TÉCNICO
LISBOA

Evo DB - Bringing Evolutionary Database Design for Schema Editing Tools with a Version Control System

Gonçalo Pereira da Costa

Thesis to obtain the Master of Science Degree in

Engenharia Informática e de Computadores

Supervisor: Prof. Paulo Jorge Fernandes Carreira

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Paulo Jorge Fernandes Carreira
Member of the Committee: Prof. Francisco Afonso Severino Regateiro

June 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Prof. Paulo Carreira for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

The main focus of Schema Editing Tools (SETs) is to facilitate database development and management. They provide an easy-to-use GUI with which Database Administrators (DBAs) and developers can manipulate the database schema and data easily and quickly, without even needing to know what SQL is issued. However, the use of these tools blocks the use of more evolutionary practices (more iterative and incremental practices) in the database development process. These tools do not allow the developer to save the statements the SET issued “under the hood” in a consistent and organized way. That makes it very difficult for developers that use SETs to do database versioning and transmit the executed statements to other environments like other developers’ sandboxes or Quality Assurance (QA) or production. This difficulty is also an obstacle to the integration of the database development process into Continuous Integration (CI). Evo DB has a connection, Evo JDBC, that is passed to a SET that allows Evo DB to capture all the SQL operations successfully done by the SET, making them available in the Evo Version Control System (VCS). Evo VCS lets the developer generate and manage migrations based on the SQL issued by the SET and perform coherent database deployments.

Keywords

Databases, Database Migration, Database Versioning, Evolutionary Database Design, Continuous Integration

Resumo

O foco principal dos Schema Editing Tools (SETs) é facilitar o desenvolvimento e a gestão de bases de dados. Eles oferecem um GUI fácil de usar com o qual os Database Administrators (DBAs) e developers podem manipular o schema e os dados da base de dados facilmente e rapidamente, sem sequer precisar de saber qual foi o SQL emitido. Contudo, o uso destas ferramentas bloqueiam o uso de práticas mais evolutivas (práticas mais iterativas e incrementais) no processo de desenvolvimento de base de dados. Estas ferramentas não permitem que o developer guarde as instruções que foram emitidas “por debaixo do capô” numa forma consistente e organizada. Isso faz com que seja difícil para os developers que usam SETs fazer versionamento da base de dados e transmitir as operações executadas para outros ambientes como as sandboxes de outros developers, Quality Assurance (QA) ou produção. Esta dificuldade é também um obstáculo à integração do processo de desenvolvimento de base de dados em Continuous Integration (CI). Evo DB tem uma conexão, Evo JDBC, que é passada para um SET que permite ao Evo DB capturar todas as operações SQL que foram feitas pelo SET com sucesso, tornando-as disponíveis no Evo Version Control System (VCS). Evo VCS permite ao developer gerar e gerir migrações baseadas no SQL emitido pelo SET e realizar deployments coerentes da base de dados.

Palavras Chave

Base de Dados, Migração de Bases de Dados, Versionamento de Bases de Dados, Design de Base de Dados Evolutivo, Integração Contínua

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
BDUF	Big Design Up Front
CDBI	Continuous Database Integration
CI	Continuous Integration
CLI	Command-line Client
CMS	Content Management System
DBA	Database Administrator
DDL	Data Definition Language
DML	Data Manipulation Language
ERD	Entity Relationship Diagram
GUI	Graphical User Interface
JDBC	Java Database Connection
ORM	Object-relational Mapping
QA	Quality Assurance
RDBMS	Relational Database Management System
SCM	Software Configuration Management
SET	Schema Editing Tool
VCS	Version Control System

1 Introduction

Contents

Several Database Administrators (DBAs) and developers use Schema Editing Tools (SETs) or Database IDEs to manipulate databases. These tools allow them to do simple and complex operations easily and rapidly through a Graphical User Interface (GUI) without writing any SQL. While developing, this can be a quick solution to create, alter or drop tables, columns or views. However, when it is time to deploy the changes into another database, problems arise. It is difficult to know at any time which statements the SET issued, which makes it difficult to transcribe them into a script that can be shared with the other databases or integrate them with database migration tools (like Flyway, Liquibase, or DBDeploy [4]). That makes it even more difficult for these professionals to adopt an Evolutionary Database Design [3].

Nowadays, trends in software development promote a more evolutionary approach, which means an iterative and incremental way. However, the developers who use SETs do not know which SQL statements were made until that point. Some SETs show the executed statements, but they never allow the developer to store them in a consistent and organized way.

This makes it difficult for these developers to apply evolutionary approaches to database development. These tools do not allow database versioning or integration with the tools that already exist for this purpose, database migration tools. Fowler and Sadalage even discourage the usage of SETs because of this lack of integration [12]. Consequently, it is hard to transmit the SQL statements executed in the development process to other sandboxes.

Database versioning is already tackled not only with tools like database migration tools but also with extensive literature about the topic [1, 2, 3, 4, 8]. The novelty in our problem is bringing the usage of SETs to the mix.

There are already tools that try to provide some solutions. Almost all of them involve making schema comparisons between the source database where the development was made and the target database where the changes will be applied. This process looks at the source and target schemas, finds the differences, and generates an SQL script that implements the differences. Then, the developer inserts this script into some database migration tool.

But, without the development context, the schema comparison technique cannot handle some situations. For example, a RENAME TABLE will be mistakenly interpreted as a CREATE TABLE followed by a DROP TABLE, which means losing valuable data [19].

Solving well this problem will allow: (1) developers that use SETs can enjoy database versioning, which means that replicating the work done in a sandbox is now easy, (2) database development and deployment cycles can be shorter, without fear of upgrade [16], (3) integration with Continuous Integration (CI), (4) more developers can do database development, even those that know little or nothing about SQL and (5) integration between database migration tools and SETs.

In our point of view, the better approach is to intercept the SQL statements made by the SET, at the development time, store them and then allow the developer to aggregate them in a migration that transforms a state of the database into another. The migrations generated can be sequential, which means that applying the first migration followed by the second will always have the same result. This migration is a set of SQL statements with a version that positions the migration between all other migrations. Therefore, the migrations can be shared across all database instances and the result will be always coherent.

To validate our approach, we first evaluated the impact of our solution while making the deployment process, which is insignificant. Then, we evaluated if our solution follows the best practices and techniques referred to by the literature, like techniques that allow an Evolutionary Database Design [3] and the practices that enable Continuous Database Integration (CDBI) [2, 8]. We concluded that it follows almost all the techniques and practices we found.

We argue, based on our validation, that the applicability of our solution is similar to the combination of the applicability of the database migration tools (database versioning which allows tracking, managing, and applying database changes), the database testing tools (regression testing of the database and its logic) and SETs (ease in managing and manipulating databases).

1.1 Motivation

Let's look at the following scenario: at the beginning of a project related to restaurant management, a developer must create restaurants and accounts and connect the accounts to the restaurants where they belong, and put one of the accounts as a restaurant manager. So, the developer must do the SQL that is in listing 1.1.

Imagine now the developer did all these operations directly through the interface of a SET, like DBeaver, DBVisualizer, or DataGrip. This means that she probably never saw the SQL of those operations and she even never needed to know what SQL was executed.

After she changes the database, she does the code to allow addition/updating/deletion of restaurants and accounts. When she wants to commit and push the code into production, she needs to obtain the SQL performed by the SET to execute them in production. There are four valid options:

(i) the developer tries to write the SQL that the SET supposedly did

 (a) as she was executing each operation

 (b) only at the end of the development

(ii) she does the deployment live with the SET

(iii) she uses a Schema Comparison Tool

Listing 1.1: Operations for Restaurant Management

```sql
1  CREATE TABLE restaurant(
2      `restaurant_id` BIGINT(20) NOT NULL,
3      `manager_id` BIGINT(20) NULL DEFAULT NULL,
4      `restaurant_name` VARCHAR(255) NOT NULL
5  );
6
7  CREATE TABLE account(
8      `account_id` BIGINT(20) NOT NULL,
9      `password` VARCHAR(255) NOT NULL
10 );
11
12 CREATE TABLE restaurant_account(
13     `restaurant_id` BIGINT(20) NOT NULL,
14     `account_id` BIGINT(20) NOT NULL,
15     PRIMARY KEY (`restaurant_id`, `account_id`),
16     INDEX `FK_restaurant_account_account` (`account_id`),
17     CONSTRAINT `FK_restaurant_account_restaurant`
18         FOREIGN KEY (`restaurant_id`)
19         REFERENCES `restaurant` (`restaurant_id`),
20     CONSTRAINT `FK_restaurant_account_account`
21         FOREIGN KEY (`account_id`)
22         REFERENCES `account` (`account_id`)
23 );
24
25 ALTER TABLE restaurant ADD CONSTRAINT `FK_restaurant_restaurant_account`
26     FOREIGN KEY (restaurant_id, manager_id)
27     REFERENCES restaurant_account(restaurant_id, account_id);
28
29 /* adding an admin to the system */
30 INSERT INTO account VALUES ('admin', 'password');
```

Both approaches (i-a) and (i-b) are bad because **the developer does not know the SQL she has to write completely or, even if she does exactly what to write, she may forget details**. For example, if she misses the detail of adding the PRIMARY KEY in line 15 from listing 1.1, the same account can be added several times to the same restaurant. Another example is if she forgets line 25 from listing 1.1, there is the possibility that the account placed as the restaurant manager is not even part of that restaurant.

Writing SQL by hand also has its problems. In [6], it is shown that, in the MediaWiki open-source project, 11,7% of all migrations executed have at least one "Syntax Fix" operation, which demonstrates a preponderance for syntax errors to happen. For example, the DBA can misspell some statements in the migration.

The approach (ii) is also bad because **the developer is doing the work twice and probably she will forget something**.

The focus of the approach (iii) is on capturing Data Definition Language (DDL) operations (schema modifications). In the example we provided, there is one Data Manipulation Language (DML) operation (line 30 in listing 1.1), which is not captured by any Schema Comparison Tool automatically.

Also, when using Schema Comparison Tools, there is the problem of identifying incorrectly the operations or even the order of the operations. For example, the developer renames a column (like line 2 in listing 1.2), but her Schema Comparison Tool identifies that the column with the previous name no longer exists and a new column appeared with the new name (like lines 5 and 6 in listing 1.2). This can be fixed, but only through manual indication. At this point, with the loss of the development context, it is almost impossible to identify these cases.

Listing 1.2: Alter name of column

```
1 /* rename column - operation done */
2 ALTER TABLE user RENAME COLUMN name TO username;
3
4 /* instead, Schema Comparison Tool identifies these operations: */
5 ALTER TABLE user DROP COLUMN name;
6 ALTER TABLE user ADD COLUMN username VARCHAR;
```

1.2 Problem Statement

DBAs and developers work with SETs to facilitate the database development process. However, these tools do not allow the developer to do database versioning or even use other complementary tools that are designed for database versioning.

When using a SET, we perform operations by interacting with its GUI. Under the hood, the SET issues multiple SQL statements to the database. However, SETs do not allow storing these statements anywhere directly. In some SETs, the developer can visualize which statements are issued, but only at the execution time, and others do not even show them to the user.

If the developer never knows the SQL statements performed or has difficulty consulting them, it becomes hard to transmit those to other databases, like other developers' databases, Quality Assurance (QA) databases, and production databases. So the problem is to identify and choose the statements that correspond to the operations we did to our development database and then save them in a consistent and organized way so that later they are executed in the correct order in other instances of the database.

This problem has several consequences:

- **The database version is most of the time unknown**. There is no place where we can see the exact database status (or version). For example, the developer never knows her local database version, or most developers do not know the state of the production database. It becomes very difficult to compare instances of databases since there is no easy way to check their versions and compare them.

- **Uncertainty about the statements already executed**. A developer never knows if the changes made by others are fully applied to her database instance. Someone can try some operation that goes well in her sandbox and forgets to communicate with the rest of the team.
- **It works in someone's sandbox but not in all sandboxes**. Related to the previous point, if there is no communication about some operation made, some features will not work in some sandboxes (described in [8]).
- **Hard to replicate changes**. With no database versioning, it is always hard to know which changes are to execute.
- **Extremely difficult to coordinate application and database deployments**. Most software teams use automatic techniques for application deployment, which makes possible several deployments per day. With a manual and uncertain database deployment process, it is difficult to coordinate changes in both application and database.
- **Fear of changing the database**. As explained in [16], there is a fear of changing the database because of the confusion and uncertainty of the manual database deployment process. So updates to the database happen rarely. But, if any new update takes months, there will be a lot more changes to do and a lot more risks to take since any change can cause problems with the data stored. An endless loop.

The state of the art misses some way to store SQL statements resulting from SET activity that later can be transformed into scripts that convert one database version into another.

We intend to allow any developer that uses a SET in their database development process to take advantage of all the easiness and quickness of SETs, guaranteeing, at the same time, the coherence of the database state across the different sandboxes (developers, QA, production, etc).

1.3 Methodology and Contributions

The focus of this work is to provide a way for SETs users to be able to enjoy database versioning and consequent better integration with CI pipelines and better replication of the changes between sandboxes. For that, we intend to enable an Evolutionary Database Design.

To do that, we developed a Java Database Connection (JDBC) that connects to a SET and intercepts automatically every single SQL operation that goes through it, sending them to our Version Control System (VCS). In our VCS, the developer has the power to create migrations based on those intercepted statements and even track, manage and apply those same migrations in other sandboxes.

The main contributions of this solution are the following:

- a Version Control System (VCS) for SQL statements;
- framework for testing databases only in SQL;
- integration of SETs with a database migration tool (our VCS);
- deprecation migrations;
- a way to resume the deployment where it failed.

1.4 Document Structure

The outline of the document is as follows. Chapter 2 provides some background about how to do a solution that meets the best practices for a more evolutionary approach. Chapter 3 lists several types of solutions that allow some techniques and practices we want to implement. Chapter 4 explains how our solution behaves and works internally. Then we evaluate quantitatively and qualitatively our solution in Chapter 5 and lastly, Chapter 6 states the conclusion and future work.

2 Background

Contents

This chapter provides some insights into the best techniques we found for good evolutionary database development and deployment processes. First, we explain in detail the five techniques that allow Evolutionary Database Design, in section 2.1. In section 2.2, we display the advantages of CI and what practices the database development should follow to be integrated into this process. Section 2.3 provides which scripts the DBA should maintain and why. Finally, section 2.4 explains how Ambler and Sadalage visualize a smooth system deployment.

2.1 Evolutionary Database Design

The database development, in several cases [16], is yet made in serial approach (waterfall or "Big Design Up Front (BDUF)"). This database development approach is not appropriate for continuously delivering features and updates.

So, [3] describes the **Evolutionary Database Design**, which, in simple words, is how to develop for databases in a more iterative and incremental way. This is allowed by these five techniques:

a. **Database Refactoring**

b. **Evolutionary Data Modeling**

c. **Database Regression Testing**

d. **Configuration Management of Database Artifacts**

e. **Developer Sandboxes**

2.1.1 Database Refactoring

Refactoring is the process of improving the internal structure of the software system without changing its external behavior. The goal is to clean up the code to minimize the chances of appearing bugs, by improving the design of the code, even after it has been written [10].

Inspired by this concept, Ambler and Sadalage transited it into the databases, coining the term **"database refactoring"**, which means **"a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics"** [3].

This concept has a big caveat: **this is only a valid concept when we look to the system as the user of an application that interacts with the database**. In other words, the goal of the author is to maintain the same black box functionality at the system level.

Because of that, the author provided another similar concept, but simpler and without this caveat: "database refactoring, the process". In this case, this is "the act of making the simple change to your database schema". To explain better this term, the author said the following: **"One way to look at**

database refactoring is that it is a way to normalize your physical database schema after the fact".

One example provided was a column named *FirstDate* that is in the *Person* table. In this table, a row can represent: (1) a *Customer*, in which *FirstDate* represents their birth date, or (2) an *Employee*, in which *FirstDate* represents their hire date.

However, now there is a need to support a *Person* that can be both a *Customer* and an *Employee*.

The database refactoring applied is "Split Column". Replaces *FirstDate* column with *BirthDate* and *HireData* columns. To maintain the behavioral semantics, all source code that accesses *FirstDate* must be updated to work with *BirthDate* and *HireDate*. To maintain the informational semantics, the developer must write a migration that translates all *FirstDate* values to *Birth Date* and *HireDate* values.

The author provide also what a database refactoring is not: for example, a small transformation to your schema to extend it, like adding a column.

2.1.2 Evolutionary Data Modeling

Agile methodologies appeared because before the way to go was to do a "Big Design Up Front (BDUF)". But, it does not mean that even with an agile or evolutionary approach, we can ignore the designing process. It is still important that, before doing some code, it should put some effort into thinking and designing the architecture. One way of doing it is by modeling the data.

The difference is that it is not supposed to design the whole system at once, but there must be iterations of the design. The whole process should be performed iteratively and incrementally.

If we have to define **Evolutionary Data Modeling** [1], we can define it as the act of exploring data-oriented structures performed iteratively and incrementally. This can also be a highly collaborative process, but this will depend on the project and the team behind it.

2.1.3 Database Regression Testing

Regression testing is important in software because allows verifying that the whole system is working as expected and satisfies its requirements, especially after it is modified [15].

This practice is already made extensively in application land. However, most data professionals think that tests are not needed.

There is yet another view: since the database is used by one or more applications, if we test the applications the database is connected with, we test the database indirectly. And this is possibly true, even though the tests can be not as complete as they should be.

But there are some problems with this last approach [17]:

- **Slow Tests**. Tests that require access to a database are, on average, two orders of magnitude slower than the same tests without the database. And, nowadays, with the usage of CI/CD pipelines, application tests are executed so often that there is the need to reduce the time each test takes to run to prevent deployments from pilling up.
- **Erratic Tests**. Tests that sometimes pass and sometimes fail. Some examples:
  - **Unrepeatable Tests**. Test that behaves differently the first time it is executed when compared with subsequent test runs. This can be caused by using a database that comes from other tests.
  - **Lonely Tests**. Test that will depend on other tests being performed first. It will pass when those tests are executed first and it will fail when it executes alone.
  - **Test Run War**. If a shared database is used between more than one developer and there are at least two developers executing tests, the content of the database will be inconsistent and the tests can pass or fail randomly.
- **Obscure Tests**. Tests that we do not know exactly what is being tested. For example, when using a pre-populated database, probably it will be used for several tests for different purposes.

Because of all these cases, **the application tests should simulate the database in memory**, by replacing the data access layer with an in-memory data structure, and **the database should be tested separately with its tests**.

The database tests should test the database schema and database logic. For example, it should test stored procedures, triggers, data validation rules, referential integrity rules, etc.

2.1.4 Configuration Management of Database Artifacts

Every artifact related to the database should be under Software Configuration Management (SCM) control. SCM is the discipline of controlling changes in large and complex systems [5, 18].

The intention behind this is to allow database versioning and the ability to roll back (in the case of a database, this is only possible if it was provided rollback operations before). By allowing rollback versions, it will allow database refactoring, consequently.

2.1.5 Developer Sandboxes

A sandbox is an environment where projects can be implemented, tested, and run. Ambler enumerated the following four types of sandboxes [1] (illustrated in fig. 2.1):

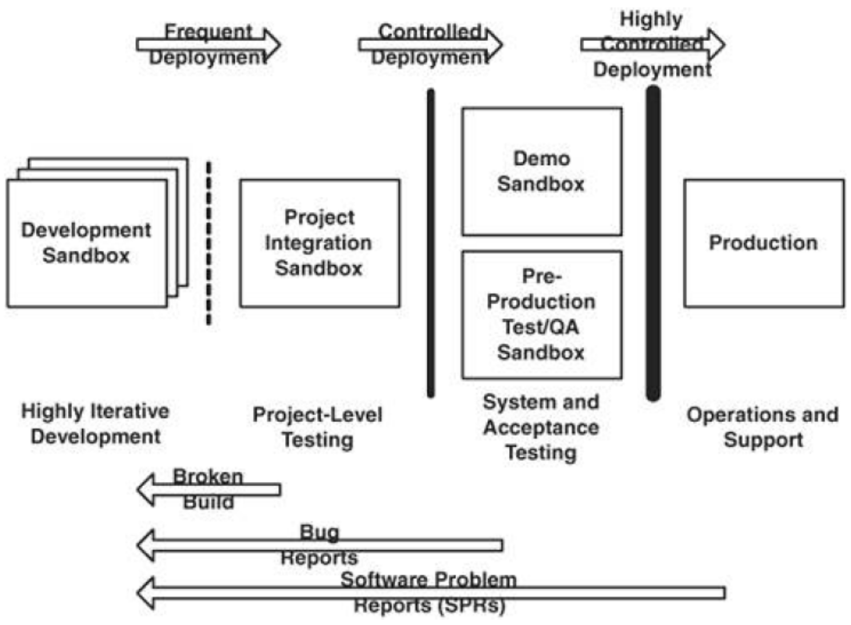

Frequent Deployment
Controlled Deployment
Highly Controlled Deployment
Development Sandbox
Project Integration Sandbox
Demo Sandbox
Pre-Production Test/QA Sandbox
Production
Highly Iterative Development
Project-Level Testing
System and Acceptance Testing
Operations and Support
Broken Build
Bug Reports
Software Problem Reports (SPRs)

Figure 2.1: Different types of sandbox [1]

- **Development**. The working environment of individual developers, programming pairs, or individual feature teams where a system may be built, tested, and/or run. In such an environment, the developer can work isolated from the rest of the team, which means that she can make whatever changes she wants and validate them without having to worry if it will affect the other members of the team. After testing successfully here, the developer can send her work to the Project integration sandbox.
- **Project integration**. Often referred to as a build environment, here the work of the whole team is combined, validated, and tested before being promoted to Test/QA sandbox.
- **Test/QA**. Also known as a "Pre-production Test sandbox", this simulates a production environment where the QA team can perform their tests, similar to real-world end-users. This environment is also important to test multiple projects being executed in the same environment.
- **Production**. Where the project will be deployed after passing with success in all the previous stages. This is the sandbox that will be interacting with real users.

In [3], it is referred one more type of sandbox, the "Demo sandbox". The team should make an effort to deploy the project in this sandbox at least once an iteration to produce working software that can be used by the project stakeholders.

This technique is also described in [17], as a way to avoid some of the problems with tests described in section 2.1.3, like *Obscure Tests* and *Test Run War*.

2.2 Continuous Integration

*"Continuous Integration is a software development practice where members of a team **integrate their work frequently**, usually each person integrates at least daily - leading to multiple integrations per day. **Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible**. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."* [11]

According to [8], CI usually follows these six steps: (1) **Source Code Compilation**, (2) **Database Integration**, (3) **Testing**, (4) **Inspection** (automated code inspections to enforce some code rules), (5) **Deployment** (generating a software artifact package with the latest code changes and making it available for a test environment) and (6) **Documentation and Feedback** (documentation generation based on the code and provide feedback via email or another method).

2.2.1 The Value of Continuous Integration

Duvall et al. state the following advantages of CI:

- **Reduce Risks**. 1. **Defects are detected and fixed sooner** since every integration generates a build and run tests and there are several integrations a day; 2. **the health of software is measurable** since complexity and code standards can be evaluated and 3. **reduce assumptions about the system**, for example, environment variables, because rebuilding and testing in a clean environment reduce assumptions about the system (e.g., environment variables)
- **Reduce Repetitive Manual Processes**. The automation of these processes enables that **every time a commit occurs, all these processes can be triggered** in an **ordered way**. If they are manual, it would be very difficult or even impossible to do these processes so often.
- **Generate Deployable Software**. The project has at any time **updated deployable software** with the most recent changes. This enables constant releases with new features and bugs are fixed quickly, almost immediately.
- **Enable Better Project Visibility**. CI provides information on the recent build status and code quality metrics, which can induce more **effective decisions**, and also provides the ability to notice trends related to the project as overall quality, build success or failure, etc.
- **Establish Greater Product Confidence**. Imagine for a second that we don't have CI. **Integrations probably wouldn't be as frequent as they should be** and so some teams would feel some discomfort because **they wouldn't know the impact of their code changes**. With CI, the development team can have greater confidence in what they are producing because they know the

software behaves as expected due to testing, the standards set for design and code are met, and they know that at the end of the day they have a functionally tested product.

2.2.2 Continuous Database Integration

"*CDBI is the process of rebuilding your database and test data any time a change is applied to a project's version control repository.*" [8]

As we see until now, CI offers many advantages to development teams. However, it can only be complete if we also integrate the database continuously. If we don't take this step, the advantages of CI go away.

So, to adopt CDBI, it is described in [2] the best practices for such an approach:

- **Automate the build**. The database build process should be automated, integrated with the overall build process, and should detect if there is the need to apply some changes, apply them and run the test suite.
- **Put everything under version control**. All database artifacts should be placed under version control.
- **Give developers their database copies**. What is already referred to as developer or database sandboxes 2.1.5. The goal of this one is to allow the developer to work in her code and her database instance and try whatever she wants, providing her a safe environment for experimenting with things. After fully testing changes and verifying that they are safe, they can with a great level of certainty apply the changes to the project integration environment, reducing the probability of individual developers breaking the build.
- **Automate database creation**. A process that is done often, so it should be automated.
- **Refactor each database individually**. Allow to have different databases in different versions and evolve them to any other version.
- **Adopt a consistent identification strategy**. To apply database changes, they must be ordered. Three options are suggested: incremental integer number (1, 2, 3, 4, ...), date/time stamp, or a release number (v2.3.1.5).
- **Bundle database changes when needed**. Be able to apply several database changes at once. For example, the database version is 1701 and must be applied to three new database changes (1702, 1703 and 1704).
- **Ensure that the database knows its version number**. To execute the right database changes, the system must know which is the current database version.

There is one more interesting point that CDBI enables: **the automatic creation of documentation for the database** [8] to help all the stakeholders of the project to understand the relationships. For example, it can be possible to generate an Entity Relationship Diagram (ERD) to facilitate the view of the database.

2.3 Scripts to Maintain

There are three types of scripts that were recommended in [1] to be maintained by the DBA:

Database change log. This log stores all changes made to the schema (DDL) by the order of its execution.

Data migration log. All DML are made to reformat or cleanse the data throughout the life of the project.

Update log. Sometimes, some databases are used by multiple applications and, when some change is made by one team, for example, moving a column, the others will not update their code right away to match with the new schema. So it is important to maintain retro-compatibility. This is achieved by this log. In this example, it is added to the Database change log the CREATE COLUMN the new column in the other table and the creation of triggers to maintain the data updated in both columns. In the Data migration log is added the process that copies the data. In the Update log is added the respective DROP COLUMN and the destruction of the triggers associated with a deprecation period - the period that all other teams have to update their applications.

2.4 Process to Deploy The System

Ambler and Sadalage have described in [3] what steps that a system with a database must take to prevent errors and failures and to always allow a way back to the previous state before the deployment process has started. The steps are the following:

1. **Back up the database**. Large databases are difficult to back up, but if possible, do it. This becomes more important as the wait for a new deployment extends in time because it means that there will be more changes to make.

2. **Run previous regression tests**. The goal of this step is to guarantee that the database is really in the version the system thinks it is. If someone did some change directly in the production database, the state of the database will be different from the one expected. Be careful with the chosen tests to be executed because they can cause some unwanted changes.

3. **Deploy the changed application(s)**. Usual deployment of the application.

4. **Deploy the database refactorings**. Apply the right migrations.
5. **Run the current regression tests**. The execution of this test suite is useful to understand if the system is running properly or not. Again, be careful with the tests executed.
6. **Back out if necessary**. If something went wrong while testing, the backup made preciously should be restored. A piece of advice given by the authors is that if a deployment is complex, deploy in increments. This is more difficult because can raise other problems, but it may be better since the entire deployment does not fail just because one portion of it has something wrong.

3 Related Work

Contents

In this section, we explain what solutions we found that try to solve similar problems to ours. We start by detailing how the main two database migration tools in the market currently work: Flyway, in section 3.1, and Liquibase, in section 3.2. After that, we describe DBDeploy, in section 3.3, which is a similar solution to the previous ones but simpler. Next, we show two database testing tools: DBUnit, in section 3.4, and utPLSQL, in section 3.5. In the end, we analyze P6Spy, a JDBC Proxy, in section 3.6.

3.1 Flyway

Flyway was created to solve database-related problems such as tracking changes, ensuring that changes are applied at the deployment, and checking which changes have already been applied and which should be applied.

Flyway is a Java tool for **tracking, managing, and applying database migrations**, which supports several different Relational Database Management Systems (RDBMSs). Flyway ships with its own Java version and several JDBCs. A JDBC other than the packaged ones can be added to the *classpath*.

3.1.1 How Does It Work?

As explained in [9, 14], Flyway can be run within the app at startup or as a separate tool. It connects to the database through JDBC, since it is a Java application.

It keeps track of available and applied migrations in the past in a dedicated schema history table. This table is created within the database Flyway is working with. The search for migrations is usually made in a folder, a default one or another defined by the developer, or in the *classpath*.

By knowing the applied migrations and the available migrations, it can compare both sets and know which available migrations are not yet applied. These, known as **pending migrations**, are then applied against the database when the **migrate command** is issued. This command executes this group of pending migrations against the database from the lowest to the highest existing version.

In this process, Flyway does some validations to avoid corrupting the database.

3.1.2 Migrations

Migration is a set of instructions in a file. This can be a SQL file (.sql) with SQL statements or can be, for trickier situations, a script made in Java (Java class that inherits from JavaMigration class), Python (.py) PowerShell (.ps1), Dos Batch (.bat or .cmd) and Linux (.sh or .batch).

There are four migration types:

- **Versioned Migration**. It takes the database from the preceding version into the version expressed in migration filename. Naming convention: V{*version*}__{*description*}.{*extension*}

- **Undo Versioned Migration**. It takes the database from the version expressed in migration filename into the preceding version. Naming convention: U{*version*}__{*description*}.{*extension*}

- **Repeatable Migration**. It is executed every time its name changes. These migrations should be idempotent to not cause any error. They are usually used for views, procedures, functions, or packages and they can also be used for simple statements. They are always applied last, after all other migrations. Naming convention: R__{*description*}.{*extension*}

- **Baseline Migration**. A long project will usually have hundreds of migrations and every time a new sandbox is created, those hundreds of migrations must be executed, doing so many useless operations, like creating objects that do not even exist anymore. To avoid this scenario, there is the concept of baseline migration. Baseline migration is the result of applying all regular migrations up to the version associated to it and all repeatable migrations. Then, when a new sandbox is created and exists a baseline migration, the migration process will start from the most recent baseline migration. Naming convention: B{*version*}__{*description*}.{*extension*}

The file name consists of the following parts:

- prefix: V for versioned, U for undo, R for repeatable, and B for baseline migrations.

- version: string with dots or underscores separate as many parts as you like. Repeatable migrations do not have a version.

- separator: __ (two underscores).

- description: string in which underscores or spaces separate the words.

- suffix: any extension. The type of file is identified by its extension. Java-based migrations do not have an extension.

3.1.3 Schema History Table

Flyway stores the metadata about migrations in the Schema History Table (described in table 3.1). This table is created in the current database in the default schema. Its name is *schema_version* (default name that can be changed).

For every copy of a database, Flyway knows what migrations have already been applied, who applied them, when they were applied, and what was the outcome (success or not) through this table. This table also provides an audit trail of all the changes executed against the schema of the database.

Schema History Table Fields	Field Description
installed_rank	auto-incremented identifier to allow know out of order detection
version	string in the form of one or more numeric parts separated by dots or underscores
description	string that describes what migration does
type	type of migration (can be SQL, JDBC, etc)
script	migration full name
checksum	part of verification mechanism ensuring that migration scripts haven't been changed since they applied to the database
installed_by	who installed migration
installed_on	when migration was installed
execution_time	how many milliseconds migration took to run
success	migration execution was successful, not successful or pending

Table 3.1: Flyway's Schema History Table fields

3.1.4 Validations

There are some validations that Flyway does to avoid corrupting the database, such as:

- There are some non-empty schemas?
  - Yes. There is already the Schema History Table?
    - Yes, which means that Flyway already recognizes such database.
    - No, which means Flyway will think that it is talking with the wrong database and abort the process of migrating. To avoid this scenario, the developer should do the baseline command.
  - No. So, it creates Schema History Table.
- Do all checksums of known migrations match? If yes, proceed. If not, migration does not even start.
- Are there any unknown applied migrations? For example, has someone deleted an older migration already applied? If so, the developer is warned and the process ends there. If not, proceed.
- Do all new migrations have a strictly higher version? If yes, migration process starts. If not, the process is aborted.

3.1.5 Apply Migrations

Migrations are always executed in order (lowest version to the highest). Flyway is responsible for finding if all statements present in migration are transactional, meaning that they are ACID, and if they are, Flyway executes the migration inside a transaction. If multiple migrations are applied at once, Flyway can perform all migrations inside a transaction. Again, Flyway must verify if all statements inside those migrations are transactional.

This varies from RDBMS to RDBMS. For example, in PostgreSQL, almost all DDL statements are transactional. This means that regardless of what statements (DDL or DML) are in the migrations, they will be executed within a transaction, with very few exceptions. But in MySQL, only DML statements are transactional, which means that only migrations that have only DML statements can be executed within a transaction.

Flyway reports all warnings and messages returned by the RDBMS and if there is some error, it displays the message that detailed the error and marks the migration as failed. Flyway automatically rolls the migrations back if possible.

3.1.6 Available Commands

- **migrate**. Migrates the current database to the latest version found.
- **clean**. Drops all objects (tables, views, triggers, stored procedures, etc) in the configured schemas.
- **info**. Provides the status of the system, like which migrations have already been applied, which migrations are still pending, which ones are successful or not, and when they were executed.
- **validate**. Verifies that the migrations applied to the database match the available ones through checksums.
- **undo**. Undoes the most recently applied versioned migration if it is undoable, which means that there is an Undo migration with the same version.
- **baseline**. Command designed to be used when a Flyway project starts with an existing database. It takes a snapshot of the database and generates a baseline migration. After the execution of this command, when migrate command is executed, all migrations up to and including baselineVersion are ignored, executing instead the baseline migration created in the process.
- **repair**. It makes some operations to repair the Schema History Table, or at least try to do so.

3.2 Liquibase

"*At its core, it's a tool to make sure that your database matches your application code. It's a schema migration tool or whatever you want to call it. Let's you define the series of steps that it takes to get your database from an empty database to the state that your application requires it to be in and then*

Liquibase makes sure that for any given database it gets it to that state you need." - Voxland, the Liquibase founder.

Liquibase is also a Java tool that supports multiple RDBMSs to manage the database changes. Currently, it ships with its own Java and with some mainstream JDBCs.

In the Liquibase world, there are two main concepts:

- **Changelog**. The file contains a series of changesets, usually made to a single database by all the developers. The developer can also include some other files to a changelog, avoiding, for example, that a single changelog becomes too big.

- **Changeset**. Sets of changes performed inside a transaction when possible. It is advised by Liquibase's founder to have one single statement per changeset or a set of statements if they are all transactional. Each changeset has a unique identifier composed of id, author, and changelog filename.

3.2.1 How Does It Work?

Whenever updates are required to be made, the developer adds a changeset to the changelog of the project describing the changes that are needed. Each changeset has a unique identifier formed by an id, the author name, and the changelog filename where it is written. The id and the author name are provided in the metadata of each changeset. It is advised that the id behaves like an auto-increment, but this is left entirely to the responsibility of the developer. Also, different authors can have different ids, with different counting. In Liquibase, authors work almost as workspaces.

To track which changesets are already executed, Liquibase uses an auxiliary table, DATABASECHANGELOG, where it stores every changeset executed by its identifier (id, author, file-name). Liquibase also stores an MD5 hash sum that is computed from the metadata and content of a changeset, to identify possible future changes in the changesets that have already been applied [13].

Every **update command** issued finds out which is the last executed changeset, by reading the DATABASECHANGELOG table and comparing the identifier stored in this table and the info provided in the metadata of a changeset. Then, Liquibase looks to the changelog file, finds the changesets that come after, and executes them. The order of the execution is obtained by the order of the changesets in the changelog file.

Each changeset is run inside a transaction and, when it ends, its result is stored in the DATABASECHANGELOG table. If there are multiple operations and one of them does auto-commit, the changeset may be not set as executed, but some statements inside the changeset were already applied. It is advised that a changeset has only one statement or several transactional statements.

3.2.2 Different Changelog Formats

Liquibase allows several changelog formats: SQL, XML, JSON, and YAML.

3.2.2.A SQL Format

The most basic and user-friendly format for new users is the **SQL format** (example in listing 3.1). To be a SQL-formatted changelog, it must start with a comment saying that the format used in that file is SQL.

In this format, each changeset starts with a comment that contains the metadata of the changeset, where it must have at least the id and the author of the changeset. After the initial comment, the SQL statements of this changeset are listed.

To highlight two more things: 1. in this format there is no automatic rollback, so it must be provided a rollback for each changeset if the developer plans to have a rollback and 2. the developer can provide specific changesets for an RDBMS (in fact, this happens across all formats).

Listing 3.1: Liquibase Changelog in SQL format

```sql
--liquibase formatted sql

--changeset nvoxland:1
create table test1 (
    id int primary key,
    name varchar(255)
);
--rollback drop table test1;

--changeset nvoxland:2
insert into test1 (id, name) values (1, 'name 1');
insert into test1 (id,  name) values (2, 'name 2');

--changeset nvoxland:3 dbms:oracle
create sequence seq_test;
```

3.2.2.B XML, JSON, and YAML Formats

The **XML, JSON and YAML formats** (example in listing 3.2) behave in the same way and have the same functionality. The difference is the style in which the changelog is written. The developer can opt for the one she prefers.

With these formats, the developer can use the transformations already created by Liquibase. If for some reason, the developer needs to do a different transformation from the ones predicted by Liquibase, she must include another SQL changelog in this changelog.

Listing 3.2: Liquibase Changelog in XML format

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog (some standard attributes)>
    <preConditions>
        <runningAs username="liquibase"/>
    </preConditions>
    <changeSet id="1" author="nvoxland">
        <createTable tableName="person">
            <column name="id" type="int" autoIncrement="true">
                <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="name" type="varchar(50)"/></column>
        </createTable>
    </changeSet>
    <changeSet id="2" author="nvoxland">
        <addColumn tableName="person">
            <column name="username" type="varchar(8)"/>
        </addColumn>
    </changeSet>
</databaseChangeLog>
```

When using these formats, there are the following advantages [20]:

- **Auto Rollbacks**. Out of the box, each changeset has auto rollbacks. The developer can still provide her rollback, but can also rely on the default one. For example, for a CREATE TABLE changeset, Liquibase will automatically know that the inverse is DROP TABLE and it will be applied when a rollback is requested.

- **Cross-platform Compatibility**. Nathan Voxland worked at a consulting company when he created Liquibase. At the time (2006), there were a lot of clients requesting similar websites, like E-commerces or Content Management Systems (CMSs). He needed to provide similar database schemas to several different clients who were using different RDBMSs. With these formats, Voxland could write the changesets in a language-agnostic to the RDBMS, thus being able to use the same changesets. Liquibase would then generate specific SQL code for each RDBMS where the changelog was executed. This cross-platform compatibility of these formats mixed with the Liquibase snapshot/backup feature allows migrating a database completely from one RDBMS to another. Another interesting use case is that in almost all situations, the project can use one RDBMS in production, but test another, even maybe an in-memory RDBMS. Of course that this use case is trickier and not very recommended, but the possibility is still there.

3.2.3 Audit for DBAs

In some projects, some DBAs are responsible for the management of databases and they mostly like to have control over which SQL statements are executed in the database. So, to allow the developers to use Liquibase to change the database accordingly to their needs and also allow the DBA to (1) check which SQL statements will be executed, there are commands like **update-sql** (to inspect which code will be executed with the **update command**) and **rollback-sql** (to inspect which code will be executed with the **rollback command**), and (2) sometimes make some changes to the generated SQL, by using *modifySql* tag in the XML, JSON, and YAML changelog formats or by changing directly the SQL changelog.

3.3 DBDeploy

As explained in [4], DBDeploy is a database refactoring manager that *"automates the process of establishing which database refactorings need to be run against a specific database to migrate it to a particular build"*.

Important to state that DBDeploy is developed as an Ant task, meaning that Ant must be installed in the environment to use DBDeploy. Ant is a build tool that automates the steps to build and deploy software. There are also other versions for Maven and Gradle. But it seems that it cannot work alone. The purpose of this approach is to be well integrated with build tools and consequently be well integrated into CI processes.

The steps that DBDeploy does are the following:

1. In a specified directory, **DBDeploy finds all .sql files** (supposedly those will be the migrations) and **orders them by name**. Advice given by the author is that the **migration file name should begin with an incremental number**.

2. DBDeploy **reads the changelog table** in the specified database. This table knows **which migrations have already been applied** to the database.

3. Knowing the applied migrations and all migrations files, DBDeploy determines **which migrations have not been run** against the database and then **generates a script containing all statements that should be applied**.

Attribute	Description
driver	Specifies the jdbc driver.
url	Specifies the url of the database that the refactorings are to be applied to.
userid	The ID of a dbms user who has permission to select from the changelog table.
password	The password of the user specified by userid.
dir	Full or relative path to the directory containing the delta scripts.
outputfile	The name of the script that dbdeploy will output. Include a full or relative path.
dbms	The target dbms. Valid values are: ora ->Oracle syb-ase ->Sybase ASE hsql ->Hypersonic SQL mssql ->MS SQL Server mysql ->MySQL Database

Table 3.2: DBDeploy arguments

This script should be then revised by the DBA that is also responsible for executing them. The scripts are generated with the statements responsible to create and update the changelog table.

The arguments that must be provided when the developer must call DBDeploy as an Ant task are in table 3.2.

3.4 DBUnit

DBUnit is a JUnit extension and is a unit testing tool to test relational database interactions in Java that tries to answer a single problem stated by its own creator: "*You have a SQL database, some stored procedures, and a layer of code sitting between your application and the database. How can you put tests in place to make sure your code really is reading and writing the right data from the database?*"

It allows to set up the database into a known state before each test run and provides a very simple XML-based mechanism for loading the test data (example in listing 3.4). The difference with other unit testing tools is that it works with a real and live database and not with mock objects.

DBUnit allows not only testing the database directly but also the data access layer, which abstracts the database access, usually to transform relational-oriented data into object-oriented data.

So, any database test should follow these three steps:

1. Removes old data that could be left in the database from previous tests

2. Loads some data into the database from an XML file

3. Runs the test.

Listing 3.3: XML Dataset Format

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <!-- each element is a row -->
    <!-- the tag name is the table where row will be inserted -->
    <!-- each attribute represents a value for a column -->
    <!-- (attr_name, attr_value) = (col_name, col_value) -->
    <CLIENTS id='1' first_name='Charles' last_name='Xavier'/>
    <ITEMS id='1' title='Grey T-Shirt' price='17.99'
        produced='2019-03-20'/>
    <ITEMS id='2' title='Fitted Hat' price='29.99'
        produced='2019-03-21'/>
    <ITEMS id='3' title='Backpack' price='54.99'
        produced='2019-03-22'/>
    <ITEMS id='4' title='Earrings' price='14.99'
        produced='2019-03-23'/>
    <ITEMS id='5' title='Socks' price='9.99'/>
</dataset>
```

DBUnit allows not only testing the database directly but also the data access layer, which abstracts the database access, usually to transform relational-oriented data into object-oriented data.

3.4.1 How Does It Work?

To create a test in DBUnit, the developer must implement a class that extends the *DBTestCase* class, in which, she must provide some properties like a JDBC driver class, the connection url, database username and database password and must also implement the following methods:

- **getDataSet**. In this method, the developer provides the dataset DBUnit has to work with. She must provide which XML file (example in listing 3.3) has the information about the data to be loaded before each test case.
- **getSetUpOperation**. Here is defined what behavior the developer expects DBUnit to have before each test.
- **getTearDownOperation**. Where the developer defines what behavior DBUnit should have after each test.

- **Real Tests**. These methods can have any name, and each should implement a unit test. Each method must have the @Test annotation.

There is also another way to implement a test without implementing these methods, by using:

- **@Before**. Annotation used in a method that implements what should be done by DBUnit before each test.
- **@Test**. Used in a method that implements a test.
- **@After**. Placed in a method that describes what should be done by DBUnit after each test.

To execute the tests, at least when using DBUnit integrated with Maven, the developer only have to run *mvn clean verify*.

3.4.2 Pre-defined Behaviors

To facilitate, DBUnit provides several behaviors that we can use in any method, but they are expected to be used in **getSetUpOperation**, **getTearDownOperation**, **@Before**, and **@After** methods. These behaviors are the following[1]:

- **CLEAN_INSERT**. Deletes everything from any database table referred to in the dataset and inserts new content. Equivalent to calling *DELETE_ALL* followed by *INSERT*.
- **DELETE**. Deletes database table rows that matches rows from the dataset.
- **DELETE_ALL**. Deletes everything from any database table referred to in the dataset. Tables that are not in the dataset remain unaffected.
- **INSERT**. Inserts new database tables and content from the dataset.
- **REFRESH**. Refresh the content of existing database tables by inserting or replacing existing data based on rows from the dataset. Any rows that are not in the dataset remain unaffected.
- **TRUNCATE_TABLE**. Deletes everything from any database table referred to in the dataset. Tables that are not in the dataset remain unaffected. Identical to *DELETE_ALL*, however, this operation cannot be rolled back and is supported by few database vendors.
- **UPDATE**. Updates the contents of existing database tables from the dataset.

[1] https://springtestdbunit.github.io/spring-test-dbunit/apidocs/com/github/springtestdbunit/annotation/DatabaseOperation.html

Listing 3.4: DBUnit Test

```java
public class Test extends DBTestCase {
    public Test(String name) { super(name); /* set properties */ }

    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSetBuilder().build(
            new FileInputStream("user.xml"));
    }
    protected DatabaseOperation getSetUpOperation()
            throws Exception {
        return DatabaseOperation.REFRESH;
    }
    protected DatabaseOperation getTearDownOperation()
            throws Exception {
        return DatabaseOperation.NONE;
    }
    @Test
    public void testById() {
        int userId = 5;// get user id from database
        assertThat(1, is(userId));
    }
}
```

Annotation	Level	Description
–%suite(<description>)	Package	Mandatory. Marks package as a test suite with an optional description.
–%test(<description>)	Package/procedure	The annotated procedure is a unit test with an optional description.
–%throws(<exception>, [, ...])	Procedure	To be marked as successful, the following annotated test procedure must throw one of the exceptions provided.
–%beforeall	Procedure	The following procedure should be executed before all tests.
–%afterall	Procedure	The following procedure should be executed after all tests.
–%beforeeach	Procedure	The following procedure should be executed before each test.
–%aftereach	Procedure	The following procedure should be executed after each test.
–%rollback(<type>)	Package/procedure	Defines transaction control: - auto - before block, test and after block are run inside a transaction and then rolled it back - manual - there is no automatic transaction

Table 3.3: utPLSQL Annotations

3.5 utPLQSL

utPLSQL is a testing framework for Oracle RDBMS that can be used through, for example, a Command-line Client (CLI) or Maven. This framework has two main concepts: annotations and expectations.

3.5.1 Annotations

An annotation is a single line comment (that starts with "–" - double hyphen) followed directly by a "%" signal followed by an annotation name followed by an optional text placed in single brackets. This is used to provide metadata to utPLSQL, which enables the tool to understand the context of the statements inside the package. The most relevant annotations are in table 3.3.

3.5.2 Expectations

Expectations are how utPLSQL enables the developer to compare the actual data against the expected data and so validate or not the test.

An expectation is the combination of the expected value, an optional custom message that characterizes the expectation, the matcher (operation between expected and actual value) used to perform the comparison and the actual value. An example of an expectation: *ut.expect(6/2).to_equal(3);*.

3.5.3 How Does It Work?

First of all, the developer must create a package (example in listing 3.5) that starts with the *–%suite* annotation. Then, inside this package, she creates tests by creating a procedure for each test with the *–%test* annotation above it. If for some reason there is code that should be executed before each test, the developer creates a procedure with *–%beforeeach* annotation. The same if there is code that should be executed after each test, with *–%aftereach* annotation. Sometimes, for related tests we need to make something before the execution of all tests or after that execution, so the developer creates a procedure with *–%beforeall* or *–%afterall* annotation, depending on what the developer wants.

As the last step, the developer puts an expectation comparing the actual value and the expected one. If that comparison is true, the test is successful.

For those tests that should throw some exception, like a test that tests the integrity of a foreign key, utPLSQL allows the developer to put the *–%throws* annotation with the error codes that can be raised. If the execution of the test raises one of the codes provided in that annotation, the test is successful.

All tests inside a package form one test suite, which are then executed by running the following utPLSQL command *exec ut.run('package_name');*

Listing 3.5: utPLSQL Test (from https://www.utplsql.org)

```
create or replace package test_betwnstr as
  --%suite(Between string function)
  --%test(Returns substring from start position to end position)
  procedure basic_usage;
end;
/
create or replace package body test_betwnstr as
  procedure basic_usage is
  begin
    ut.expect( betwnstr( '1234567', 2, 5 ) ).to_equal('2345');
  end;
end;
/
```

3.6 P6Spy

Java applications use JDBC to communicate with the database. And sometimes, knowing exactly what statements are being made is important, for example when using an Object-relational Mapping (ORM). This is why P6Spy exists.

P6Spy is a JDBC that logs all statements that pass through it and the developer does not need to change any logic in her Java application.

P6Spy works as a proxy, it logs the statement and then simply forwards it to another JDBC (internal JDBC) that knows how to communicate with the database. So, the developer must provide the full class name of the internal JDBC in the properties file and modify the JDBC connection URL used in the application to include “p6spy:”. For example, if the URL is “jdbc:mysql://host:port/db”, then just change it to “jdbc:p6spy:mysql://host:port/db”.

Another interesting feature is filtering just what type of statements the developer wants to log. This is made by passing regex strings for the properties file describing what should be logged.

3.7 Discussion

We analyze several types of solutions to understand how they work and how we can pick several points from each one of them to make a better solution. We analyze (1) database migration tools, which are tools that are specialized for allowing database versioning and include the database changes into CI pipelines, (2) database testing tools, which allow to write unit tests to the database and (3) a JDBC proxy that allows logging of all SQL statements that passes through it.

3.7.1 Database Migration Tools

Flyway. It is one of the most used database migration tools in the world because its migration concept is easy to grasp. Its simple operations make the database creation and updating processes easy. Support writing migrations not just in SQL, but also in Java for more complex stuff that need to implement logic and a bunch of other script types, like Python or bash.

Liquibase. It is more complex than Flyway with a lot more features and with different concepts like changelogs and changesets instead of migrations. Provide rollbacks out of the box when using XML, JSON, or YAML formats and have several tools to enable some audit of what is being made in some specific situation.

DBDeploy. It is a simple tool whose only purpose is to aggregate migrations and track which migrations were already applied or not.

3.7.2 Database Testing Tools

DBUnit. Allows unit tests written in Java, which is important when some logic is needed. It provides most operations someone will ever need while writing database tests, like cleaning a database, populating a database with a dataset, etc.

utPLSQL. Only for Oracle users, its usage is made basically by tags. The developer creates a package and inside has a test suite written in PLSQL, which can be good because is written in the same language as the RDBMS, which means that DBAs may have a better time using it.

3.7.3 JDBC Proxy

P6Spy. A simple JDBC that works as a proxy, which goal is to log the statement that is being executed and pass it to the internal JDBC so that this one does the communication with the RDBMS.

4 Solution

Contents

The goal of this work is to integrate Schema Editing Tools (SETs) with an evolutionary and collaborative solution that fits well with Continuous Integration (CI). This implies providing a database connection that can connect at least one SET to our VCS. Evo VCS allows the implementation of the techniques described in [3] that enables an Evolutionary Database Design and follows the best practices for CI [2].

It is worth mentioning that there were three secondary goals: (1) enable migrations that are generated by our VCS can work with other database migration tools and allow that migrations generated from other tools can work with our solution, (2) enable our VCS to work independently as a standalone program without needing a SET connection and (3) generate files that are easy to manipulate by any developer, if for some reason they need to overcome some possible bug or limitation our solution may have.

In this chapter, to help you better understand how Evo DB works, we first provide an explanation of the software components used (section 4.1), we explain the objects that our VCS knows and understands (section 4.2), and all the data structures (section 4.3) Evo DB creates to make our solution work. Then, we provide some more concepts that are used throughout this chapter (section 4.4), followed by how our JDBC works (section 4.5). Next, the architecture of the system and how everything comes together (section 4.6). We finish by stating what RDBMSs our solution supports (section 4.7).

4.1 Software Components

Evo VCS is the main component of our solution where the developer can interact with the system. Here she manages migrations, generates migrations with SQL statements that come from the SET, tests and validates the project, and deploys the database. In the following explanation there will be two “VCS”: this VCS explained here, which will always be referred to as “our VCS” or “Evo VCS”, and a VCS where application code and Evo migrations and tests should be placed, referred to as “external VCS”.

Evo JDBC is the other component of our solution. The goal of the JDBC is to connect a SET to the Evo VCS to catch all SQL statements resulting from the interaction between the developer and the SET.

4.2 Evo VCS Objects

Statement is any SQL operation, either Data Definition Language (DDL) or Data Manipulation Language (DML).

Filter is a regex string that will be evaluated against every single statement that arrives at the VCS from JDBC. If some filter matches some statement, that statement is ignored. If not, the statement will be appended to the staged migration. The purpose of this is to avoid that meaningless statements made by the SET (like SHOW or SELECT) are stored.

Staged Migration is a file where all statements that come from JDBC and pass through the filters are stored. This file is used to work in a future migration and separate the migration that the developer is currently working on and all other migrations.

Migration is a file with a group of SQL statements that is ready to be shared with others (example in listing 4.1). This means that it can be executed everywhere, including production sandboxes. This must have a migration type, a version and optionally can have a description. Migrations should be seen as closed and should not be changed. However, some situations will be described below where they can be changed (explained in section 4.6). To verify if a migration has changed, each migration has a checksum. There are three types of migrations:

- **Regular Migration**. Migration that transforms one valid state of the database into another valid state. Each regular migration has a version. This version is used by Evo DB to know in which order regular migrations are performed. A higher version means a more recent migration. Filename format: V{*version*}__{*description*}.sql
- **Baseline Migration**. In a project with a very long lifetime it is normal to have many migrations (maybe hundreds or even thousands). If the developer needs to create a new up to date database, she will have to run all project migrations from the beginning. Probably the executed operations would overlap. For example, in different migrations would exist CREATE, ALTER, and DROP operations related with the same table or the same column. To avoid all this extra work and time wasted performing unnecessary operations, there are baseline migrations. These migrations contain a set of statements that build the database exactly where it would be if the developer run all the regular migrations up to the one with the version that the baseline migration is associated with. Filename format: B{*version*}__{*description*}.sql
- **Deprecation Migration**. Migration that has a date in the future on which it must be executed (deprecation period). For example, some migrations involve creating structures and dropping others in a database. If there are several projects dependent on that database, it can happen that some projects have already updated the code to work with the new structures, but others will continue to rely on the old structures. To avoid this chaos, two migrations can be generated: (1) a regular migration with the creation of the new structures to be executed as soon as possible and (2) another deprecation migration with the deletion of the old structures to be executed on a defined date. Until that date, all applications will have to update their code to work only with the new structures. The purpose of having a version in the deprecation migration is to identify to which regular migration is associated with. We advise that each deprecation migration should be idempotent to avoid problems when applied. Filename format: D{*version*}__{*description*}__*date_to_execute*.sql

Listing 4.1: Example of a migration

```sql
/* using MySQL */
CREATE TABLE `account` (
    `account_id` BIGINT(20) NOT NULL,
    `email` VARCHAR(255) NOT NULL COLLATE 'utf8mb4_unicode_ci',
    `password` VARCHAR(255) NOT NULL COLLATE 'utf8mb4_unicode_ci',
    `name` VARCHAR(200) NOT NULL COLLATE 'utf8mb4_unicode_ci',
    PRIMARY KEY (`account_id`)
) COLLATE='utf8mb4_unicode_ci' ENGINE=InnoDB;

CREATE TABLE `restaurant` (
    `restaurant_id` BIGINT(20) NOT NULL,
    `account_id` BIGINT(20) NULL DEFAULT NULL,
    `name` VARCHAR(100) NOT NULL COLLATE 'utf8mb4_unicode_ci',
    `country` CHAR(2) NOT NULL COLLATE 'utf8mb4_unicode_ci',
    `postal_code` VARCHAR(255) NOT NULL
        COLLATE 'utf8mb4_unicode_ci',
    `phone_number` VARCHAR(40) NOT NULL DEFAULT ''
        COLLATE 'utf8mb4_unicode_ci',
    PRIMARY KEY (`restaurant_id`),
    INDEX `FK_restaurant_account` (`account_id`),
    CONSTRAINT `FK_restaurant_account` FOREIGN KEY (`account_id`)
        REFERENCES `account` (`account_id`)
        ON UPDATE CASCADE ON DELETE CASCADE
) COLLATE='utf8mb4_unicode_ci' ENGINE=InnoDB;
```

Test (example in listing 4.2) is a group of SQL statements that test some functionality of the database (e.g: state of the schema, referential integrity, stored procedures, triggers, etc). All SQL statements are executed inside a transaction, which is rolled back after the last statement to avoid that insertions, updates, or deletions that occurred during the test are persisted. This type of test only expects the usage of DML statements. Each test must end with a SELECT statement that returns a row with a single column containing a 1 (or true) if the test was successful or a value different of 1 (or false) if the test was unsuccessful.

Each migration is associated with a test group to ensure that all versions of the database are tested.

Even staged migration has its tests to allow a test-first or test-driven approach while developing. Our tests have a limitation, they don't allow the developer to test situations where an exception is expected. For example, when testing a foreign key, we want to insert one row referring to another that does not exist. Only if this operation fails, do we know for sure that the foreign key exists. Our tests do not allow such scenario.

Listing 4.2: Example of a test

```sql
/* using MySQL */
INSERT INTO `account` (`account_id`, `email`, `password`, `name`)
VALUES ('1', 'test@test.com',
    '$2y$12$tySha5WBBob9BN6yDUX.BOPXobKnH6Qj4H7TmqDSSC2ZsdQq8pvvm',
    'Best Owner');
INSERT INTO `restaurant` (`restaurant_id`, `account_id`, `name`,
    `country`, `postal_code`, `phone_number`)
VALUES ('1', '1', 'Best Restaurant', 'Portugal', '1000-000',
    '+351961234567');
DELETE FROM `account` WHERE `account_id` = '1';

/* verify if CONSTRAINT is working as expected
(expecting delete all restaurants of this account because
CONSTRAINT has DELETE CASCADE) */
/* if !COUNT(*) equals to 1, success; else failure */
SELECT !COUNT(*) FROM `restaurant` WHERE `account_id` = '1';
```

Backup is a set of SQL statements that represents the database at a given time, a snapshot of the database. Filename format: backup_{*date_and_time*}.sql

4.3 Evo Data Structures

To maintain all those objects referred to in section 4.2, Evo DB uses two structures: the **Repository** and the **Migration History Table**.

4.3.1 Repository

A repository is a directory where project configurations, filters, staged migration, migrations, and tests are stored. A repository is created by the developer inside a folder, by issuing the **init command**. The repository can also be shared with others and, for that, the repository should be created in a folder that is being tracked by the external VCS.

When it is created, the following directories and files are created:

- **db_migrations**. The folder where migrations are stored.
- **db_tests**. The folder where tests are placed. Since each migration has its own tests, inside this directory will exist a directory for each migration with its tests stored in there.
- **staged**. Directory where staged migration is ("staged.sql" file) and tests related to the staged migration are ("staged_tests" folder).
- **evo.conf**. File where basic configurations related to the project are stored:
  - project_name: name of the project;
  - schemas: schemas that Evo DB will be responsible for. The first schema of this list is the default schema;
  - jdbc_url: url for internal JDBC;
  - jdbc_driver: full class name of internal JDBC;
  - db_user: username used to connect to the database;
  - db_password: password used to connect to the database;
  - db_hostname: host name where the database is.
- **filters.sql**. File that stores filters, separated by a semicolon (;). Example in listing 4.3.

Both *staged* folder and *evo.conf* file should not be shared with others. The *staged* folder has work that is not finished yet, so others should not see it, and the *evo.conf* file has some parameters that should be private.

The goal of this folder is to store all relevant artifacts for a project, being this folder the source of all truth for our VCS.

Listing 4.3: Example of filters.sql

```sql
/* Please separate filters by semicolons (;).
Each filter is a regex string. */
SHOW *;
SELECT *;
```

Column Name	Column Description
migration_type	Can be: - ”V”, which means regular migration; - ”B”, which means baseline migration; - ”D”, which means deprecation migration.
version	Higher the version, the further down the migration history. This is how Evo DB knows how to position migrations relatively to others. Version is composed of a number or numbers separated by dots.
description	Describes what is made in this migration.
filename	Name of the migration file.
checksum	The checksum used to validate if the content of the migration file is still the same.
executed_at	When migration was executed.
is_current	If 1, it means that this migration is the one the database is in.
is_target	If 1, it means that this migration is marked as the target.
last_executed_stmt_pos	Relative position of the last executed statement inside the migration.
last_executed_stmt_line	Line of the file of the last executed statement.
last_executed_stmt_first_char	Position of the first char in the line of the last executed statement.
last_executed_stmt	Last executed statement.
success	Marks if the migration was successful or not when executing.

Table 4.1: Columns of Migration History Table

4.3.2 Evo Migration History Table

This table named *__evo_migration* is stored in the default schema of the project and its goal is to store data related to each local migration. The columns of the migration history table are in table 4.1. It should be noted that most columns exist for internal use and some others exist to inform the developer of the state of the project if something goes wrong or if there is the need to audit what happened in the project until now.

4.4 Support Concepts

Local Migration is a migration that is locally and Evo DB has already registered it in the migration history table.

Migration History is the set of all local migrations.

Migration Version is a version that is used to position the migration in the migration history. It can be a number or several numbers separated by dots(.) or underscores (_).

Checksum is a digest generated by applying the SHA-256 algorithm to the migration type, version, description and content. The goal is to know if the migration changes over time. SHA-256 is a secure cryptographic hash algorithm, which means that "it is computationally infeasible to find two different messages that produce the same message digest" [7].

"Past" is all migrations that have a lower version than the current migration.

"Future" is all migrations that have a higher version than the current migration.

4.5 Evo JDBC

The purpose of our JDBC is to **intercept the successful statements** that a SET issues to the RDBMS. This means that has to behave as a normal JDBC that allows the SET to communicate with the RDBMS and, at the same time, stores the statements executed successfully in the *staged/staged.sql* file (staged migration).

This means that our JDBC must know how to communicate to an RDBMS. However communicating with one RDBMS is a difficult task, because the JDBC must implement the communication protocol of this RDBMS at the bit-level. The problem gets worse if we want to communicate with multiple RDBMSs because now we must implement the communication protocol with multiple RDBMSs. This is too difficult to implement well.

So, our solution is simple: since there are already lots of well-developed, well-maintained, and mature open-source JDBCs for all major RDBMSs, we simply use them internally to communicate to the required RDBMS. We call this JDBC that we use internally the internal JDBC.

The internal JDBC must be installed and placed on the *classpath*. Then, when the developer uses our JDBC, she must provide which internal JDBC Evo DB can work with. The internal JDBC is provided through the JDBC URL.

For example, let's imagine that the intended internal JDBC URL is the following:

```
jdbc:mysql://localhost:3306/db
```

The URL that the developer must set up in the SET is the following:

```
jdbc:evo:<project_name>:mysql://localhost:3306/db
```

Basically, between "jdbc:" string and the rest of the internal JDBC URL it should be "evo:<project_name>".

4.5.1 Connection with Evo Projects

Since it can exist several projects in the same sandbox, the JDBC must know to which project the intercepted statement belongs. Both internal JDBC and project name are provided in the JDBC URL.

So, the JDBC has to have a dictionary that maps the names of projects to the paths of projects. When the **init command** is executed, an entry in this dictionary is created connecting the name the developer provides to the directory where the command is executed.

4.6 Evo Architecture

4.6.1 The Beginning

Everything starts with the process of **creating a repository** with our VCS by running the following command:

```
$ evo init --project-name <project_name> --schemas <schemas_separated_by_commas> \
    --jdbc-url <internal_jdbc_url> --jdbc-driver <internal_jdbc_driver_class_name> \
    --db-user <db_user> --db-password <db_password> --db-hostname <db_hostname>
```

This process creates the *default_schema* (first schema on the list of schemas), if it does not exist in the database, and creates there the migration history table. It creates the repository of the project in the directory where the command was executed, including the first migration, which content is the SQL statements to generate the *default_schema* and other schemas that are already exist and were referred in the command. And it also registers this first version in the migration history table. This migration is placed in *db_migrations* folder with the name "B0__create_initial_schemas.sql". For the other schemas that were referred to and do not exist, Evo DB expects that the developer will create those schemas later.

This first migration is obtained through **mysqldump** or **pg_dump**, depending on whether the RDBMS used is MySQL/MariaDB or PostgreSQL.

There is one last thing this command does: make the JDBC driver aware of our repository, to allow it to send the intercepted statements to the correct staged migration.

4.6.2 New Migrations

With the repository and migration history table created, the developer can start to develop new migrations, which are the working units to make the database evolve from one state to another. To generate new migrations, the developer can have two approaches:

- **Working in staged migration and then transforming it into a migration (recommended)**. To develop a new migration, the developer can work through a SET (step 1 of the fig. 4.2) by using the Evo JDBC connected to it. For that, she must provide the following JDBC URL to the SET:

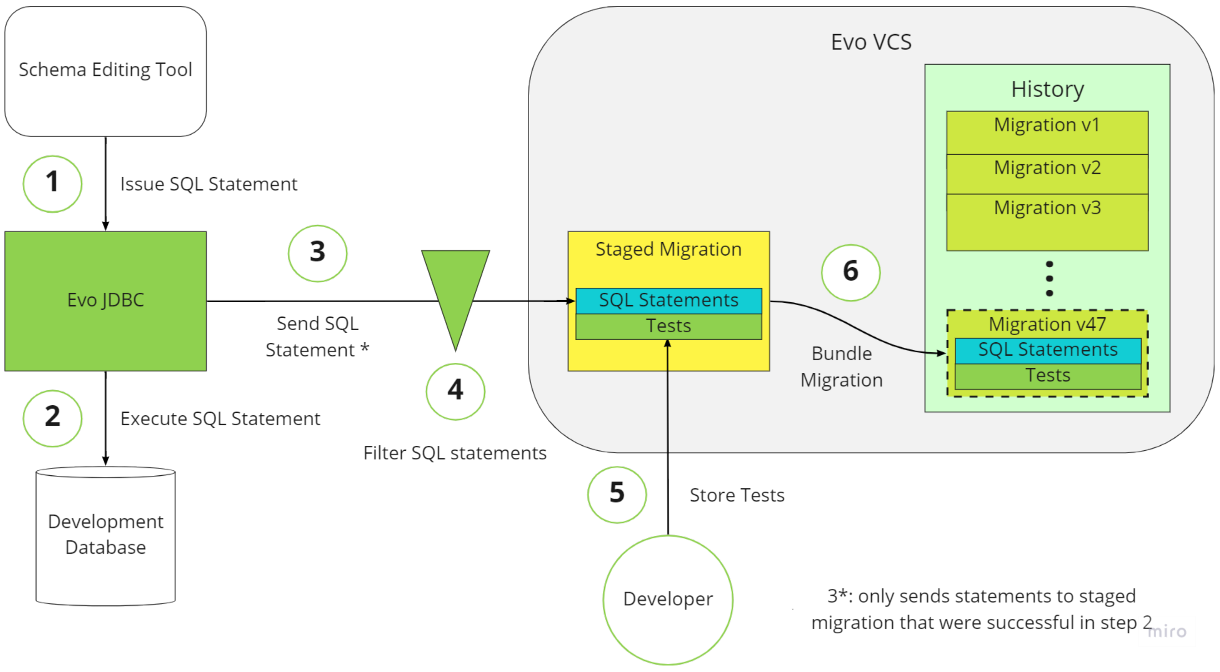


Figure 4.1: Recommended Development Process with JDBC and SET

```
jdbc:evo:<project_name>:mysql://localhost:3306/db
```

From what we found, the developer can use at least these three SETs with our JDBC: DBeaver, DbVisualizer, and DataGrip.

Evo JDBC sends all statements to the database (step 2) and then sends the successful ones to the staged migration (step 3).

Before storing the statements in the staged migration, the Evo JDBC filters them (step 4). This step exists because the SET generates several useless statements, like SHOW or SELECT statements, that are responsible for getting the structure of tables, views, and other objects as well as data. So, the developer can add filters to *filters.sql* to block what she thinks as useless statements. However, the developer should not set too tight filters to not lose important statements. The developer can also set no filters and then go to the staged migration file directly and remove the statements that are not needed.

While developing the staged migration, the developer can also add staged tests (step 5). Later, when the staged migration looks finished, the developer can transform the staged migration into migration (step 6) by executing:

```
$ evo bundle --version <migration_version> --description <migration_description>
```

This command tests the staged migration and, if all tests are successful, creates a new migration with the content of the staged migration and all staged tests are copied for

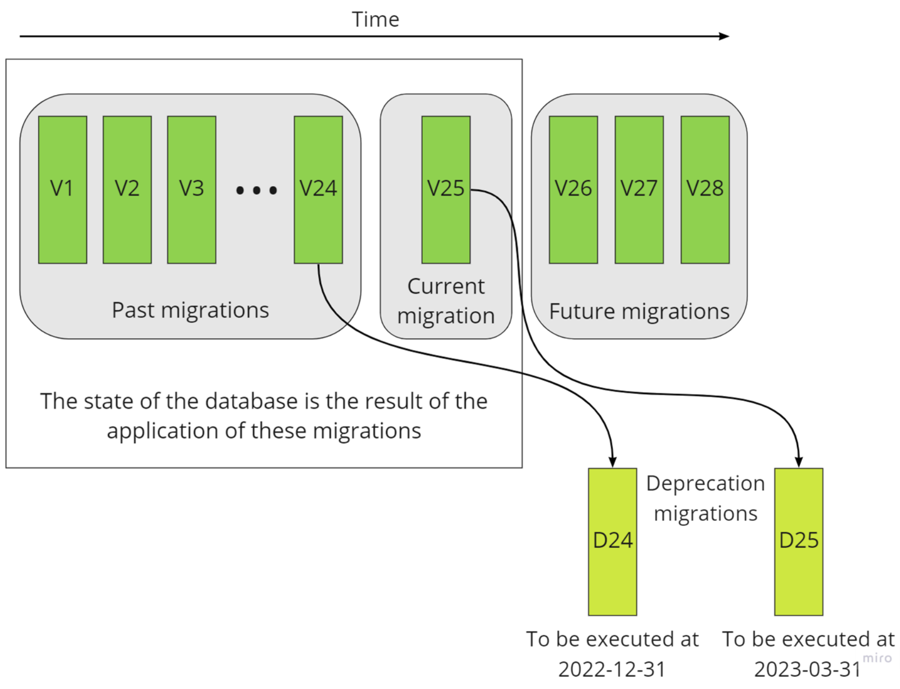

Time
V1
V2
V3
V24
Past migrations
V25
Current migration
V26
V27
V28
Future migrations
The state of the database is the result of the application of these migrations
D24
Deprecation migrations
D25
To be executed at 2022-12-31
To be executed at 2023-03-31
miro

Figure 4.2: Visualization of Past and Future Migrations

db_tests/<migration_version>. After it, the *staged.sql* file is truncated to allow the developer to start working on a new migration right away. After this process, Evo DB adds this migration to the migration history table.

- **Creating a new file**. Inside *db_migrations* folder, the developer creates a file with a name with the right structure for a migration where the developer inserts whatever SQL statements she wants. She can also create the folder *db_tests/<migration_version>* and create the tests there.

4.6.3 Current State of the Project

Every time the developer wants to see the real current state of the project, she can use:

```
$ evo status
```

This command provides the following information:

- **Current migration**. The last migration applied to the database.
- **Conflicts**. There are two types of conflicts: (1) migration file with the same type and version as another migration file and (2) migration file with the same type and version as some registered migration in the migration history table, but with different checksums.
- **New past migrations**. Migration files that were added to the "past", which means before the current migration.

- **Lost past migrations**. Migration files that were removed from the “past”.
- **New future migrations**. Migration files that were created with a higher version than the current migration (“future”).
- **Lost future migrations**. Migration files that were deleted in the “future”.
- **Deprecation migrations**. This command provides which deprecation migrations exist, separating the ones that should have already been executed and the ones that the deprecation period is not over yet.

This command is totally dependent on the current state of the sandbox the command is run on. For example, if the developer wants to compare the current state of the project with the current state in production to evaluate if there are conflicts or changes in the “past”, the developer must (1) clone the Evo repository that is in production for her sandbox, (2) deploy (or redeploy) it up to the migration in production, (3) update the Evo repository and (4) execute the **status command**.

The changes in the “future” are shown just to give better feedback to the developer. Just conflicts and changes in the “past” (lost and new past migrations) are dangerous because they change the history already applied.

4.6.4 Dealing with Conflicts and Changes in the Past

We strongly advise to not change the history already applied, at least the one in production. After committing a migration into the external VCS, it means that everyone can now execute that migration, even production sandboxes. So, every time the developer needs to change something in a previous migration, she should have a “roll-forward” approach, which means doing a new migration that fixes the problems of a previous one.

If something was really wrong and it could not go into production, first, the developer has to make sure that the migration was not executed in any production sandboxes, and just then she can change or remove the migration or even add a previous migration to this one. This will cause a conflict or a change in the “past”, but it is so simple to solve as:

```
// drop the database
$ evo teardown
// create the database and executes all migrations until the one with target_version
$ evo deploy [--target-version <target_version>] [--test-before] [--test-after] \
    [--deprecate]

// or she can use this command, which does the same as the two commands above combined
```

```
$ evo redeploy [--target-version <target_version>] [--test-before] [--test-after] \
    [--deprecate]
```

The developer can choose to which migration she wants to deploy to (<target_version>) or deploy simply to the last migration (migration with the highest version) by not providing the <target_version>. She can also execute this with some test flags to run also tests, both before (–test-before) and after (–test-after) the deployment. By passing the –deprecate flag, all deprecation migrations that should have already been executed (those in which the deprecation period is over) are executed.

By running those commands, any developer has a fresh copy of the database with a "new history", which contains all migrations that are in the *db_migrations* folder.

4.6.5 Deployment in Production

Now that migrations are created and committed and all conflicts and changes in the "past" are solved at the development, it is time to perform the deployment in the production sandboxes. Deployment is the process of applying one or more migrations in a sandbox.

A big important rule is to never mess up the history in production. In production sandboxes, this is the only command that should be executed:

```
$ evo deploy [--target-version <target_version>] [--backup] [--restore-backup] \
    [--test-before] [--test-after] [--deprecate]
```

As we can see, there are two more optional flags that are only important in production: (1) –backup flag which indicates to Evo DB to do a backup before the migration process starts and (2) –restore-backup flag which informs Evo DB to restore a backup if something goes wrong with the deployment.

When Evo DB is asked to do a backup, it is going to use **mysqldump** or **pg_dump** again and the result that came from there is going to be stored in a file in the *backups* folder with the following filename: backup_*yyyy_MM_dd_HH_mm_ss*.sql

This format is important because we recognize that our way to obtain a backup is good for small databases or sandboxes that have offline periods or at least low traffic periods, but it is not good for big databases or sandboxes that must be always online. So, we allow that backups can be added by the developer to be used later when a backup must be restored.

The backup selected by Evo DB when restoring a database is always the one that has a higher date and time in the filename.

Just as a note, we do not advise anyone to run tests on a production sandbox unless they are sure that the tests only use records inserted by the tests, that they do not change any data other than the data inserted by the tests, and do not expect results based on all records present in the database (COUNT(*) operation for example).

4.6.6 Deployment in a Database Not Known By Evo DB

Until now, the deployment was described as being made in a database that Evo DB already knew, which means that there was already a migration history table.

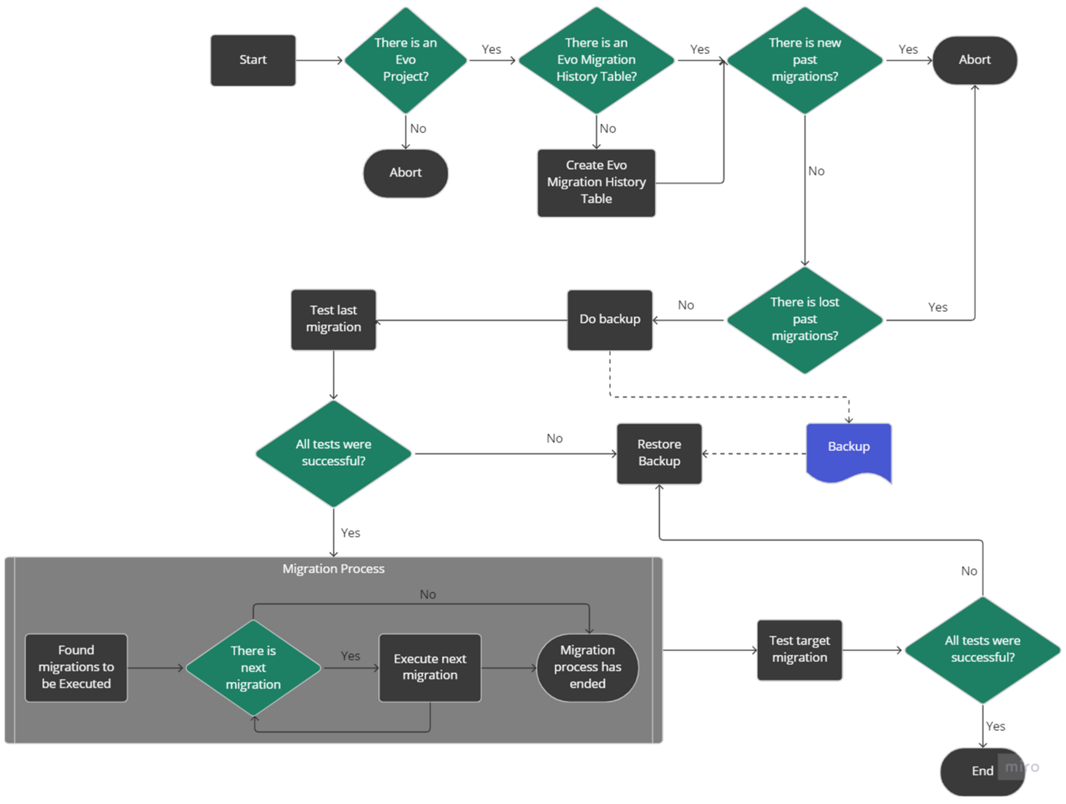
However, not all deployments are made in a known database. There are two cases of such a situation:

- **No previous database**. It happens that maybe a new production sandbox is being set up. In this case, just execute the **deploy command** as shown in section 4.6.5.
- **Database already exists but it is not known by Evo DB**. For example, some project that exists for some time now starts using Evo DB to track, control and manage their database migrations. It is expected that the project has a database already with a bunch of tables and procedures and other objects. In this case, it is expected that the state of the production database is the point of start of Evo DB. So, the suggested flow is to copy the production database into a development sandbox, remove the data generated by the application, and then do exactly what is explained in section 4.6.1.

4.6.7 While The Deployment Process Is Running

The deployment process has several steps (in fig. 4.3) which validate the migration history, perform backups and testing, and, of course, migrate between the current migration and the target migration. The steps and their order are as follows:

1. **Verify the existence of an Evo project**. If there is no Evo project where the **deploy command** was executed, the process aborts.
2. **Verify the presence of the migration history table**. If this table is not present, this process creates it.
3. **Verify conflicts**. Compares all migration filenames between them and then compares the checksums of migration files with the checksums found in the migration history table that have the same type and version in order to identify files that have changed their SQL statements or description. If it is found at least one conflict, this process is aborted.
4. **Verify new past migrations and verify lost past migrations**. Evo DB analyzes the migration history table and tries to find if there is any new migration file or if there is any lost migration file in the "past". If there is any, the deployment process is aborted.
5. **Do backup**. If –backup flag is on, at this time Evo DB does a backup of all the schemas controlled by it.



Start
There is an Evo Project?
Yes
No
Abort
There is an Evo Migration History Table?
Yes
No
Create Evo Migration History Table
There is new past migrations?
Yes
Abort
No
There is lost past migrations?
Yes
No
Do backup
Test last migration
All tests were successful?
No
Restore Backup
Backup
Yes
Migration Process
No
Found migrations to be Executed
There is next migration
Yes
Execute next migration
Migration process has ended
Test target migration
All tests were successful?
No
Yes
End
miro

Figure 4.3: Flowchart of the deployment process

6. **Test last migration applied**. If –test-before flag is on, Evo DB executes the test of the last migration applied before advancing. If some test fails, the process aborts.

7. **Add to history new future migrations and delete lost future migrations**. This type of migration does not harm the history already applied so Evo DB updates the migration history table accordingly.

8. **Migration process**. This process found which migrations have to be executed, from the migration following the last applied up to the target migration, and executes them. The execution has the following steps:

    1. Set some migration as target (is_target = 1). –target-version parameter is responsible for setting the target migration. When it is not provided, the migration with the highest version is set as the target migration. This step marks the beginning of the migration process.

    2. Executes the next migration:

        1. Fetch the next SQL statement to execute.
        2. If it is a DML statement, start a new transaction.
        3. Execute the statement.

    4. Store in migration history table the relative position of the statement in the migration (column last_executed_stmt_pos), the line of the statement inside the migration (column last_executed_stmt_line), the first character position in the line (column last_executed_stmt_first_char), and the statement itself (column last_executed_stmt).
    5. Commit the transaction if some transaction was started before.
    6. Go to the step 8.2.1 if there are more statements. Otherwise, this migration is over.
  3. Sets migration as successful (success = 1) and as the current migration (is_current = 1).
  4. If the migration is the target one, is_target is set to 0. With that, Evo DB knows that migration is over and moves to step 9. Otherwise, goes to step 8.2.

  The purpose of the transaction is to guarantee that DML statements are executed and Evo DB writes it to the migration history table or any of that happens. This is made only for DML statements because these ones are transactional in almost all RDBMS, as opposed to DDL statements.

  From our point of view, this is enough because it is very difficult to understand by looking at the schema or data if DML statements were already applied, unlike DDL statements, which we can understand if they were executed just by looking at the schema.

  If the statement following the last recorded statement is a DML, it automatically means that everything is correct. If it is a DDL, the developer will have to look at the schema and figure out whether it has already been performed or not. If it has, she must update the information in the migration history table. If not, everything is fine.

  This is important for the rare and special case that if some statement from the migration is executed and our statement to update the last executed statement fails.

9. **Apply tests associated with target migration**. After executing the migration process, tests related to the target migration are executed. If some test fails and –restore-backup flag is on, the most recent backup is restored.

4.6.7.A In Case of Failure

If something fails before the migration process starts, the developer is warned but nothing more happens. After the migration process has started:

- **Syntax error or a logic error happens**. If the –restore-backup flag is on, the backup is restored. If not, the developer is notified that an error occurred in a specific statement and she must correct the error. She goes to her sandbox, does the changes required, and redeploys the project. When everything is correct, she sends those changes to the production sandbox, updates the checksum, and then restarts the deployment process, by calling the **deploy command** again.

- **Tests fail**. In this case, they have to fix the test, if the test has some mistake, and commit it or do a ”roll-forward”, which means creating a new migration with the fix of the error detected in the test.
- **Power outage**. When the sandbox comes to life again, the developer must attest herself manually that the last executed statement is right, and if wrong, put it correctly. Then issues again the **deploy command** and Evo DB will start from where it left off.

4.7 What RDBMSs We Support?

Currently, Evo DB only supports MySQL, MariaDB, and PostgreSQL. We tried to create the most abstract solution possible to work with all RDBMSs, but we encounter two problems:

- **Create baseline migration or backup**. When we want to create a baseline migration or a backup, we use some utilities provided by the RDBMSs vendors. In this case, we use **mysqldump** and **pg_dump** utilities. We do not found any other utilities to do these processes in other vendors.
- **Vendor-specific SQL statements**. Although SQL is a standard, each vendor ends up implementing functionality in its own way. This means that sometimes to do the same operation we have to write different SQL statements depending on the RDBMS. And we need to interact with the database directly with our statements because the migration history table, the teardown command, etc. So we limited ourselves to MySQL, MariaDB, and PostgreSQL because they were the easiest RDBMSs to test since they are open-source.

5 Evaluation

Contents

In this chapter, we present the results generated by our solution. We divide it into two parts: a quantitative and qualitative evaluation. In section 5.1, we show how our solution compares with an "homemade process" for migrations and what are the justifications for the results. In section 5.2, we evaluate which techniques and principles our solution allows against the ones referred to in the chapter 2.

5.1 Quantitative Evaluation

In this section, we focus our attention on quantitative evaluation. We intend to understand what overhead is caused by our solution in the process of executing one migration when compared with a more homemade process. This is made by measuring the time spent by our solution and by a homemade process.

5.1.1 Experimental Setup

The experiments were performed on an ASUS N551JX, with an Intel Core i7-4720HQ of 4 cores with 2.60GHz and 12GB of main memory (RAM). The operating system is Windows 10, version 21H1.

5.1.2 Homemade Processes

Homemade processes are what we call any type of migrations that are created and executed without the help of any database migration tool. We described those processes in section 1.1. Almost all homemade processes we can think of can be translated into the creation of a script that is then run directly in a SET or a database client.

So, to simulate a homemade process, we did a simple Java application that only gets one database connection and runs all statements from the migration, which is similar to what is described in [16] as "Automated Schema Updates" technique. It is done this way because it allows us to measure the time, which would be difficult to be done with a SET or a database client.

We also describe in section 1.1 some "live" processes. We do not use them in this comparison because several steps are too dependent on the person who is performing them, making the values obtained meaningless for this analysis.

5.1.3 Obtaining Results

We used one table with 973.104 rows generated from the Geonames site dataset, more specifically, from alternate_name.zip[1]. In each case described below, we first execute ten times the homemade process and then execute ten times the Evo DB migration, through the **deploy command**.

[1] http://download.geonames.org/export/dump/

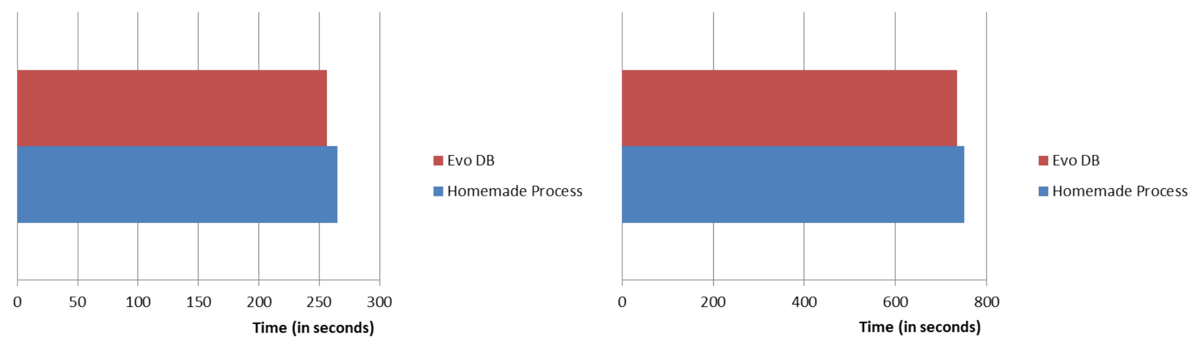


Figure 5.1: Execution of Complex DDL (left) and Complex DML (right)

5.1.4 Extra Control Operations

Evo DB does a few more operations to provide its features:

- adds new future migrations (simple INSERT for each future migration added);
- deletes lost future migrations (simple DELETE for each future migration lost);
- set a future migration as a target (simple UPDATE) - marks the beginning of the migration process;
- at the end of the execution of each statement, updates the last statement (simple UPDATE for each statement);
- when a migration ends, it is set as successful, and as the current migration (simple UPDATE for each migration);
- unset the target migration when all migrations have ended (simple UPDATE).

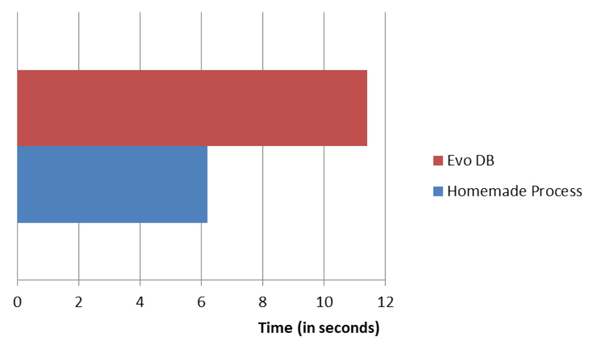
5.1.5 Results

We separate the results by migrations with two types of statements:

- **Complex statement**. SQL statement that takes 1 second or more to execute.
- **Simple statement**. SQL statement that takes less than 1 second to execute.

5.1.5.A Complex Statements

By analyzing the bar charts (in fig. 5.1), we conclude that both ways take almost the same time to execute the migrations. Interestingly, Evo DB is slightly faster, which can be explained by some unrelated processes being executed by the operating system or database memory management.



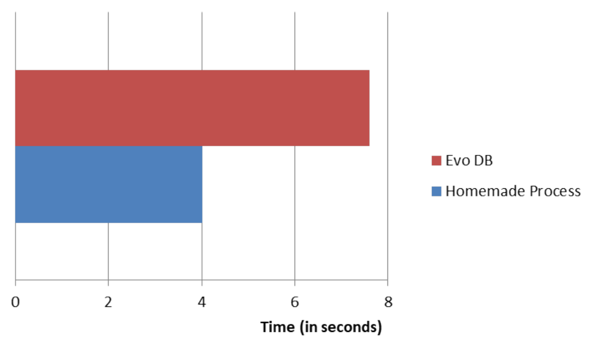


Figure 5.2: Execution of Simple DDL (left) and Simple DML (right)

With these results, we can state that the extra control operations made by Evo DB do not affect substantially the execution time, since they are extremely faster to perform compared with the migration operations.

5.1.5.B Simple Statements

Our results say that Evo DB approximately doubled the execution time of the homemade process, both DDL and DML.

Per statement in the migration, Evo DB executes one simple UPDATE. So, if we execute an arbitrary number of simple statements, Evo DB will execute approximately twice as many simple operations. In our point of view, this is the justification for the results shown in fig. 5.2.

5.1.6 Unconsidered Constraints

In this evaluation, we do not consider:

- **Execution of migrations in production**. We do not execute the processes already described in a database that is being accessed by other applications. All processes were executed in an offline database, which means that the only SQL statements that the database was processing were the ones generated by us.
- **Network**. The database we used was on localhost, but usually, the database is running in the cloud separate from the applications that access it.

5.1.7 Evo DB Real Impact

The impact caused by Evo DB is very small, almost insignificant. The Evo DB extra control operations are performed so quickly that only migrations with simple statements are affected significantly. However,

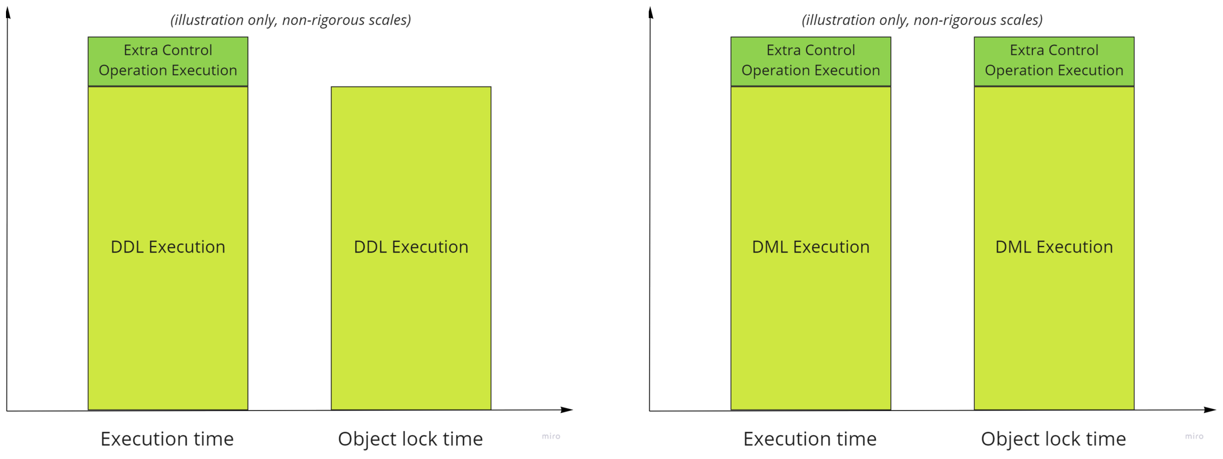

(illustration only, non-rigorous scales)
Extra Control Operation Execution
DDL Execution
DDL Execution
Execution time
Object lock time
(illustration only, non-rigorous scales)
Extra Control Operation Execution
Extra Control Operation Execution
DML Execution
DML Execution
Execution time
Object lock time

Figure 5.3: Object Lock Time using Evo DB - DDL (left) and DML (right)

these migrations take such a short time to execute that it is not relevant for systems that have been taken offline to be updated.

It is also important to take into account that the extra control operations only affect our migration history table, which implies that only our table is locked and no other tables are locked. Even with the transaction that happens when a DML statement is issued, the time spent by the database when executing our simple UPDATE is so small that the object locked by this DML statement is locked for just a few more milliseconds (illustrated in fig. 5.3).

5.2 Qualitative Evaluation

In this section, we analyze the best practices we found about database development and deployment and see if our solution answers those best practices or not, explaining why after. This evaluation is also represented in table 5.1.

5.2.1 Evolutionary Database Design Techniques

Database Refactoring. Our solution enables database refactoring since we allow the insertion of SQL statements into migrations, via SET or manually.

Evolutionary Data Modeling. We do not provide any feature to enable this technique.

Database Regression Testing. In our VCS, tests can be created and executed every time there is a deployment. Every single migration has its own tests, which allow testing of any version of the database.

Configuration Management of Database Artifacts. When creating an Evo project, the developer can put our repository inside a repository controlled by their external VCS.

Developer Sandboxes. Since our tool allows easy deployment and teardown processes, it is simple to create and recreate the database in any sandbox. Then, migrations combined with the file-sharing properties of the external VCS make it easy to set up any sandbox. Just update the content of the project controlled by the external VCS and deploy the migrations needed.

5.2.2 Continuous Database Integration Practices

Automate the build. The developer can integrate easily our automatic process of creating or updating a database in the overall build process. Just use the deploy command.

Put everything under version control. The repository of the Evo project just needs to be inside a repository controlled by the external VCS.

Give developers their own database copies. Since it is so easy to create and recreate the database, each developer can have her database copy without the help of a DBA.

Automate database creation. Just run the deploy or redeploy command and the database is created and updated up to the migration the developer wants. Totally automated.

Refactor each database individually. Our solution enables the software team to apply migrations in one database and leave the others intact. All operations performed by the Evo VCS are made strictly with the connection provided by the user in the *evo.conf* file.

Adopt a consistent identification strategy. Evo VCS allows the developer to give the version they want it to. If she wants an incremental version, she can make it. If she wants a semantic version, she can make it either.

Bundle database changes when needed. If for some reason, multiple migrations must be executed, our solution can bundle them and execute them all at once.

Ensure that the database knows its version number. Through the migration history table, the database knows always its state. The database not only knows which migration/version is but also which statement was last executed.

5.2.3 Important Scripts to Maintain

According to Ambler, there are three scripts to maintain to manage well database migrations:

- **Database change log**. Regular or Baseline migrations store all DDL operations;
- **Data migration log**. Regular or Baseline migrations also store all DML operations;
- **Update log**. Deprecation migrations do the same as the update log.

	Evo DB	Flyway	Liquibase	DBDeploy
Schema Editing Tool Integration	●			
Evolutionary Database Design				
Database Refactoring	●	●	●	●
Evolutionary Data Modeling				
Database Regression Testing	●			
Configuration Management of Database Artifacts	●	●	●	●
Developer Sandboxes	●	●	●	●
Continuous Database Integration				
Automate the build	●	●	●	
Put everything under version control	●	●	●	●
Give developers their own database copies	●	●	●	●
Automate database creation	●	●	●	
Refactor each database individually	●	●	●	●
Adopt a consistent identification strategy	●	●	●	●
Bundle database changes when needed	●	●	●	●
Ensure that the database knows its version number	●	●	●	●
Scripts to Maintain				
Database change log	●	●	●	
Data migration log	●	●	●	
Update log	●			
Deployment Process				
Back up the database	●		●	
Run previous regression tests	●			
Deploy database refactorings	●	●	●	●
Run current regression tests	●			
Back out if necessary	●			

Table 5.1: Techniques, practices, processes and migrations that Evo DB, Flyway, Liquibase, and DBDeploy allow

5.2.4 Deployment Process

Back up the database. By providing the –backup flag in the deploy command, a backup of the database is made. It is left entirely to the team to decide if they want a backup or not.

Run previous regression tests. There is the –test-before flag to trigger the testing process of the previous migration.

Deploy database refactorings. Our deployment process executes all migrations from the last executed, exclusively, up to the requested one or until the last one, if there is no target migration.

Run current regression tests. To trigger this, just pass the –test-after flag to the deploy command.

Back out if necessary. It can be passed the –restore-backup flag to restore the latest backup. This backup can be created by providing the –backup flag or it can be given by someone in the team.

6

Conclusion

Contents

Throughout this document, we described the importance of the development with Schema Editing Tools (SETs), the need to bring an Evolutionary Database Design to SETs, and how to make a solution that is compatible with Continuous Integration (CI) and Continuous Database Integration (CDBI). Then we describe what are the tools that already exist out there in the market. After it, we detailed how our solution is implemented and how the user can interact with it and we finish by stating the results of our work.

6.1 Conclusions

As we explained at the beginning of this document, we want to solve the problem of SETs users who cannot integrate their way of working with approaches and tools that enable a more evolutionary development, which makes it difficult for them to practice CI.

As we solve the problem, we show that SETs do not need to be attached with serial/waterfall approaches. In fact, they can be combined with an Evolutionary Database Design and CI. Due to the simplicity of working with SETs, it seems possible that they can enhance the quality and speed with which databases are developed. For us, it seems that the industry does not understand this yet, which may be the reason why this problem is not solved.

So, in order to solve the problem, we created two main pieces:

- **A JDBC driver** that intercepts the successful statements issued by a SET and sends them to our VCS.
- **A VCS** where the user can interact with the statements that came from the SET and store them in an organized way, in order to allow database versioning.

Now, developers do not need to choose database versioning over development with a SET anymore or vice-versa. They can have both. Sharing the work done in a SET with other sandboxes is now easy and effortless and deployments are also easy, which allows shorter development and deployment cycles. With database versioning and then testing, our solution also allows CI and CDBI. As a bonus, more developers can start to interact more with relational databases, even with no previous SQL knowledge.

The behavior that we provide with our solution is already expected for years by some folks in the community. Just look to what is described by DBDeploy's creator as his "Nirvana" in its paper about DBDeploy [4]:

"In my SQL client GUI I right click on a column and select the "Add index. . . " refactoring option. I am then presented with a dialog that allows me to name the index and then maybe adjust some of the DBMS-specific physical properties. I add a comment to summarize the refactoring and then click the "Save refactoring" button on the dialog. Save refactoring knows where my refactoring scripts live, it

calculates the next available script number and writes it to the file system. I then click the ‘Run database build’ button in my SQL Client – this performs the verification for me and ensures that my refactoring is fit for addition to the SCM repository. My final step is to click “Check in”.”

Of course, our tool does not fit perfectly the description provided, but our way of work and the described way are undeniably similar.

For how it works and the scenarios we predicted, we really think this tool can become a new standard for how SETs are expected to work.

6.2 Future Work

Our solution has a lot of space to grow. We leave it here some ways to expand our work:

- **Generate diagrams** based on the state of a database after running all migrations up to the given one, allowing a simple way to generate documentation for the database and an easy way for all project stakeholders to understand the database schema.
- **Enable undo migrations**, both manual and automatic, in order to be able to provide rollbacks.
- **Enable generation of migrations directly to a Flyway project or a Liquibase project**, in order to avoid developers having to change the database migration tool they are already familiar with.
- **Complete transactional and online migrations** through, for example, replication of structures, with the goal of being able to revert any migration without losing data and being able to perform the migrations without stopping the database.
- **Expand the number of SETs possible to work with**, other than the ones that allow JDBC connections.
- **Realize a study about the impact that our solution can have on the work of developers/teams**, both that currently work with a SET or not.

Bibliography

[1] Scott W. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley Publishing, 1st edition, 2003. ISBN 0471202835.

[2] Scott W. Ambler. Test-driven development of relational databases. *IEEE Software*, 24(3):37–43, May 2007. ISSN 1937-4194. doi: 10.1109/MS.2007.91.

[3] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006. ISBN 0321293533.

[4] Nick Ashley. Taking control of your database development. 2008. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.8126&rep=rep1&type=pdf`.

[5] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, June 1998. ISSN 0360-0300. doi: 10.1145/280277.280280. URL `https://doi.org/10.1145/280277.280280`.

[6] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *In ICEIS*, 2008.

[7] Quynh Dang. Secure hash standard, August 2015.

[8] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321336385.

[9] Phil Factor. Managing database changes using flyway: an overview, January 2021. URL `https://www.red-gate.com/hub/product-learning/flyway/managing-database-changes-using-flyway-an-overview`.

[10] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 0201485672.

[11] Martin Fowler. Continuous integration, May 2006. URL `https://www.martinfowler.com/articles/continuousIntegration.html`.

[12] Martin Fowler and Pramodkumar J. Sadalage. Evolutionary database design, May 2014. URL https://www.martinfowler.com/articles/evodb.html.

[13] Manik Jain. Automated liquibase generator and validator(algv). *International Journal of Scientific Technology Research*, 4:248–256, September 2015.

[14] Joris Kuipers. How to do database migration with flyway?, December 2017. URL https://www.youtube.com/watch?v=NDlZ6fP4X7s.

[15] David C. Kung, Jerry Gao, Pei Hsia, Lin, Jeremy., Toyoshima, and Yasufumi. Class firewall, test order, and regression testing of object-oriented programs. October 1993. URL https://www.researchgate.net/profile/Jerry-Gao/publication/274080160_Class-Firewall/links/55155f0b0cf2f7d80a32dc27/Class-Firewall.pdf.

[16] Thomas A. Limoncelli. Sql is no excuse to avoid devops. *Communications of the ACM*, 62(1): 46–49, January 2019. ISSN 0001-0782.

[17] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, USA, 2006. ISBN 0131495054.

[18] Walter F. Tichy. *Software Configuration Management*, page 1601–1604. John Wiley and Sons Ltd., GBR, 2003. ISBN 0470864125.

[19] Nathan Voxland. What is liquibase? what developers need to know about schema migration, September 2021. URL https://www.youtube.com/watch?v=Yxl1J0l3_M0.

[20] Steve Zadany. The magic of using xml changelogs in liquibase, August 2020. URL https://www.liquibase.com/blog/using-xml-changelogs-liquibase.