Thus, it is up to the framework to locate the attributes in the current schema and eventually to generate a proper joining network expression to retrieve the require attributes. The joining network expression generation relies on a novel algorithm we developed, named JNG (Chapter 4), that is based on algorithms previously proposed to handle keyword-based queries in relational databases (Hristidis and Papakonstantinou, 2002; Oliveira et al., 2018).

In the particular case of the example in Figure 3.1, the attributes involved in the query are the same in both versions, and only their locations have changed. However, it is often the case that attributes referred in queries in a given schema version are renamed, removed or relocated in newer versions of the schema. To address this, LESSQL relies on a set of attribute mappings that define the changes made on attributes between schema versions. These attribute mappings can be seen as a simplified form of the *Schema Modification Operations* or *SMOs* (Curino et al., 2013) (Chapter 2), where only attribute-related operations (e.g. RENAME COLUMN, ADD COLUMN, DROP COLUMN) are considered.

(a) Query S	**`SELECT cur_restrict`**
	`FROM cur`
	`WHERE cur_id = "X";`

(b) Query S'	**`SELECT page_restrict`**
	`FROM page`
	`WHERE page_id = N;`

(c) Query L	**`SELECT cur_restrict`**
	`WHERE cur_id = "X"`

(d) Attribute Mappings	**`cur_restrict`** $\rightarrow$ **`page_restrict`**
	`cur_id` $\rightarrow$ **`page_id`**

Figure 3.2: Example of a LESSQL query and attribute mappings

Figure 3.2 depicts another schema evolution scenario. Here the attributes are renamed and relocated from relation `CUR` to a new relation `PAGE`. The developer for the application wrote query S, supposed to work in the early version of the schema. Due to the evolution, query S is deprecated and must be rewritten as query S'. Similar to the example in Figure 3.1, the developer can write a query that works in both schema versions. To this intent, he should write query L. This query can be used as replacement for both queries S and S', with the assistance of the attribute mappings shown in Figure 3.2(d). When the query is issued to the LESSQL framework, the set of attribute mappings is used to correctly translate the query to the new schema.

Attribute mappings can be supplied by the database designer in a similar way as *SMOs*. However, there may be cases in which manually generating mappings can be burdensome and error-prone. This can happen when the schema has many attributes, when the volume of changes is large or when many schema evolutions occur in a short period of time. In particular, this last issue is commonly found in CD scenarios. If not properly addressed, these issues may hamper the CD process due to the necessity of manual intervention. To deal with them we also developed an algorithm for automatically generating attribute mappings, given two versions of the same schema, called *SchemaDiff*. This algorithm is based on ideas commonly applied in schema matching methods (Rahm and Bernstein, 2001), adapted to the context of schema changes.

3.2 Framework Architecture

In Figure 3.3, we illustrate the LESSQL framework architecture. Following next, we explain how the process of generating SQL queries takes place.

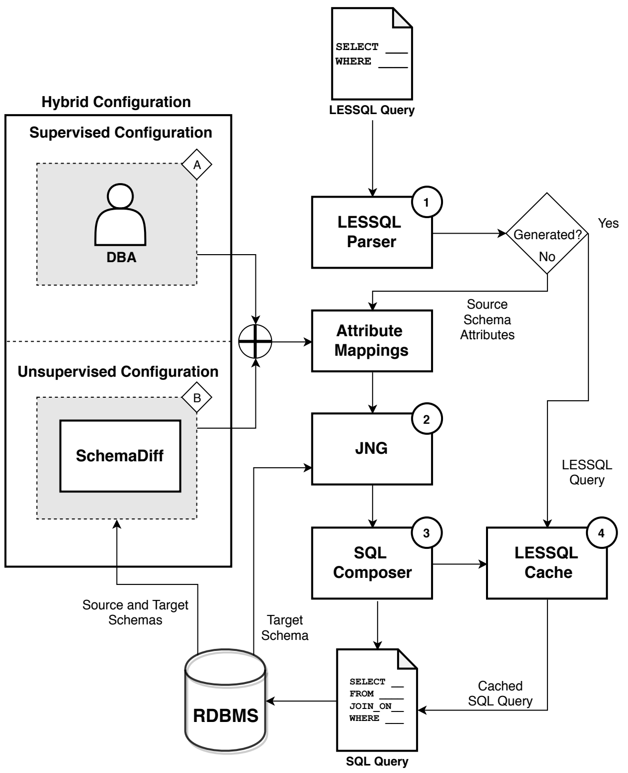

Figure 3.3: LESSQL Framework Overview

The process begins when a LESSQL query is received by the framework. The first step is to parse the query in the *LESSQL parser* (1). The parser first checks if the input LESSQL query uses the same template of another query that has been previously issued by the application for the current schema version. In this context, a query template is a query that does not present any instanced value in its `WHERE` clause, that is, it does not specify the attribute values in selection conditions that must be met. This strategy allows us to verify if a LESSQL query has been issued previously by hashing the query template. If this is the case, there is no need to generate the corresponding SQL query template once more, so we just forward the SQL query template stored and finish the process. Otherwise, if the query template is not found in the LESSQL cache, the parser extracts all the attributes that the query refers to and supplies them to the Joining Network Expression Generation algorithm, JNG.

Given the source schema attributes, the LESSQL framework may require a set of *attribute mappings* to generate the actual SQL query to the target schema. An attribute mapping is a pair which informs, for an attribute from the source schema, an equivalent attribute in the target schema, as illustrated in Figure 3.2 (d). The source schema attributes and the attribute mappings are given as input to the JNG algorithm. The source that supplies the mappings depends on the LESSQL configuration. In the case of the *supervised* configuration ⟨A⟩, the DBA is expected to manually supply all the mappings that comprise the changes between the source and the target schema. In the case of the *unsupervised* configuration ⟨B⟩, our SchemaDiff algorithm automatically generates mappings by tracking attributes that suffered transformations between two schema versions.

In the *Joining Network Expression Generation* step ②, the framework gets the set of attributes that were updated in the last step and identifies the relations that contain these attributes. With these relations, LESSQL generates a joining network expression that connects all these relations based on referential integrity constraints from the target schema.

The last step happens in the *SQL Composer* ③ module. This module translates the joining network expression generated in the previous phase to a `FROM` clause and its corresponding join conditions, outputting a full SQL query. Then, the query is sent to the DBMS, which returns the answer to the query and to the LESSQL cache ④, which stores the full SQL query and the LESSQL query template that was issued for later verification.

In some situations, the DBA applies a significant schema change and LESSQL effectiveness may be compromised if many mappings are generated. In this scenario, a possibility is to combine the benefits of the supervised configurations, manually supplying mappings that are considered difficult to track, with the benefits of the unsupervised configuration, letting the framework handle the rest of the cases. For these cases, we provide the *hybrid* configuration. This configuration keeps the overall translation process effective and allows application functionalities to keep running in the event of large schema modifications. This is very helpful in a CD context, as the service must be available at all times (De Jong et al., 2017).

3.3 Adopting LESSQL in practice

Adopting LESSQL in a development environment implies replacing the majority of SQL interactions specified in the application source code by LESSQL-based interactions. In addition, the DBA must choose the LESSQL configuration that is more suitable to their development scenario in the given time. For instance, if the changes that must be applied to the database schema are subtle and easy to identify, such as a single element renaming in a single table, then our unsupervised configuration can be leveraged. Nonetheless, if a significant volume of changes is to be applied to the database schema, then our supervised or hybrid configuration may be the best alternative.

According to the chosen configuration, LESSQL behaves differently from the point of view of how the attribute mappings are provided. When changes are applied to the database schema changes and the unsupervised configuration of LESSQL is being used, the SchemaDiff algorithm will retrieve the source schema, that is, the schema from the previous application version and the target schema, that is, the schema from the new application version. After retrieving both schemas, it will generate the attribute mappings according to our heuristics. This process happens when developing the application, more specifically, when supplying the new schema for the database. The same applies to the hybrid configuration. However, in this case, the DBA can provide a small set of attribute mappings for which SchemaDiff was not able to capture the semantics of the schema change. As for the supervised configuration, the DBA will provide the entire set of attribute mappings.

Chapter 4

Joining Network Expression Generation

In this chapter we describe our algorithm for generating joining network expressions. This algorithm, called JNG, is based on ideas from algorithms previously proposed for Relational Keyword-Search Systems (R-KwS) (Oliveira et al., 2018)(Hristidis and Papakonstantinou, 2002). Regardless of the configuration, LESSQL uses the JNG algorithm to find a way of correctly joining the relations whose attributes are used in a given LESSQL query template. Precisely, the algorithm generates a relational expression that correctly joins these relations, which, then, form a network. We also review concepts necessary for better understanding how JNG generates these joining network expressions.

4.1 Overview

As we discussed in Section 1, LESSQL queries are designed to omit the `FROM` clauses and join conditions, with the ultimate goal of minimizing code maintenance when the database schema changes. As a consequence, DBMSs cannot handle them directly. To address this, LESSQL uses the JNG algorithm to help compose a full SQL query template. JNG receives as input the attribute mappings and the attributes from the LESSQL query template, which are used to filter only the mappings that will be used to generate a joining network expression. Essentialy, joining network expressions are sets of relations that are connected to each other and contain attributes from the extracted query attribute set. After generating the correct joining network expression, LESSQL composes an actual SQL query that is forwarded to the RDBMS for processing the answer to be returned to the application and to the LESSQL cache for later retrieval in case a similar structured query is issued against the LESSQL framework.

4.2 The JNG algorithm

Before presenting the JNG algorithm, we introduce some important concepts that will be used in the algorithm description.

The first concept we introduce is the one of a *query template*. Although we have been using it informally through the text, it is formally stated below.

Definition 1. Let Q = `SELECT user_id, old_text WHERE page_id = 10` be a LESSQL query. A LESSQL query template is a query that does not present any instanced value in the `WHERE` clause.

The corresponding LESSQL query template L for Q is given by `SELECT user_id, old_text WHERE page_id = 'X';`.

Definition 2. Let L be a LESSQL query template. Also, let $A_S = \{A_1, \ldots, A_k\}$ be the set of attributes in the `SELECT` clause and $A_W = \{A_{k+1}, \ldots, A_n\}$ be the set of attributes in the `WHERE` clause. The *attribute set* of L, A_L, is given by $A_S \cup A_W$.

Consider the following LESSQL query template L:

```
SELECT user_id, old_text WHERE page_id = 'X';
```

According to Definition 2, the attribute set A_L of L is given by {`user_id`, `old_text`, `page_id`}.

Definition 3. Let S be a database schema version. Let A_L be the attribute set of a LESSQL query template L. If a relation R in S contains any of the attributes from A_L, it is called a *non-free relation* with respect to L, otherwise its called a *free relation.*

Definition 4. Let S be a database schema version. The *schema graph* of S is a pair $G_S = \langle V, E \rangle$, where V represents the set of relations in the database schema and E represents the set of referential integrity constraints between the relations in V.

In Figure 4.1, we illustrate an example of a schema graph that corresponds to an excerpt of a database schema version from the Wikipedia database, which we have been using in our examples. In this schema graph, we also indicate the relations where each attribute of A_L is defined.

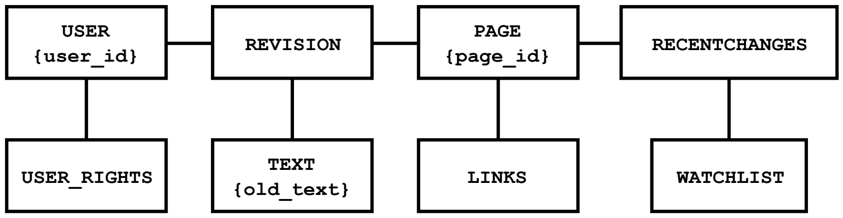

Figure 4.1: An excerpt of the schema graph from the Wikipedia dataset.

Definition 5. Let G_S be a schema graph for a given LESSQL query template L. We call a *joining network expression* a tree $J \subseteq G_S$ in which all leaves are *non-free relations* with respect to L.

Figure 4.2 illustrates some of the joining network expressions that are built using the schema graph supplied in Figure 4.1.

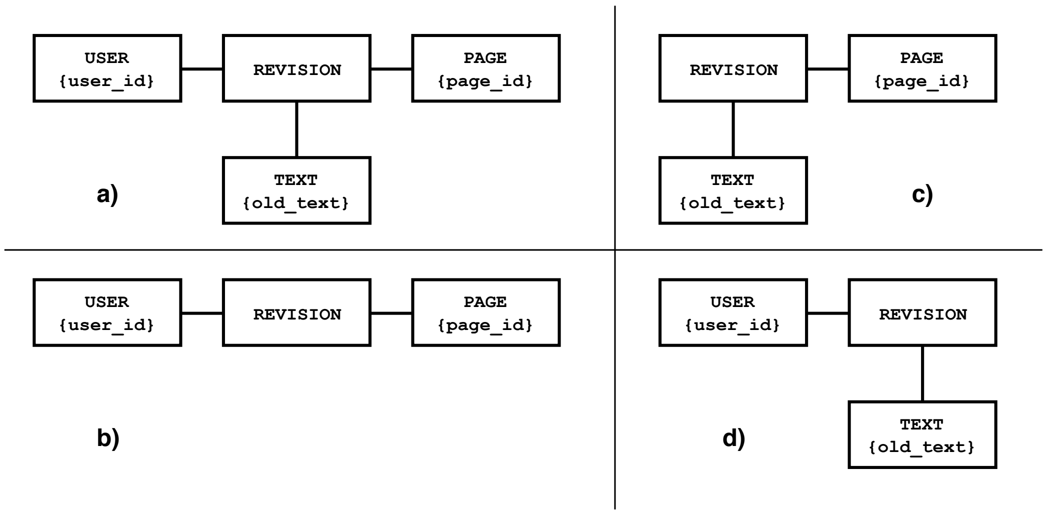

Figure 4.2: Joining network expressions built from the Wikipedia schema graph.

The main objective of JNG is to find a joining network expression on the schema graph that is composed of relations that contain all the elements of the attribute set of a LESSQL query template. When a joining network expression J contains all attributes of an attribute set A, we say that J *covers* A. As a consequence, this network is said to be a *query joining network expression*.

Definition 6. Let L be a LESSQL query template whose attribute set is A_L. Also, let J be a joining network expression in a schema graph G_S. We say that J is *total* in A_L if every attribute from A_L appears in at least one non-free relation of J. We say that J is *minimal* in A_L, if removing any non-free relation from J, it is no longer total in A_L. If J is *total* and *minimal* in A_L, we say that it *covers* A_L and that it is a *query joining network expression* for L.

An example of query joining network expression J that covers the attribute set A_L is shown in Figure 4.2 (a).

The JNG algorithm is detailed in Algorithm 1. It receives as input the attribute set of a LESSQL query template and a schema graph corresponding to the version of the schema over which this query template will be executed. JNG outputs a query joining network expression that is used to compose the `FROM` clause of the final SQL query template.

Algorithm 1: JNG

```
Input:
$A_L$: the attribute set of a LESSQL query template.
$G_S$: a graph for a given schema version.
Output:
$J'$: a joining network expression.
1  begin
2      $\mathcal{P} \leftarrow \emptyset$;
3      $J \leftarrow \emptyset$;
4      let $A_L = \{A_1, \ldots, A_n\}$
5      let $R_1$ be the relation which contains attribute $A_1$ from $A_L$
6      $J.V \leftarrow \{R_1\}$;
7      $enqueue(\mathcal{P}, J)$;
8      while $\mathcal{P}$ is not empty do
9          $J \leftarrow dequeue(\mathcal{P})$;
10         foreach relation $R_i \in G_S$ adjacent to some $R_u \in J.V$ do
11             if $R_i \notin J.V$ then
12                 $J' \leftarrow J$;
13                 Expand $J'$ by adding $R_i$ to $J.V$ and $\langle R_u, R_i \rangle$ to $J.E$
14                 if $J' \notin \mathcal{P}$ then
15                     if the relations in $J'.V$ covers $A_L$ then
16                         return $J'$
17                     else
18                         $enqueue(\mathcal{P}, J')$;
```

JNG starts by fetching the first attribute in the attribute set and storing its respective relation in the joining network expression J (Lines 5–6). This relation is used as the starting node when traversing the graph. Then, JNG fetches relations that are adjacent to any relation present in the current joining network expression J (Line

10). While traversing the graph, JNG expands J with either free or non-free relations. After each expansion, JNG checks if the joining network expression was not generated previously (Line 14). If this is the case and the joining network expression covers the attribute set A_L, it is returned. Otherwise, it is enqueued for later processing.

For a better understanding of JNG, we illustrate its execution in a real case scenario of the Wikipedia database. In this execution, consider the schema graph to be the one illustrated in Figure 4.1. Considering the same LESSQL query template L given previously and its attribute set, JNG generates several joining network expressions to find the first that covers the attribute set. Notice that in Figure 4.1, we indicate the relations of the attributes from the LESSQL query template, so the query joining network expression must contain three non-free relations: `USER`, `TEXT` and `PAGE`.

#I	Joining Network	Action
0	`USER`	initialize J (enqueued)
1	`USER — REVISION`	expand 0 (enqueued)
2	`USER_RIGHTS — USER`	expand 0 (enqueued)
3	`USER — REVISION — PAGE`	expand 1 (enqueued)

4	`USER — REVISION`
	`|`
	`TEXT`
		expand 1 (enqueued)

5	`USER_RIGHTS — USER — REVISION`	expand 1 (enqueued)
6	`USER_RIGHTS — USER — REVISION`	expand 2 (pruned)

7	`USER — REVISION — PAGE`
	`|`
	`TEXT`
		expand 3 (**returned**)

Figure 4.3: A brief JNGen execution

In Figure 4.3, for each iteration of JNG, identified in the first column, we show the joining network generated and the action that was executed. The actions always refer to the iteration number as it uses the joining network of that iteration. The joining network expression is initialized once and from this point on, it can only be expanded. After each expansion, the resulting network can be pruned, enqueued or returned. In the first case, we generated a similar network in a previous iteration. In the second case, the network did not satisfy the cover criteria. In the last case, the network satisfies the cover criteria and is returned as an answer. It is worth noticing that, although more than one query joining network expression can be generated, only the first is returned as answer. Also, we shortened the real example to a few steps, for better visualization.

Given the attribute set A_L of L, JNG first extract the attribute `user_id` from A_L and stores its relation, `USER`, in the set of vertices of the tree J. This joining network is enqueued in $\mathcal{P}$. From the moment the queue is initialized, JNG starts generating joining network expressions by traversing the schema graph G_S. Since the `USER` is the first relation identified in the graph, JNG uses it as starting node. Then, JNG searches in a breadth-first fashion for relations that are adjacent to some relation in $J.V$. In the current state of the joining network (Iteration 0), only adjacent relations to `USER` are fetched. From this point on, we have two relations that can expand the current joining network: `REVISION` and `USER_RIGHTS`. JNG builds two joining network expressions (Iterations 2 and 3), each with a different adjacent relation. Then, these joining network expressions are enqueued and the process repeats. If an expanded joining network expression was already generated in a previous iteration, it can be pruned (Iteration 6) to avoid unnecessary processing. In the end, JNG generates a query joining network expression that covers L (Iteration 7).

4.3 SQL Composition

After successfully generating a query joining network expression, a corresponding `FROM` clause can be generated to compose the SQL query template that replaces the LESSQL query template. The query joining network expression is translated in a two-step process: (i) nodes in the query joining network expression are included in the `FROM` clause as database relations and (ii) edges in the query joining network expression are included as join conditions between the nodes above in the `FROM` clause.

(a) Query Template L

```
SELECT user_id, old_text
WHERE page_id = "X"
```

(b) Query Template S

```
SELECT user_id, old_text
FROM (((USER U
JOIN REVISION R ON U.user_id = R.rev_user)
JOIN TEXT T ON R.rev_id = T.old_id)
JOIN PAGE P ON R.rev_page = P.page_id)
WHERE page_id = "X"
```

Figure 4.4: A translation from LESSQL to SQL query template

Using the query joining network expression obtained in Figure 4.3 (Iteration 7) for L, we convert it to a `FROM` clause and generate an actual SQL query template as seen in Figure 4.4.

With the generated SQL query template, LESSQL sends the query to the DBMS, which returns the result to the user of the application, and to the LESSQL cache, where it is stored for later retrieval in case a similar LESSQL query template is received by the framework.

Chapter 5

Attribute Mapping Generation

In this chapter, we propose a strategy to detect changes between two versions of a database schema and automatically generate attribute mappings for these versions. In particular, we present *SchemaDiff*, a novel algorithm we propose to carry out this task. SchemaDiff is used both in the unsupervised and hybrid configurations of LESSQL, since these configurations entail the automatic generation of attribute mappings and, as a consequence, aim at reducing human intervention when the schema changes.

In the next sections, we show an overview of the SchemaDiff algorithm steps and introduce the definitions needed for better understanding the process of generating attribute mappings.

5.1 Overview

For generating proper SQL queries to a target schema, LESSQL relies on attribute mappings that define the changes made on attributes of the source schema. In the supervised configuration, the DBA is expected to specify a set of attribute mappings. If the number of changes happen to be significantly large, or if they are very frequent, manually providing attribute mappings becomes a burdening task.

To tackle this issue, we propose the SchemaDiff algorithm which tracks changes between two schema versions, as illustrated in Figure 5.1. SchemaDiff works at two levels: the relation level and the attribute level. At the relation level, SchemaDiff analyses the schema and associates missing relations with added relations leveraging Referential Integrity Constraints (RIC) defined for the schema versions. The missing and added relations are associated in a way that attribute mappings can only be generated between specific pairs of relations from source and target schema. That is, the mappings cannot be generated between a removed relation from the source schema and every relation in the target schema, but rather to a subset of relations as to not generate superfluous mappings. At the attribute level, missing attributes are associated with added attributes based on the associated relations, consequently generating a list of attribute mappings.

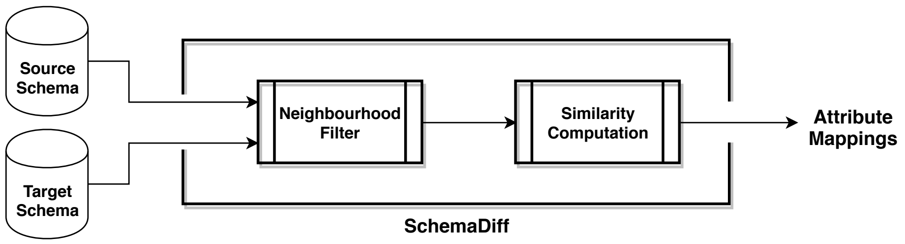

Figure 5.1: SchemaDiff Overview

The next section details how SchemaDiff filters relations that are more likely to yield suitable attribute mappings by introducing the concept of relation neighbourhood. Then, the last section details how SchemaDiff uses this concept when generating attribute mappings.

5.2 Neighbourhood Filter

The first step of SchemaDiff is to filter changes at the schema relation level. Specifically, this filter is used when relations from the source schema are missing in the target schema, affecting queries that referred to attributes located in these relations. To solve this issue, SchemaDiff associates relations from source and target schema by analysing the relations neighbourhood.

Definition 7. Let R be a relation from a schema version. Any relation N, such that there is a Referential Integrity Constraint (RIC) between R and N in the same schema, is called a *neighbour* of R. The *neighbourhood* of R, $\mathcal{N}(R)$, is the set of neighbours of R in a given schema.

In our work, the concept of relation neighbourhood is used to estimate the similarity between relations from distinct database schema versions.

We use the neighbourhood intersection size between two relations to filter out relations that are not likely to be candidate substitutes.

Definition 8. Let R_a be a relation from a source schema S_a that was removed from the target schema and let R_b be a relation that was added in the target schema S_b, such that the intersection $\mathcal{N}(R_a) \cap \mathcal{N}(R_b)$ between the neighbourhoods of R_a and R_b

is the largest one among all relations from S_b that were not present in S_a, that is, the added relations. The set of *candidate substitutes* for R_a in S_b, denoted as $\mathcal{C}(R_a, S_b)$, is given by $\mathcal{C}(R_a, S_b) = (\mathcal{N}(R_b) - \mathcal{N}(R_a)) \cup R_b$.

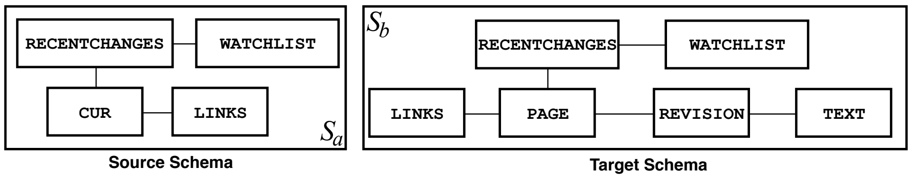

Figure 5.2: Neighbourhood Filter example

In Figure 5.2, we give a brief example that illustrates how the neighbourhood filter is applied in the process of generating mappings by using an excerpt of a real case scenario from the Wikipedia dataset. In the figure, the nodes represent relations and the edges represent RICs. On the top-right corner of every relation, is specified if the relation was added or removed. If there is not an indication, the relation did not change between schema versions. Let $S_a = \{cur, recentchanges, links, watchlist\}$ and $S_b = \{page, revision, text, recentchanges, links, watchlist\}$ be the source and target schemas, respectively. From S_a to S_b, the set $R_{S_a} = \{cur\}$ is the set of removed relations and the set $R_{S_b} = \{page, revision, text\}$ is the set of added relations. By Definition 8, the relation with largest common neighbourhood is the *page* relation, so $R_b = \{page\}$. Since there is a single removed relation, let $R_a = \{cur\}$, so $\mathcal{N}(R_a) = \{recentchanges, links, watchlist\}$ and $\mathcal{N}(R_b) = \{recentchanges, links, watchlist, revision, text\}$. Finally, applying Definition 8 results in the candidate substitutes $\mathcal{C}(R_a, S_b) = \{page, revision, text\}$ that will be used when computing mappings when tracking schema changes. Thus, the set $\{page, revision, text\}$ of relations represent the candidate substitutes to the source relation *cur*.

5.3 SchemaDiff

After computing candidate substitutes for relations in the target schema, the final step of SchemaDiff is computing the similarity between attributes of a source relation and their respective target candidate substitutes. From now on, we refer to attributes in the source schema as source attributes and attributes in the target schema as target attributes. The SchemaDiff algorithm is described in Algorithm 2.

Given a source attribute a that went missing in the target schema and its corresponding relation R_a in the source schema, for each candidate substitute relation

$R_i \in \mathcal{C}(R_a, S_b)$, SchemaDiff computes a similarity score between the source attribute a and each target attribute $b \in R_i$. If the similarity score is higher than a given threshold θ, the target attribute b is stored according to its score. After the computation of all similarities for a source attribute, SchemaDiff gets the attribute b with highest similarity score between a and b. Effectively, this pair is the attribute mapping for a given source attribute. Then, the process repeats for other source attributes. By default, SchemaDiff adopts *Jaccard with n-grams* as the similarity metric. Finally, the attribute mappings are forwarded to the JNG algorithm.

Algorithm 2: SchemaDiff

```
Input:
$S_a$: the source schema.
$S_b$: the target schema.
Output:
$AM$: attribute mappings.
1  begin
2      $AM \leftarrow \emptyset$;
3      foreach attribute $a \in S_a - S_b$ do
4          let $R_a$ from be the relation from $\mathcal{S}$ containing $a$
5          let $\mathcal{C}(R_a, S_b)$ be set of candidate substitutes for $R_a$ in $S_b$
6          $M \leftarrow \emptyset$;
7          foreach relation $R_i \in \mathcal{C}(R_a, S_b)$ do
8              foreach attribute $b \in R_i$ do
9                  $sim \leftarrow JaccardNGram(a, b)$;
10                 if $sim > \theta$ then
11                     $M[sim] \leftarrow M[sim] \cup \{b\}$;
12         let $V$ be the highest similarity value from $M$
13         let $P$ be a pair $\langle a, m_i \rangle$ where $m_i \in M[V]$
14         $AM \leftarrow AM \bigcup P$;
15     return $AM$
```

Chapter 6

Empirical Study

In this chapter, we report a set of experiments we performed with LESSQL on two open-source applications, *Wikipedia* and *WebERP*. The experiments consist of submitting LESSQL queries to our framework, which is responsible for translating LESSQL queries into regular SQL queries. As stated in Chapter 1, we propose three configurations for the LESSQL framework: supervised, unsupervised and hybrid configurations. Every configuration is evaluated separately, since each one presents different procedures for dealing with database schema changes. However, in all configurations, we evaluate the query generation *Success Rate* (SR), this is, given a query generated by the LESSQL framework, we evaluate if the generated query is identical to a golden standard query.

6.1 Experiment Setup

In this section, we explain the steps for preparing the dataset for evaluating the LESSQL framework in practice. In Table 6.1, we introduce the configuration of the machine used in all experiments.

CPU	Intel Core i7 7700 3.4GHz
RAM	32GB
OS	Ubuntu 18.04

Table 6.1: Machine configuration

The first dataset is the well-known Wikipedia dataset which contains 127 schema versions of the Wikipedia database, covering around 4 years of schema versions (2004-2007). Also, it includes a set of 80 query templates built from more a set of about one thousand most used queries in Wikipedia by time they were extracted. This dataset was provided by Curino et al. (2013) and was used in other studies of co-evolution of software and databases (Qiu et al., 2013; Vassiliadis, 2016). In Table 6.2 we present a summary of the schema versions in the Wikipedia dataset from 2004/01 to 2007/01. The versions spam a range of 37 months, from the first to the last. In some semesters, there were dozens of versions, e.g., in 2005/01.

Semester	**Versions**	**Total**
2004/01	0, 1, 2, 3, 4	5
2004/02	5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21	16
2005/01	22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51	29
2005/02	52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65	13
2006/01	66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80	14
2006/02	81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104	23
2007/01	105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127	22

Table 6.2: Summary of the schema versions in the dataset from 2004/01 to 2007/01

From the 80 query templates in the *Wikipedia* dataset, we discarded a few ones for which a join path could not be generated. They are: (1) Query templates in the form of `SELECT * FROM relation`. As queries of this type do not mention any attributes, its is not clear what must be included in the join path. There are just one case of such template in the dataset; (2) Query templates that use outer joins and self joins, which are application-specific and cannot be inferred automatically from the query attributes. The dataset includes 7 of such templates. Thus, from now on, we will consider the remaining 72 query templates. In a practical situation, the 8 removed templates can be submitted to LESSQL in their original form, since when our framework receives a full SQL query, it simply forwards it to the DBMS, without any processing.

The second dataset we use is WebERP, a web-based accounting and business management system, containing a total of 188 schema versions and covering around 7 years of development (2005 - 2012). From these schemas, we removed 76 schema versions that presented syntax errors or missing information. Finally, the remaining 112 schema versions were used for experimentation. Differently from the Wikipedia dataset, there is no log available from WebERP, so we have no queries for testing. In addition, due to lack of synchronization between the code versioning system and the schemas made available (Qiu et al., 2013), we could not extract queries from the code base. Thus, for the case of this dataset, we generated a synthetic query load by choosing a subset of the tables and attributes from the initial database schema that presented changes over the entire dataset. In particular, we selected 8 out of 90 relations for query generation, representing 9% of the schema relations. Although the database relations coverage percentage is not high, previous studies show that, typically, only a small subset of relations from a schema suffers significant changes (Vassiliadis, 2016). We did this to generate both the LESSQL query templates and the golden standard SQL query templates.

Overall, we selected a set of application databases to experiment with, mostly based on databases previously used in research (e.g. (Qiu et al., 2013; Curino et al., 2013)). Originally, our experiments would be executed over all of them. However, due to a lack of synchronization between the provided database schemas and the original source code repository, we were unable to manually generate, with confidence, the set of attribute mappings and schema changes that were applied to the set of schemas. From all these datasets, WebERP provided more understandable schema changes, which turn out to be easy for detecting them and generating the attribute mappings to use on experimentation.

In our discussions, we use the term *changes* to refer to changes in the schema versions that affect the queries templates we use, and we disregard any other kind of changes. In the Wikipedia dataset, notice that although all versions from version 17 on have changes with respect to the initial schema version 0, only 5 versions have changes with respect to the immediately previous version. These are versions 17, 20, 21, 24, 41, 42 and 45. In the WebERP dataset, from version 11 on we have changes with respect to the initial schema version, but only versions 11, 36, 71 and 72 present changes with respect to the immediately previous version. In both datasets, those versions that present changes with respect to the immediately previous version are called *major* versions.

To give an idea of the changes in the schema along the time, we show in Figure 6.1 and Figure 6.2, the types of attribute changes observed in the major versions with respect to the initial schema (version 0), that is, attributes that were removed (Dropped), attributes that were moved to another relation (Relocated), attributes that were renamed (Renamed) and attributes that were both renamed and relocated (Ren&Rel). The numbers inside the bars indicate the number of occurrences of each type of change. In the Wikipedia dataset, when comparing version 21 with the initial schema version we can see that 2 attributes were renamed, 5 were relocated to another table keeping their original name, 22 where were relocated to another table and their name changed and 2 attributes where dropped from the schema. In the WebERP dataset, we observed a more homogeneous distribution in what refers to the number

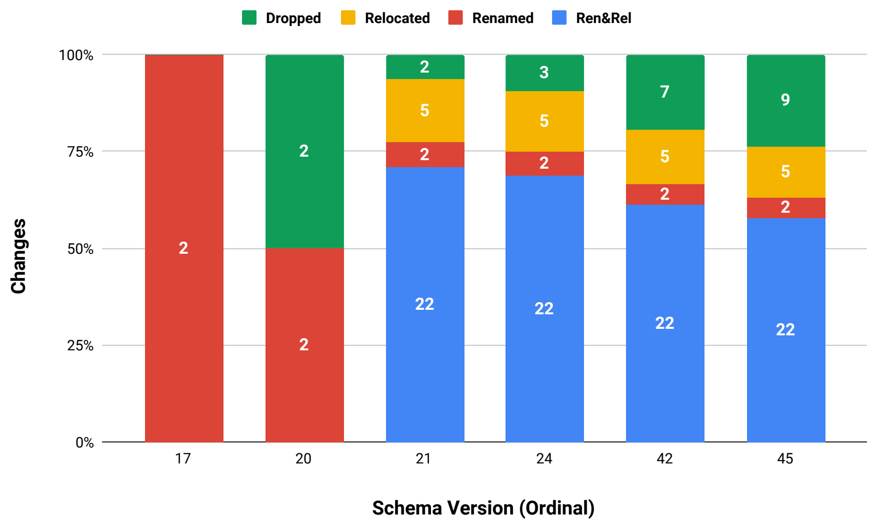

Figure 6.1: Types of attribute changes observed in the major versions of Wikipedia.

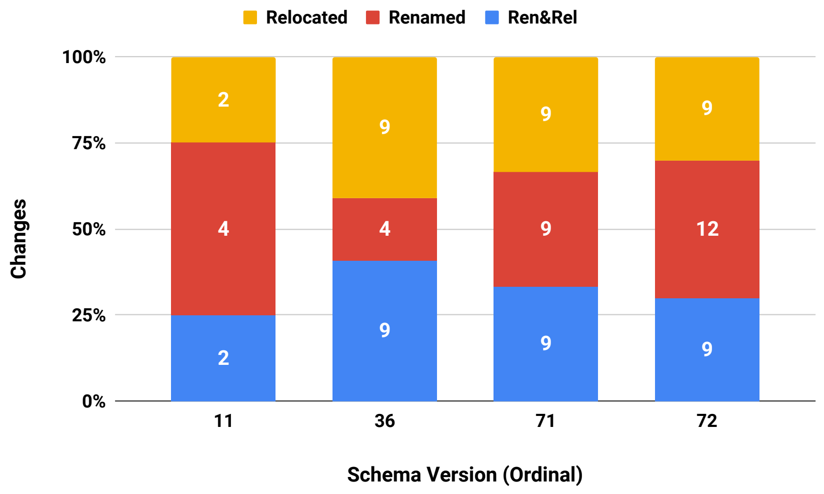

Figure 6.2: Types of attribute changes observed in the major versions of WebERP.

of changes according to the predefined categories. For instance, developers applied a major change to the WebERP database core relations in version 36. This change aimed at refactoring the software module responsible for tracking client orders along with the items and requirements registered for each order. Also, no dropped category was identified in the manual verification for this dataset.

Strictly for the Wikipedia dataset, which presents a log of queries extracted from the application production environment, several queries are generated from each of the

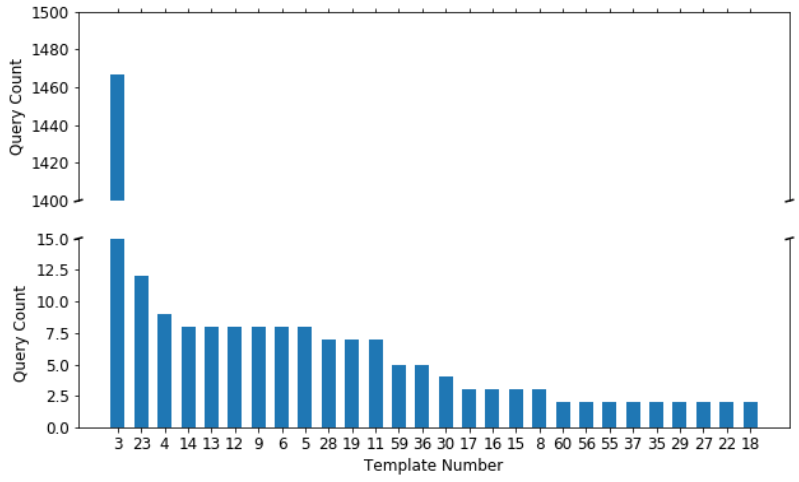

Figure 6.3: Number of queries per each query template.

query templates. In Figure 6.5 we present the number of queries derived from each of the 28 templates that generates more than one query. The queries were obtained from a log of the queries issued by Wikipedia applications over an instance of the initial version 0 of the Wikipedia database schema.

Overall, 38% of the templates generate more than one query. Each template generates, on average, 82 queries. In fact, there is one single query template, number 3, that generates 1467 queries, that is, more than 88% of the queries. Thus, failing in processing a single template due to schema changes, may have a huge impact on the system's operation.

Recall from Chapter 4 that the generation of SQL query templates from LESSQL query templates uses the JNG algorithm, and that this algorithm requires the PF/FK constraints of the database schema. As the original *Wikipedia* dataset does not include such constrains explicitly, we had to manually build them. This process is straightforward and we carried it out by looking at the joins in the queries from the dataset. On the other hand, the WebERP dataset included all PK/FK constraints in the database schema versions.

To form the final set of LESSQL query templates used in the experiments, we took the remaining original 72 query templates from the Wikipedia dataset and the generated 80 query templates from WebERP and removed their FROM clauses and explicit join conditions.

Finally, notice from Figure 6.1, that some attributes were simply removed in some versions of the Wikipedia dataset. This means that not all 72 query templates are *viable* in every schema version, that is not all initial query templates can generate a query for every version. The graph in Figure 6.4 shows the number of viable templates per Wikipedia schema version. From the set of generated queries for WebERP, all query templates could be executed along all schema versions.

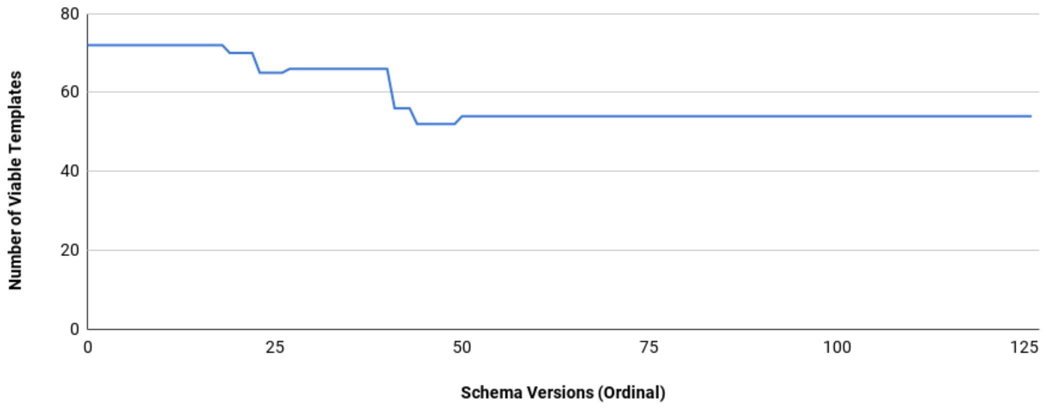

Figure 6.4: Number of viable query templates per schema versions.

6.2 Overall Results

In this section, we compare the impact of using LESSQL and original query templates from version 0. We call *Success Rate* (*SR*) in a schema version v, the fraction of query templates whose derived queries can be successfully executed in this version. In the case of the original SQL query templates, SR simply indicates the fraction of templates whose attributes and relations match the schema version v. In the case of LESSQL, SR indicates the fraction of LESSQL from which our framework was able to generate a SQL template whose attributes and relations match the schema version v.

Recall from Figure 6.4 that in newer schema version there are less than 72 valid templates for Wikipedia. Thus, for the case of LESSQL evaluation, SR metric also includes LESSQL query templates for which the framework correctly detected that there is no valid SQL query template in schema version v.

As Figure 6.5 shows, LESSQL was very effective. When using only the original SQL query templates, after a major schema change in schema version 21, SR was reduced to 0.23. In the case of supervised LESSQL, in the large majority of the cases, 95% of all generated SQL templates were correct, and in all cases SR was no less than 0.95. The unsupervised configuration was also effective, achieving an SR value of 0.55 after a major update. This is more than twice as expected in the original templates

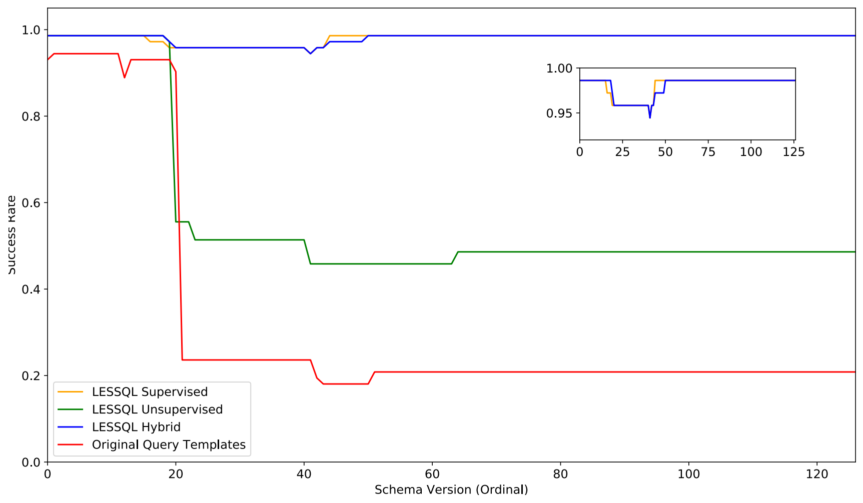

Figure 6.5: Success Rate per Wikipedia schema version obtained with the original SQL templates and with SQL templates generated by different LESSQL configurations.

from version 21 on. However, this configuration was also affected by the major schema change in version 21, in particular those of Ren&Rel type (Figure 6.1). The hybrid configurations consisted in using the unsupervised approach for all versions, except for the major version 21. Overall, this configuration achieved the same SR levels of the supervised configuration, but it required much less effort from the user. While the supervised configuration required a total of 3941 attribute mappings to be supplied across all version, the hybrid configuration only required 27 mappings to be supplied to handle a single major change. It is worth noticing that the whole set of 3941 attribute mappings contains only 38 distinct mappings. Still, the whole set would have to be manually supplied.

Also, the detailed window in Figure 6.5 shows a comparison between the supervised and the hybrid configurations. We compare the SR of our supervised and hybrid configurations, since they presented closer results. The overall difference presented by both SR is explained by the use of the SchemaDiff algorithm. The supervised configuration relies solely in the attribute mappings provided by the DBA, while the hybrid configuration mixes the unsupervised and supervised configurations. Using the unsupervised configuration means relying on the SchemaDiff algorithm to correctly assign the mappings between a pair of schemas. Although the overall mappings provided by the SchemaDiff algorithm are correct, the decrease in SR was due to an attribute ambiguity problem. That is, between versions 17 to 48, more than one relation contained an attribute with the same name. This impacts the generation of the SQL query since any of the attributes that share the same name satisfy the LESSQL query referring to that single attribute.

As illustrated in Figure 6.6, LESSQL showed a similar behavior if compared to the Wikipedia dataset experiment. In particular, we focus on the major change introduced by schema version 36 which reduced the query SR to 0.13. When adopting the LESSQL supervised configuration, above 95% of the generated SQL query templates were correctly generated, achieving a SR value of 0.96. In the case of the unsupervised configuration, the SR value achieved 0.91, representing a huge benefit over the original query template approach since it does not rely on human effort. The difference between unsupervised configuration in the WebERP dataset with respect to the unsupervised configuration in the Wikipedia dataset can be explained by the easier mapping generation faced by the SchemaDiff algorithm. In the hybrid configuration, the results obtained were similar to the results from the supervised configuration. In this case, we provided a set of 22 attribute mappings, comprised of 8 attribute mappings from the schema version 11 on and 14 attributes from the schema version 36 on. Overall, LESSQL was mainly affected by schema changes of the type Ren&Rel, similar to the Wikipedia dataset experiments. It is worth noticing that, in WebERP, all queries could be executed over all database schema versions.

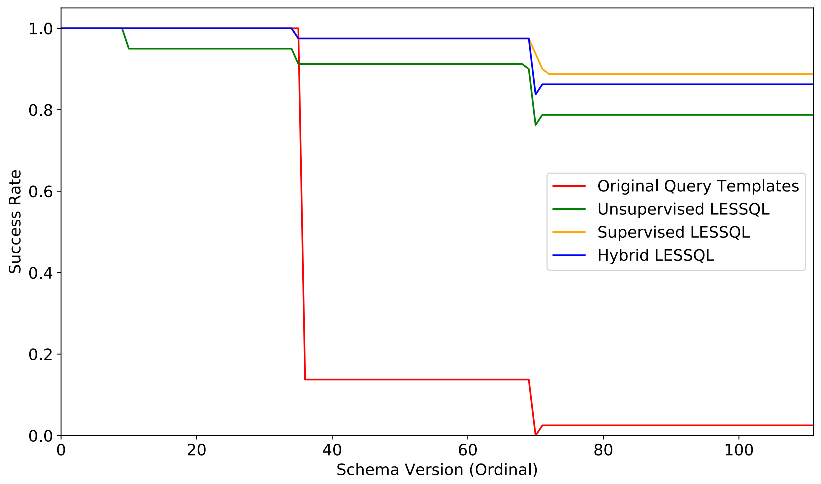

Figure 6.6: Success Rate per WebERP schema version obtained with the original SQL templates and with SQL templates generated by different LESSQL configurations.

6.3 Unsupervised LESSQL

This section we present a detailed evaluation of the unsupervised configuration, which requires more steps than the supervised configuration. In particular, the unsupervised LESSQL configuration relies on the SchemaDiff algorithm presented in Chapter 5. SchemaDiff generates mappings between attributes using our proposed neighbourhood filter and a similarity function. However, automatically finding attribute mappings between schemas is inherently difficult since there is not a pattern to follow when changes are applied to schema.

To evaluate SchemaDiff, we compute the Precision, Recall and F-measure as follows. Let S_i be the set of mappings from the attributes of the version 0 to the attributes of version i generated by SchemaDiff and M_i be the set of *correct* mappings from the attributes of the version 0 to the attributes of version i. The SchemaDiff precision and recall at version i are given by $P_i = |S_i \cap M_i|/|S_i|$ and $R_i = |S_i \cap M_i|/|M_i|$, respectively. Finally, the F-measure is given by $F_i = 2 \times (P_i \times R_i)/(P_i + R_i)$.

This experiment measured the effectiveness of the candidate attribute mappings found by our algorithm, that is, among the attributes retrieved from the target schema, if the correct attribute was identified. Formally, given a source version v, a target version vt, a set of attribute mappings and the golden standard for the target version, respectively, AM_{vt} and GS_{vt}, if a mapped attribute $x \in AM_{vt}$ is correctly mapped as explicit by GS_{vt}, the accuracy is increased. The next subsections show these metrics in both datasets, Wikipedia and WebERP.

Overall, for the Wikipedia dataset, the framework average precision was above 0.9 and the average recall was above 0.8, as seen in Figure 6.7. As there can be many candidate attribute mappings with a very low similarity score, recall from Chapter 5 that our SchemaDiff algorithm uses a top-k score data structure to store the k highest similarities calculated per attribute removed. In this experiments, we set the value of k to 3 by empirically testing values that ranged from 1 to 5, analyzing the overall precision.

In the WebERP dataset, the framework average precision was 0.61 while the average recall was above 0.65, as shown in Figure 6.8. The precision metric was mainly affected by the number of false positives raised by our similarity function inside this dataset. In particular, attributes that are not related between two schema versions present a similarity value above our predefined threshold, thus, they presented a negative impact in the precision metric. However, the overall result of our framework is not affected since the high scored similarities are usually the correct ones. This is shown in the MRR experiment below.

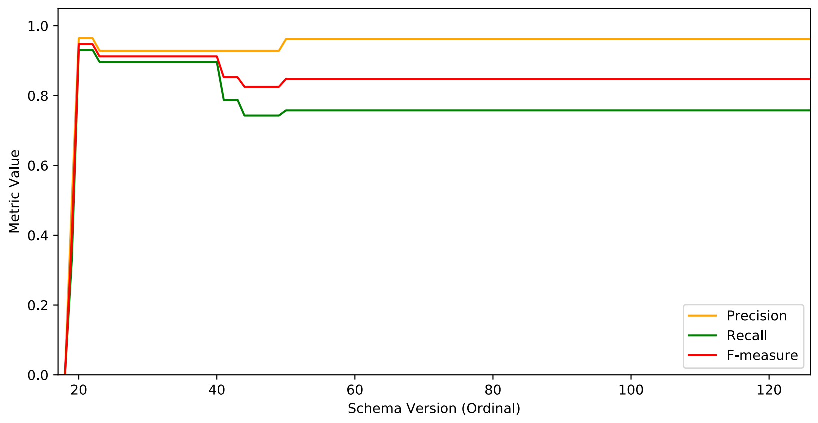

Figure 6.7: SchemaDiff evaluation across all Wikipedia schema versions.

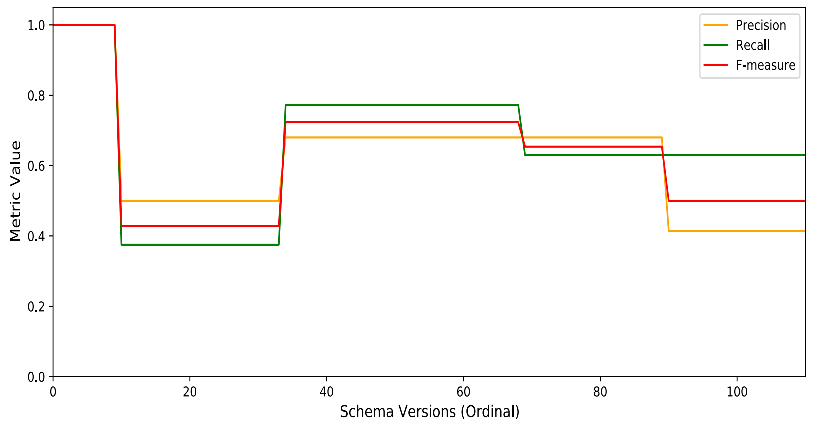

Figure 6.8: SchemaDiff evalution across all WebERP schema versions.

The candidate attribute mappings precision results are shown in Figures 6.9 and 6.10 for the Wikipedia dataset and Figure 6.11 for the WebERP dataset. Notice that the plot does not contain all of the tested values since from value 3 on, the results are equal. This means that our SchemaDiff algorithm finds the correct attribute at most in the second position of the candidate attribute mappings rank. Also, we tested the precision using a slightly different approach. In Figure 6.9, we consider that, when an attribute from the source schema is mapped to NULL and, according to the golden standard, it should be mapped to NULL, the overall precision is increased. On the other hand, Figures 6.10 and 6.11 do not consider these mappings, so the overall precision is decreased. Different from Wikipedia, the experiments on WebERP did not present dropped attributes and, for this reason, we only present the chart containing precision without NULL mappings.

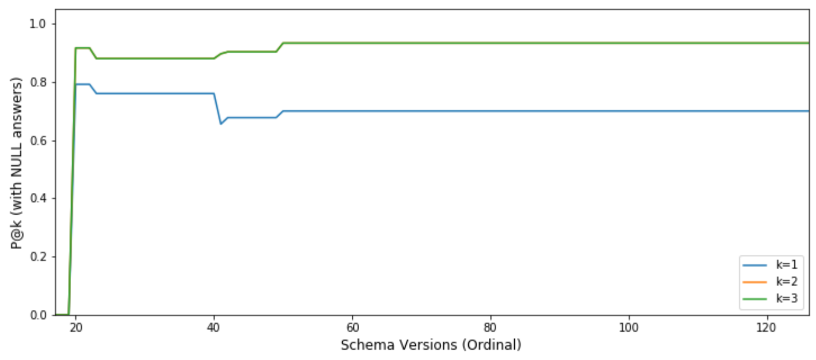

Figure 6.9: SchemaDiff Precision on Wikipedia, considering NULL mappings.

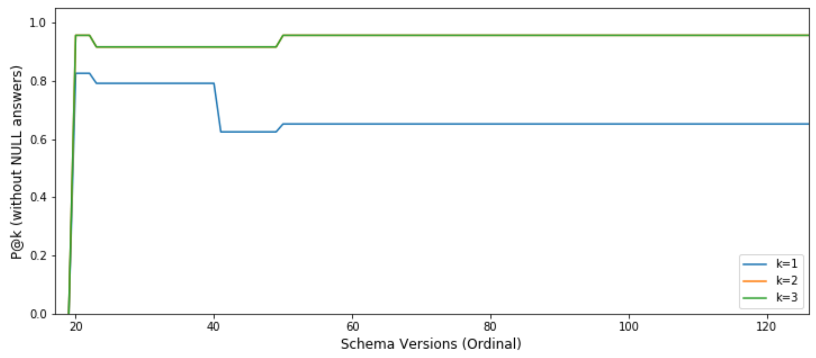

Figure 6.10: SchemaDiff Precision on Wikipedia, not considering NULL mappings.

As said in the last section, the unsupervised LESSQL query generation SR after a major update reached 0.57 while in the original scenario the expected SR would be 0.2. This increase, approximately triple of the expected value in the real case, shows us that application maintenance impact can be reduced with the help of our framework. In the WebERP scenario, the unsupervised configuration presented a SR value of 0.91, which represents a huge impact in development scenarios since it does not rely on manual effort from developers, thus, reducing source code maintenance. This configuration, besides eliminating the need of supervision from the DBA, will present an overall quality lower than the other two configurations because of the mapping generation process. This happens when there is a schema update with a large quantity of mappings and the system must look for attribute mappings among many possible attribute transformations.

To further investigate the efficiency of our SchemaDiff algorithm, we illustrate in Figures 6.12 and 6.13 the attributes mappings MRR. This experiment shows the

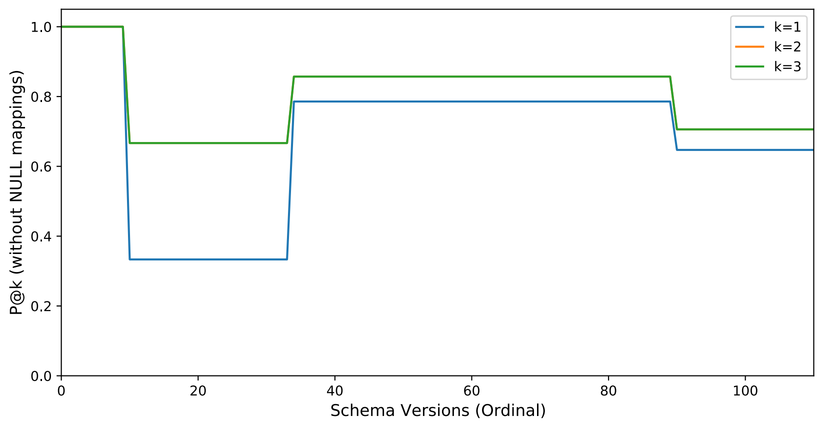

Figure 6.11: SchemaDiff Precision on WebERP, not considering NULL mappings.

position of the rank in which the attributes from the target schema are located according to their similarity values. Similarly to the precision experiment, this experiment was executed considering the first three rank positions and calculated the MRR with respect to these three positions. That is, if k is 1, then our precision only retrieves attributes that are located in the 1^{st} position, and so on. In the Wikipedia dataset, the computed MRR value stays above 0.5, meaning that the correct attribute in the target schema is at most in the 2^{nd} position of the ranking. In the WebERP dataset, the average MRR value, considering from the 2^{nd} position, was above 0.8. The MRR metric is used to provide us with insights on the effectiveness of the LESSQL attribute mappings generation process. That is, given that the correct attribute mapping was not found in the first position, in which position it could be found on the mappings ranking.

When computing the attribute mappings MRR, we removed mappings in which the attribute from the source schema split into multiple attributes in the target schema. The MRR metric computes a value according to the position of the first relevant answer. However, in this scenario, the relevant answer involves many itens in the same ranking position. Thus, it is not possible to evaluate these cases. We use those mappings only when computing the precision metric. In Table 6.3, we show the number of occurrences of attributes that split into multiple attributes in the target schema and also the number of mappings that represent these cases in both datasets.

For instance, in the WebERP database, schema version 11 presented one occurrence of an attribute that was split into more than one attribute in the target schema. Specifically, the attribute `taxauthority` split into `taxauthority`, `taxgroupid` and `taxprovinceid`. From these, the attribute `taxauthority` in the target schema was located in two relations so, in practice, two mappings were affected.

Dataset	**Schema version**	**Occurrences**	**No. of mappings**	**Total**
Wikipedia	21	1	2	2
WebERP	11	1	4	11
	36	3	7	

Table 6.3: Number of mappings removed from the MRR computation

Keeping all the queries updated according to the schema requires the schema changes to not be significantly large. That is, the more significant changes to a database schema are, the more difficult is the problem to keep the application functionalities running in the event of schema changes.

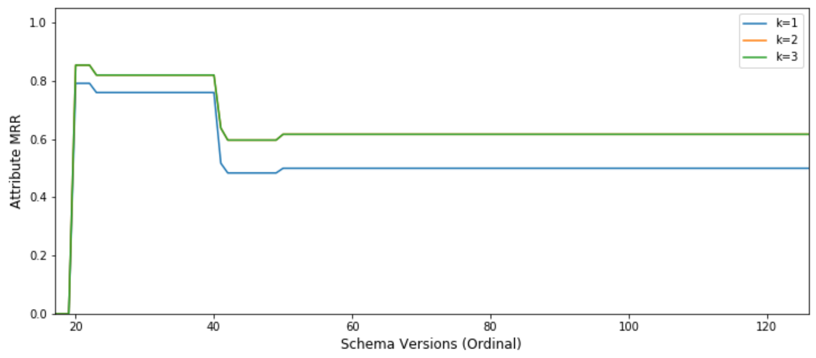

Figure 6.12: Wikipedia Attribute MRR over all schema versions.

Although our SchemaDiff algorithm deals with a large number of candidate mappings from a pair of schemas, resulting in a precision and recall decrease, we will show that our framework is capable of generating the correct query even if it is not ranked with the highest score. Similar to the attribute mappings MRR, the MRR metric for the query generation process indicates whether LESSQL generated the correct SQL query in a position that is not the first. Figure 6.16 shows that, after the major update in Wikipedia version 21, on average, the correct query is found between the 2^{nd} and the 3^{rd} position. The same is valid for the WebERP dataset, where the correct query can be found at most at the 3^{rd} position with an average MRR of 0.72.

Over time, according to the schema changes, more candidate attribute mappings may be identified and the query MRR may be affected. This scenario is depicted in Figure 6.18 between Wikipedia schema versions 20 and 21. Version 21 presented more mapping alternatives than the previous versions due to relation additions and removals

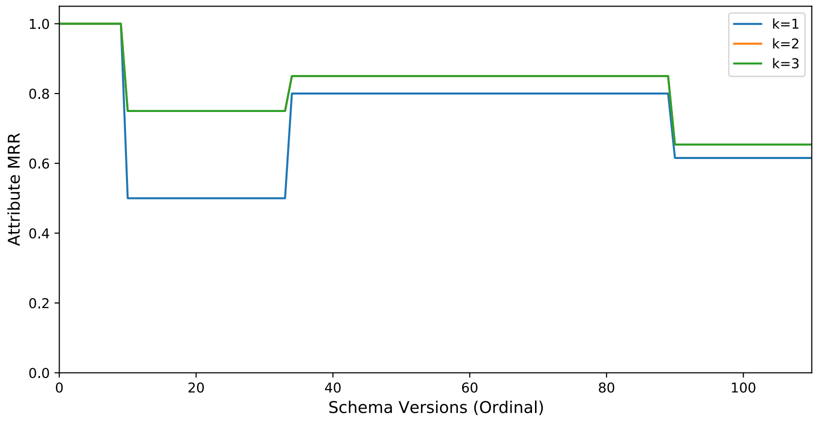

Figure 6.13: WebERP Attribute MRR over all schema versions.

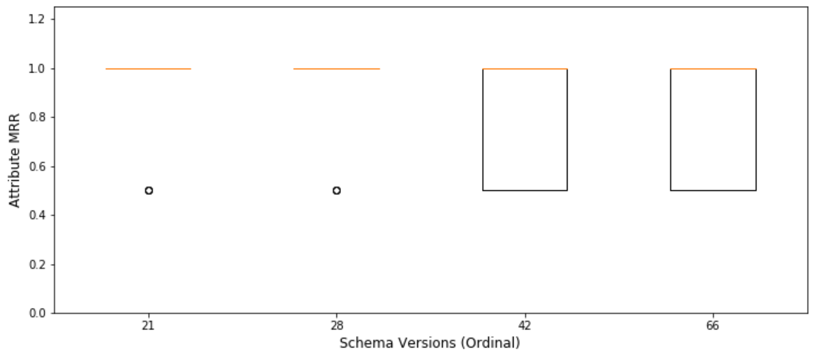

Figure 6.14: Wikipedia Attribute MRR boxplot in four different versions.

in the database schema, since it was classified as a major version. This reflected directly in the query MRR inside version 21.

6.4 Performance Issues

The adoption of LESSQL introduces new steps in the application and DBMS environment. In the event of schema changes, LESSQL will trigger the SchemaDiff algorithm, when using unsupervised or hybrid configurations, computing the mappings between the source and the target schema, then forwarding it to the JNG algorithm. In our experiments, the longest time was 0.06 seconds.

As for the query generation process, when a LESSQL query is issued against the framework it will trigger a SQL template generation through JNG. In some cases, this

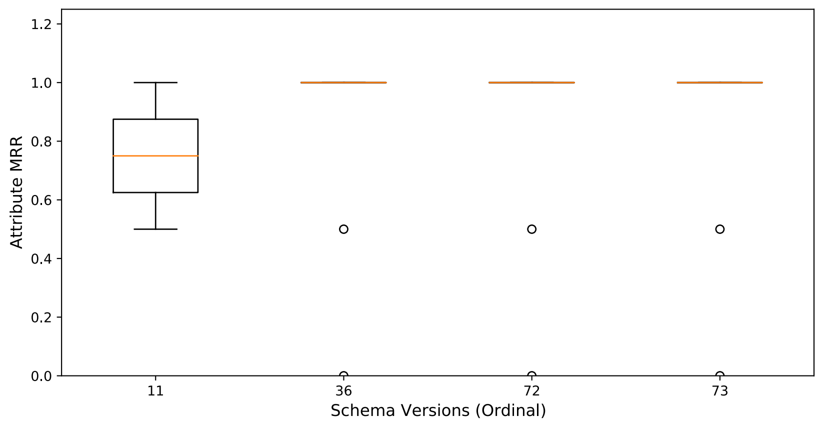

Figure 6.15: WebERP Attribute MRR boxplot in four different versions.

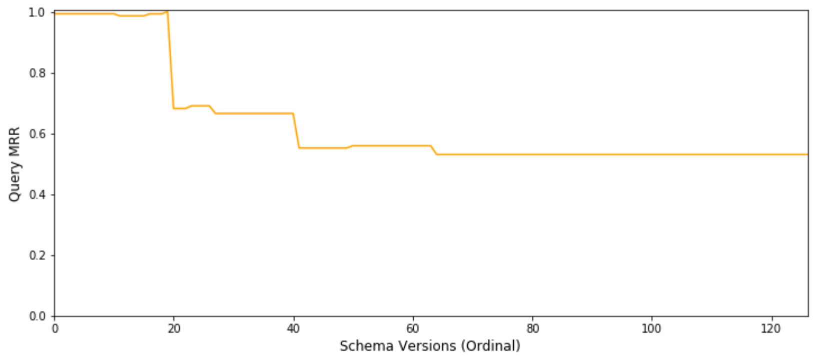

Figure 6.16: Wikipedia Query MRR.

process can introduce a significant time overhead when the query size is large and many candidate attribute mappings are generated. The most expensive SQL generation time seen in the Wikipedia dataset experiments was of 39 seconds, for a query referencing 9 attributes. In the WebERP dataset, the most expensive query took 8 seconds and referenced 7 attributes. An average of the query generation time per schema version is depicted in Figures 6.20 and 6.21, for Wikipedia and WebERP, respectively.

For a better comprehension of the overhead introduced by the query generation process, we selected four schema versions that presented a significant time increase as considered to the version right before, this scenario is shown in Figures 6.22 and 6.23.

As can be seen in Figure 6.22, the majority of SQL query templates are generated between 0.01-1.0 second in both datasets. Overall, LESSQL introduces little overhead for querying the database. However, some queries can be burdensome to generate

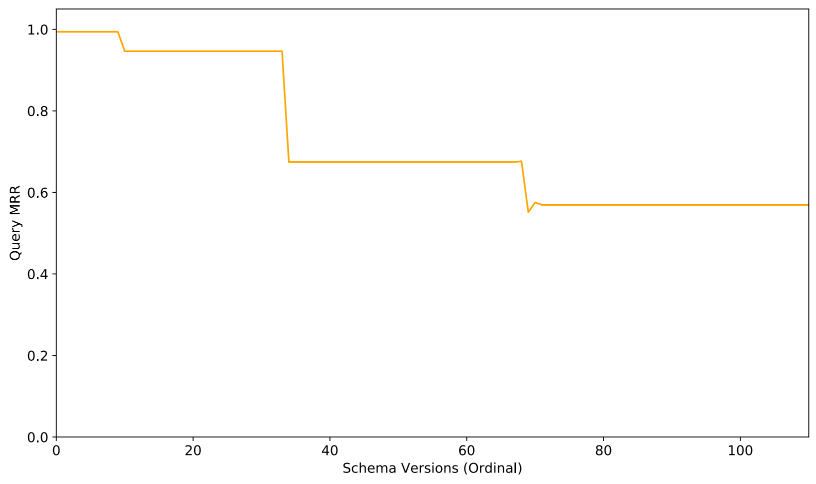

Figure 6.17: WebERP Query MRR.

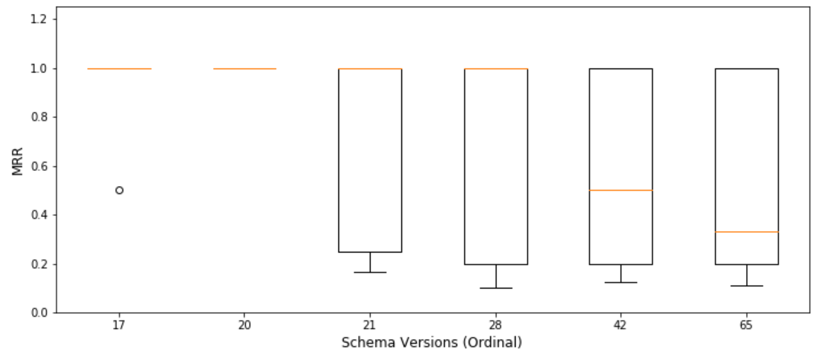

Figure 6.18: Wikipedia Query MRR boxplot in major versions.

and, as a consequence, they may interfere in the application response time. For these cases, our framework provides an embedded cache feature for storing already generated queries. In particular, when a LESSQL query is issued, our framework generates an appropriate SQL query template and stores it, waiting for another LESSQL query with identical structure to be executed. This makes the generation time negligible after the generation of a SQL query template for a given LESSQL query and encourages the adoption of LESSQL since it has little to none impact in the application production environment.

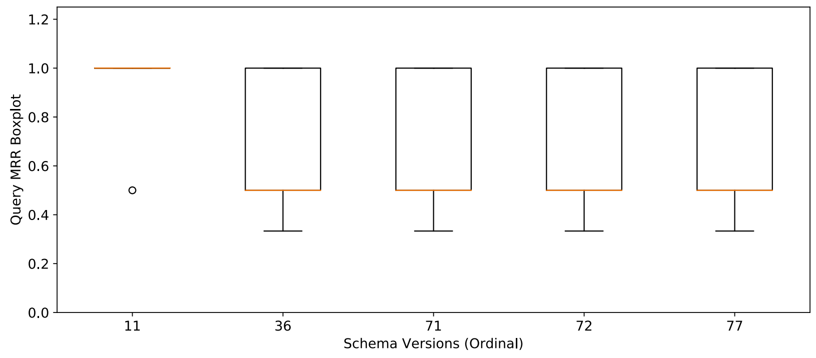

Figure 6.19: WebERP Query MRR boxplot in major versions.

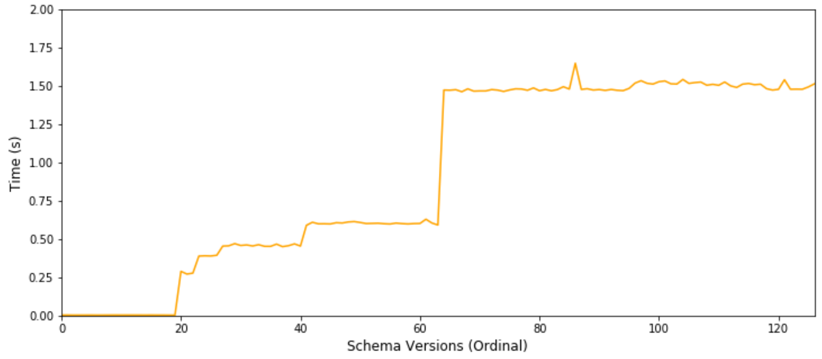

Figure 6.20: Average query generation time per Wikipedia schema version.

6.5 Limitations and Threats to Validity

Although we have used only two datasets in our evaluation, we argue that these datasets are representative for all the problems we tackle. The first dataset, Wikipedia, is a real and popular system among millions of users. In addition, as described in its web page[1], the project went through numerous schema and code changes through time and endured an exemplary case of database decay. Moreover, as the dataset covers 4 years of evolution, it represents a high diversity of schema refactoring types (Figure 6.1). For these reasons, this dataset has been previously used in other studies on co-evolution of software and databases, e.g., (Qiu et al., 2013; Vassiliadis, 2016). Furthermore, WebERP was also used in previous studies and present similar evolution scenarios to Wikipedia, emphasizing even more the necessity of adopting a framework that minimizes the source code maintenance when changes are applied to the schema.

[1]https://www.mediawiki.org/wiki/Manual:MediaWiki_architecture

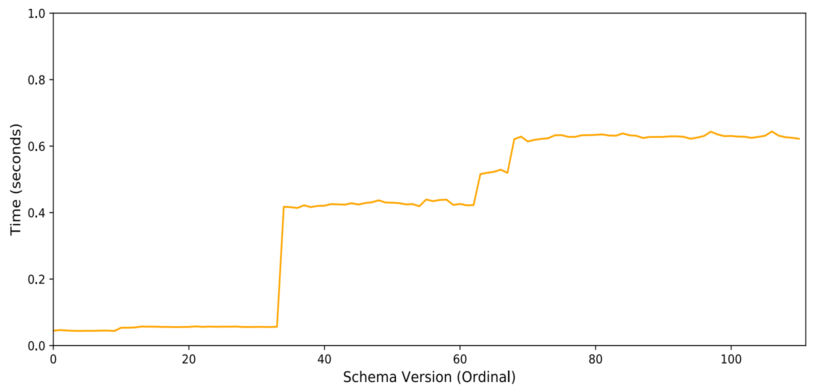

Figure 6.21: Average query generation time per WebERP schema version.

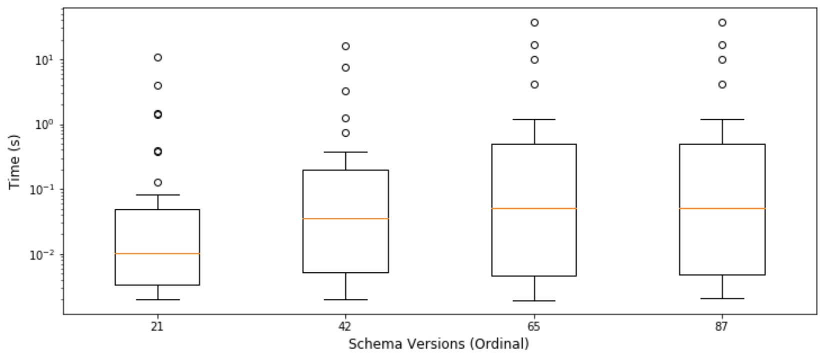

Figure 6.22: Query generation time in four different versions.

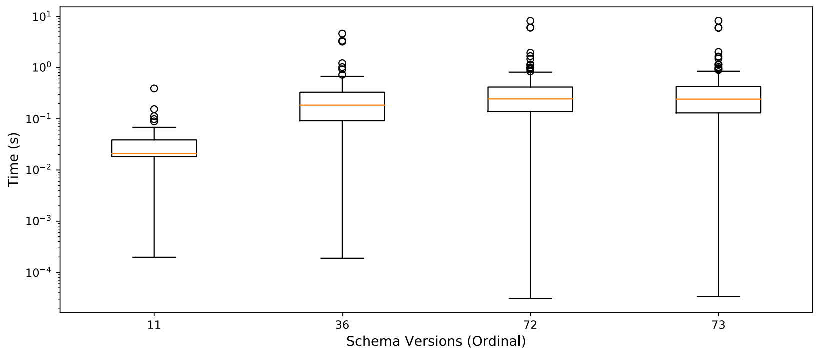

Figure 6.23: Query generation time in four different versions.

We were not able to explicitly measure the extent of remaining code modifications, *i.e.*, those caused by schema changes not successfully handled by LESSQL. In order to tackle this issue, we computed how often LESSQL was not successful, which ended up being very rare (less than 5% of the cases) with both supervised and hybrid configurations for the Wikipedia dataset. In WebERP, LESSQL was more affected due to the number of cases where the attribute split into more than one alternative. Nonetheless, we emphasize that the amount of code modifications was lower if compared with the amount of code modifications in case one had to change the original schema template over time. The same reasoning applies to the unsupervised configuration when it was exposed to simple structural schema changes. Other threats and corresponding mitigations are discussed in depth in our supplementary material.

Chapter 7

Conclusions and Future Work

In this chapter, we review our main contributions, discuss the conclusions we obtained in our experiments and outline the improvements for future work.

7.1 Conclusions and Main Contributions

In this work, we presented an alternative approach for handling database schema changes in development scenarios that choose to adopt Continuous Deployment, named LESSQL, which is a framework-based approach that aims at minimizing application source code maintenance in the queries affected by database schema changes. LESSQL is based on queries that are less dependent on the database schema and do not contain the `FROM` clause. This enables LESSQL to generate actual SQL queries that can be issued against the updated schema.

Overall, LESSQL supports three different configurations: (i) supervised - where the DBA supplies attribute mappings; (ii) unsupervised - where the attribute mappings are automatically generated by our novel algorithm, SchemaDiff; and (iii) hybrid - where the DBA provides a small and critic set of attribute mappings in face of major updates in the database schema.

In all configurations, we use the JNG algorithm, which is based on Keyword-Search Systems for Relational Databases, mainly from Oliveira et al. (2018). JNG is an algorithm we proposed that is responsible for generating joining network expressions, which are translated into FROM clauses that are used to compose the actual SQL query to be executed in the RDBMS.

In the unsupervised and hybrid configurations, we adopt the *SchemaDiff* algorithm. *SchemaDiff* is also an algorithm we proposed to derive the difference between two schema versions. This difference is used in the generation of *attribute mappings*, which represent the changes that represent the transformation of the source schema in the target schema. This representation is used when composing the SQL query to update the references to attributes that were renamed and relocated to different relations when the schema changes were applied.

We conducted our experiments using two datasets: the well-known Wikipedia dataset and the WebERP dataset. With the obtained results, we indicate that our approach can cope with the database decay problem by improving queries' stability in the presence of structural changes to the database schema. As a result, our approach favors faster CD cycles and also keeps application services operational. Moreover, LESSQL would help to better streamline changes that span across multiple teams, which otherwise significantly harm the efficient generation of builds. In fact, as demonstrated by our experiment, the number of affected queries can be huge depending on the kind of schema refactoring. Structural changes to the database schema could be the most costly and LESSQL provides a way to gracefully handle almost all of them.

Additionally, LESSQL is a concrete approach to handle the database decay problem stated by Stonebraker et al. (2016). The decay is introduced by schema changes that result in a poor database schema design. Since our approach keeps the queries updated according to the changes applied to the schema, we expect it to encourage DBAs to keep up with good practices with respect to database schema normalization.

Finally, as a result of this research we had a paper accepted for presentation in the 27th *International Conference on Software Analysis, Evolution and Reengineering* (SANER) in 2020 (Afonso et al., 2020). This emphasizes even more the interest of researchers in tackling the problem of database schema changes when applied in development environments that would otherwise be affected by the applied schema changes.

7.2 Future Work

The promising results we obtained so far with LESSQL raised several ideas for future work. These ideas are enumerated and discussed below.

7.2.1 Expand LESSQL support for CRUD statements

The LESSQL approach avoids rewriting queries by allowing a flexible query syntax. We will leverage LESSQL queries and expand them to work with other CRUD statements such as INSERT, UPDATE and DELETE. This will allow our method to reduce impact of structural database schema changes in application source code. After implementation, we will evaluate the impact of our framework in open-source/proprietary datasets as to ensure the generality of our method with respect to different development scenarios.

7.2.2 Dealing with Database Instance Migration

Since LESSQL did not provide a database instance migration feature, we will design a method that is capable of assisting DBAs when migrating database instances. This method will allow data to be migrated between schema versions in the background while the application service is running, thus, not impacting application users nor functionalities. We will evaluate this method based on its efficiency in the data migration process, that is, its capability of generating low overhead when migrating data across database instances, keeping the application services operational with minimal interruption. Also, we will classify the overhead according to the database schema change applied, generating a best practices report on the use of schema change operations.

7.2.3 Coupling a Unified Framework into an ORM

We will unify both previous developed methods to implement a full-fledged framework. That is, we will implement a framework that can handle both problems, source code maintenance and database instance migration, simultaneously. Also, to widen the use of our solution, we will implement it alongside an Object-Relational Mapper (ORM). ORMs are widely adopted in the industry and are preferred as a method of accessing database resources. To the best of our knowledge, an approach such as this is not present in the literature.

Bibliography

(2019). Facebook schema change. `https://github.com/facebookincubator/OnlineSchemaChange`. Accessed: 2019-05-29.

(2019). Soundcloud large hadron migrator. `https://github.com/soundcloud/lhm`. Accessed: 2019-05-29.

Afonso, A., da Silva, A., Conte, T., Martins, P., Cavalcanti, J., and Garcia, A. (2020). Lessql: Dealing with database schema changes in continuous deployment. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pgs. 138–148. IEEE.

Bernstein, P. A. and Melnik, S. (2007). Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pgs. 1–12. ACM.

Berquist, J. and Gunson, G. (2018). Mysql infrastructure testing automation at github. In *Large Installation System Administration Conference, LISA 2018*, pgs. –.

Caruccio, L., Polese, G., and Tortora, G. (2016). Synchronization of queries and views upon schema evolutions: A survey. *ACM Trans. Database Syst.*, 41(2):9:1–9:41.

Curino, C., Moon, H. J., Deutsch, A., and Zaniolo, C. (2013). Automating the database schema evolution process. *The VLDB Journal—The International Journal on Very Large Data Bases*, 22(1):73–98.

Curino, C. A., Moon, H. J., and Zaniolo, C. (2008). Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772.

De Jong, M., van Deursen, A., and Cleve, A. (2017). Zero-downtime sql database schema evolution for continuous deployment. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pgs. 143–152. IEEE.

Delplanque, J., Etien, A., Anquetil, N., and Auverlot, O. (2018). Relational database schema evolution: An industrial case study. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pgs. 635–644. IEEE.

Hintikka, J. and Bachman, J. (1991). *What If...?: Toward Excellence in Reasoning.* McGraw-Hill Humanities, Social Sciences & World Languages.

Hristidis, V. and Papakonstantinou, Y. (2002). Discover: Keyword search in relational databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pgs. 670–681. Elsevier.

Laukkanen, E., Itkonen, J., and Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, 82:55–79.

Oliveira, P., da Silva, A., de Moura, E., and Rodrigues, R. (2018). Match-based candidate network generation for keyword queries over relational databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pgs. 1344–1347. IEEE.

Polese, G. and Vacca, M. (2009a). A dialogue-based model for the query synchronization problem. In *2009 IEEE 5th International Conference on Intelligent Computer Communication and Processing*, pgs. 67–70. IEEE.

Polese, G. and Vacca, M. (2009b). Notes on view synchronization using default logic. In *SEBD*, pgs. 253–260.

Qiu, D., Li, B., and Su, Z. (2013). An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pgs. 125–135. ACM.

Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350.

Rossi, C., Shibley, E., Su, S., Beck, K., Savor, T., and Stumm, M. (2016). Continuous deployment of mobile software at facebook (showcase). In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pgs. 12–23. ACM.

Saur, K., Dumitraş, T., and Hicks, M. (2016). `https://www.facebook.com/notes/mysql-at-facebook/online-schemachange-for-`

`mysql/430801045932`. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pgs. 166–176.

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. (2016). Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pgs. 21–30. IEEE.

Stonebraker, M., Deng, D., and Brodie, M. L. (2016). Database decay and how to avoid it. In *2016 IEEE International Conference on Big Data (Big Data)*, pgs. 7–16. IEEE.

Vassiliadis, P. (2016). Schema evolution for relational databases. In *DATA 2016 - Proceedings of 5th International Conference on Data Management Technologies and Applications, Lisbon, Portugal, 24-26 July, 2016.*, pg. 5.

Wang, Y., Dong, J., Shah, R., and Dillig, I. (2019). Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pgs. 286–300. ACM.