FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Tool for Incremental Database Migration

Fernando Jorge Alves

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Pascoal Faria

Co-Supervisor: Eurico Inocêncio

July 23, 2021

Tool for Incremental Database Migration

Fernando Jorge Alves

Mestrado Integrado em Engenharia Informática e Computação

July 23, 2021

Abstract

Cloudware S.A. develops accounting and management software used by over 120.000 active companies. These companies are distributed over several databases, and each of them has its independent database schema. With the current infrastructure, the database schema of all companies must be identical, meaning that any change to the logical model must be reflected in all individual company schemas.

To do so, Cloudware developers currently use two tools to perform database migrations: dbmate and Ruby on Rails' ActiveRecord. Currently, the process of creating migrations is manual and prone to human error, and therefore it needs to be enhanced.

As such, the goal of this dissertation was to implement a tool capable of automating further the process of creating database migrations, while preserving all of the functionality provided by the previous migration tools.

This tool was named NANDO and possesses all functionalities expected from a database migrator, such as the ability to create, apply and revert migrations. Besides these features, the tool also provides support to developers when creating migrations that either add/update functions to the database, or that make changes to multiple database schemas, by automating a large amount of the process.

When adding or updating functions to a database, the tool only requires the developer to specify the files that contain the functions being included in the migration. With this information, the tool is able to automatically fill the migration with all of the code necessary to update these functions. To do so, a system of annotations was implemented to keep track of the files that will be included in each migration.

In regards to migrations that perform changes on multiple database schemas, the tool provides a command for developers to compare two database schemas and determine their differences, as well as the respective SQL commands required to apply these changes to the production database.

NANDO was tested by several of Cloudware's developers in order to observe the time required to perform certain tasks when compared to the previous migration tools. During these tests, NANDO was considerably faster than the previous methods, with the most common tasks taking up to 81% less time to complete, on average. Besides performing tests that measured the relative performance of the tool, a survey was also developed to assess the usability of NANDO, especially when compared to the previous tools. This survey was administered to the developers that performed the previously mentioned tests, and the results were extremely positive.

Keywords: Database structure, Database migration, Version control

Resumo

A Cloudware S.A. desenvolve software de contabilidade e gestão utilizado por mais de 120.000 empresas ativas. Estas empresas estão distribuídas em várias bases de dados e cada uma delas tem seu esquema de dados independente. Com a infraestrutura atual, o esquema de base de dados de todas as empresas deve ser idêntico, o que significa que qualquer mudança no modelo lógico deve ser refletida em todos os esquemas individuais das empresas.

Para fazer isso, os desenvolvedores da Cloudware atualmente usam duas ferramentas para realizar migrações de base de dados: dbmate e o ActiveRecord de Ruby on Rails. Atualmente, o processo de criação de migrações é manual e sujeito a erros humanos, portanto precisa de ser melhorado.

Assim, o objetivo desta dissertação foi implementar uma ferramenta capaz de automatizar ainda mais o processo de criação de migrações de base de dados, preservando todas as funcionalidades fornecidas pelas ferramentas de migração anteriores.

Esta ferramenta foi chamada de NANDO e possui todas as funcionalidades esperadas de um migrador de base de dados, tal como a capacidade de criar, aplicar e reverter migrações. Além destas funcionalidades, a ferramenta também fornece suporte para desenvolvedores na criação de migrações que adicionam ou atualizam funções na base de dados, ou que fazem alterações em vários esquemas da base de dados, automatizando uma grande parte do processo.

Ao adicionar ou atualizar funções numa base de dados, a ferramenta requer apenas que o desenvolvedor especifique os ficheiros que contêm as funções que estão a ser incluídas na migração. Com essa informação, a ferramenta é capaz de preencher automaticamente a migração com todo o código necessário para atualizar essas funções. Para isso, foi implementado um sistema de anotações para identificar os ficheiros que serão incluídos em cada migração.

Em relação às migrações que realizam alterações em múltiplos esquemas da base de dados, a ferramenta fornece um comando para os desenvolvedores compararem dois esquemas da base de dados e determinar as suas diferenças, bem como os respetivos comandos SQL necessários para aplicar essas alterações nas bases de dados em produção.

O NANDO foi testado por vários desenvolvedores da Cloudware para observar o tempo necessário para realizar certas tarefas quando comparado com as ferramentas de migração anteriores. Durante esses testes, o NANDO foi consideravelmente mais rápido do que os métodos anteriores, com a maioria das tarefas necessitando, em média, de 81% menos tempo para serem concluídas. Além de terem sido realizados testes que medem o desempenho relativo da ferramenta, também foi desenvolvido um inquérito para avaliar a usabilidade do NANDO, especialmente quando comparando às ferramentas anteriores. Esse inquérito foi aplicado aos desenvolvedores que realizaram os testes mencionados anteriormente e os resultados foram extremamente positivos.

Keywords: Estrutura de base de dados, Migração de base de dados, Controlo de versões

Acknowledgements

I want to thank everyone who has, in one way or another, helped me complete this milestone.

Firstly, I want to thank my supervisor, Prof. João Pascoal Faria, for his guidance and assistance throughout this entire journey. His advice was invaluable in many of the decisions made, and without him, this dissertation would not have been possible.

Secondly, I want to show my utmost gratitude to Cloudware for proposing the topic of this dissertation in a subject that I genuinely enjoy. They also gave me the freedom to manage my schedule, which allowed me to continue working during the entire process without feeling overwhelmed. A special thanks to Eurico Inocêncio for being present every step of the way and always being available whenever I needed assistance. I would also like to thank every developer at Cloudware for their feedback and suggestions throughout the development phase and their availability and advice during the various testing phases.

Next, I would like to thank my family for their support during this last year and throughout my life. They allowed me to focus on this goal, and without them, I would not be where I am right now.

A special thanks to my friends for the help and support they provide every day and for the laughs we shared this last year. To those that were present during the late-night coding sessions and endless rants, I can't even begin to express my gratitude. I could not have asked for a better rubber duck.

Finally, I want to thank the Faculty of Engineering of the University of Porto for the skills I've learned in the past five years and for introducing me to some of the best people I know.

Fernando Alves

"There are only two hard things in Computer Science:
cache invalidation and naming things"

Phil Karlton

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
DBA	Database Administrator
DBMS	Database Management System
DSL	Domain Specific Language
SMO	Schema Modification Operator
NANDO	Nando AdmiNs Database Objects

Chapter 1

Introduction

1.1 Context and Motivation

Cloudware S.A. develops accounting and management software that is used by over 120.000 active companies. To accommodate for such a large user-base, these companies are distributed over several databases, and each of them has its independent database schema. This approach is essential to:

- Reduce the data size of the tables, thus improving performance
- Ensure profitability by allowing a larger amount of companies to share the same database server

All companies' database schema must be identical with the current infrastructure, meaning that any change to the logical model must be reflected in all individual company schemas. As such, Cloudware developers currently use two tools to perform database migrations: dbmate[1] and Ruby on Rails' ActiveRecord[2].

dbmate is simpler to use, as it allows developers to create database migrations using SQL code that will be executed directly. This means that developers create a migration script with only the SQL commands that will be executed, with no extra logic. However, this means dbmate cannot run migrations on all independent schemas since no SQL methods handle those scenarios, meaning it requires extra logic. As such, dbmate is only used in Cloudware's smaller projects.

ActiveRecord is a part of the Ruby on Rails[3] framework that follows the Model-View-Controller pattern. ActiveRecord corresponds to the Model, the layer responsible for the business logic. It allows developers to create database migrations using different templates and was expanded by Cloudware to run migrations on all independent company schemas. This means it has

[1] https://github.com/amacneil/dbmate
[2] https://guides.rubyonrails.org/active_record_basics.html
[3] https://guides.rubyonrails.org

a wider range of functionalities than dbmate, and for this reason, it is used on Cloudware's more complex projects, whose databases have independent company schemas.

With either of the tools mentioned above, to create a migration, a developer must implement two methods, `up` and `down`. The `up` method will contain the code for performing the desired change, and the `down` method will contain the code for reverting said change. This approach is followed to allow a change to be easily reverted in case a problem is detected.

Currently, this process of creating migrations is manual, tedious and prone to human error, and as such, it needs to be enhanced. The more common mistakes occur in the `down` method since to revert changes, it is necessary to consider multiple factors, including the order of operations executed in the `up` method.

Since Cloudware offers cloud-based products, database migrations are created and applied incrementally on a daily basis, with no downtime allowed.

All of Cloudware's databases are PostgreSQL[4] relational databases, all of them running version 11.4 or higher. As will be further explained, each company has its own company schema composed of many tables (around 300 on average). Adding onto this, a large amount of the business logic is implemented by the database, using tools provided by PostgreSQL such as triggers and user-defined functions written in PL/pgSQL[5].

1.2 Migration Problem

Figure 1.1 represents the process of database evolution at Cloudware currently. On the left, we have the local development environment, where programmers develop new features (in this context, by evolving the database). When these changes are finalized, it is necessary to deploy them and, to do so, a migration must be written. Migrations should be reversible to allow a database to be reverted to a previous version, if necessary. Migrations can affect the schema, the functions, or the data in a database. As can be seen in Figure 1.1, all migrations have to be executed in M databases, and migrations that perform changes on all schemas must also be executed in N schemas across those databases. Functions are not replicated once per schema and are instead shared by the entire database.

During development, it is also common to use preliminary versions of migrations to test multiple changes at once. This means the workflow may involve working on features locally, and when they are finalized, a migration is created; or instead, a migration is created from the start and is improved upon incrementally.

While in the local development environment, we only need to consider the schema and functions, we have to consider the users' data for a production environment. The schema is replicated in all companies, and it is then populated with the corresponding data. This means that occasionally it is required to execute migrations that make transformations to the data in a database without affecting its schema or functions.

[4] https://www.postgresql.org
[5] https://www.postgresql.org/docs/11/plpgsql.html

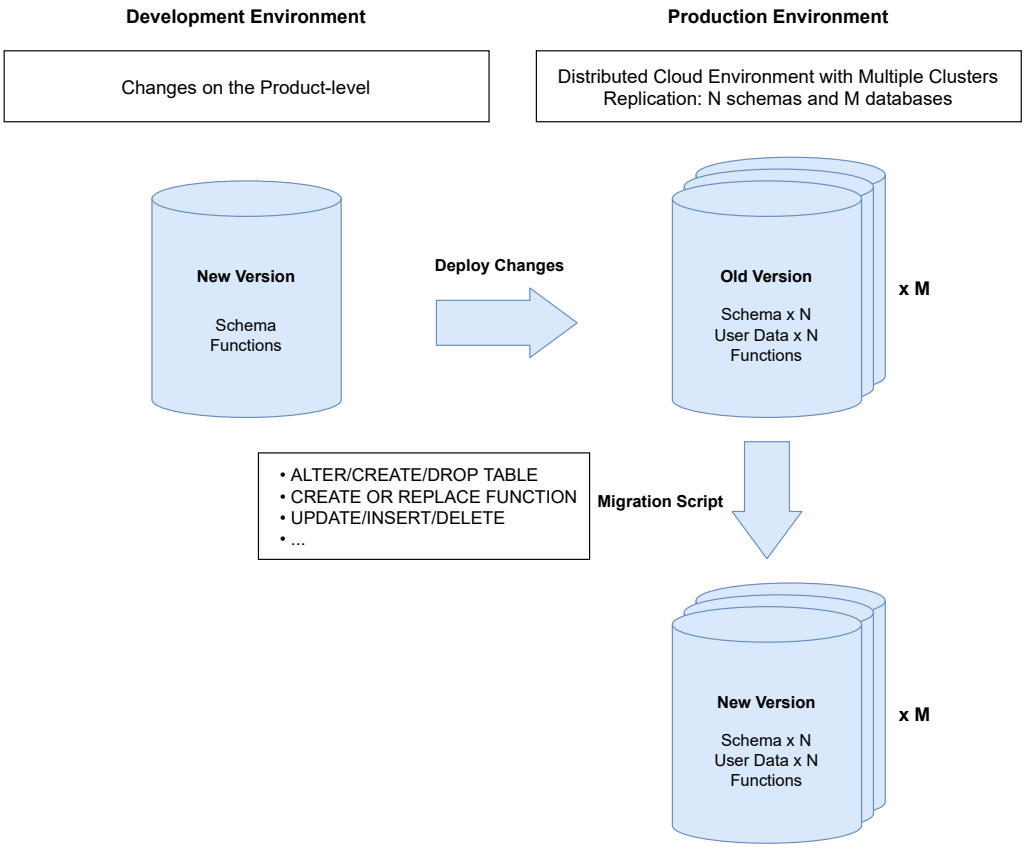

Figure 1.1: Overview of Cloudware's Current Method of Database Evolution

Supporting graceful schema evolution represents an unsolved problem for traditional information systems that is further exacerbated in web information systems [5, 4], as is the case with Cloudware. Deploys to the production environment are made daily, and every change can impact the database structure since many of these changes include database migrations. Cloudware's main product is web-based, and as such, there is no downtime allowed when deploying a change.

1.3 Objectives

The main goal of this dissertation work is to develop an approach and a tool capable of streamlining and automating the process of creating database migrations for developers, useful not only for Cloudware but also for other companies and developers facing similar problems. This tool should:

- Assist developers in the area of database evolution, by turning the process of creating database migrations more automated and reliable. This includes schema-changing migrations as well as function migrations.

- Preserve the ability for developers to manually edit the generated migrations, giving them the freedom to make any adjustments necessary. This is crucial for allowing non-standard operations, such as the migrations that perform data transformations previously mentioned.

This tool was validated in Cloudware's development environment to judge its viability to be added to the production process.

1.4 Document Structure

The remainder of this dissertation is structured as follows: Chapter 2 describes the current state of the art. Chapter 3 contains a description of the tool's design and implementation details. Chapter 4 describes all of the validation performed. Finally, Chapter 5 lists the conclusions for this dissertation.

Chapter 2

State of the Art

2.1 Database Evolution

In the current state of the art, the database administrator is essentially left alone in the process of evolving a database schema [5]. Given the available tools, this process is often not incremental, and there is usually no system support to check and guarantee data preservation. As such, schema and database evolution are critical, time-consuming, and error-prone activities [5] [14], and this has caused the area to be researched for several years now.

In the field of database evolution, in order to perform a change on a database, it is standard practice to write a migration script, which is essentially a list of SQL commands to be performed on said database. These migrations should be reversible to allow the database to be updated to a new version or rolled back to a previous version if necessary. These migrations may also possess a layer of scripting above the SQL commands, such as with ActiveRecord migrations, that can include logic on the Ruby level before executing the desired commands.

Migrations can be of several types, depending on what type of operations are executed. Schema migrations perform changes on the database's structure; function migrations create, update or erase functions from a database; and data migrations make changes to the data stored in a database. It is important to note that a migration can simultaneously change the schema, update functions and alter the data in a database.

In the database management system used by Cloudware, PostgreSQL, there is the possibility to add user-defined functions written in several languages such as PL/pgSQL. A vast majority of Cloudware's business logic is embedded in the database, making use of these functions. In PostgreSQL, a user-defined function is distinct from a stored procedure, in the sense that a function cannot start a new transaction inside itself, while a stored procedure can [12].

In Cloudware's specific context, and as was previously stated, although there are independent schemas for each company, each with its own tables, that is not the case for user-defined functions. Functions are not replicated in each company schema and are instead shared by the

entire database. These user-defined functions mainly deal with consulting and updating data in the database, although some perform schema-altering actions, such as activating modules and creating tables.

As was stated in [7], user-defined functions often have dynamic logic and queries, meaning that the validity of a function must be verified by actually calling it and seeing if it produces a runtime error [7]. In the same paper, a situation where several databases were distributed and had to be updated consistently was analyzed. This scenario is similar to the problem at hand since a change to the schema has to be replicated in multiple databases [7]. In this study, the problem was modeled not as a database one, but as a software evolution one [7].

This study also had other similarities to Cloudware's situation, such as the fact that the database architect decided to implement as much as possible the behavior of clients directly inside the database (as stored functions) [7], a practice also followed by Cloudware. Another conclusion from this study is that some dependencies are impossible to detect because of the difficulty to analyze dependencies inside the source code of stored functions [7] due to the potential for conditional logic and queries being built in runtime. Although this study has proved itself very useful in regards to having very similar restraints to the problem at hand, all of its conclusions are theoretical, with no practical developments conducted.

The topic of schema evolution in databases has also been studied in [2], where it goes over three different methods to evolving a schema. These approaches include allowing schema modifications, without retaining the pre-modification state, meaning that each schema change is applied irreversibly to the database [2]; the schema evolves independently of the data, then, after stabilization, the transformations are reflected on the data [2]; the state of the schema before modification is conserved, which means managing a set of schema versions [2].

This last approach contains two ways of organizing schemas, leading to two types of version: historical and parallel [2]. Moreover, with the first two approaches, the evolution of the database is not controlled, the validation of a new schema leads to the destruction of its predecessor, together with the corresponding data [2].

Although this paper is relatively old (1991), it is important to note that at that time, the problem of schema control was a recent research topic [2]. Furthermore, another interesting conclusion is that in practice, it is often not necessary to generate full versions of the schema, particularly when the modifications are minor and only concern a small part of the schema [2]. Although in Cloudware's case, getting a dump of the database or even just its schema is not a viable solution, since the databases' catalogs are massive since each company schema has a replica of all the necessary tables, and as such cannot be obtained in useful time.

2.2 Approaches Based on Scripting Languages with Embedded SQL

Currently, there are already several different tools and frameworks in the market that aim to solve or minimize the problems that arise with database evolution. Some of which, such as dbmate and ActiveRecord, are already currently used by Cloudware.

2.2.1 dbmate

dbmate[1] is a database migration tool. It is a standalone command-line tool that is framework-agnostic, meaning it can be used with any language or framework. Migrations for this tool are written in SQL files with two sections, the `up` and the `down`. As with other migration tools, the `up` section will contain all the code for implementing the desired change, and the `down` section will contain the code for reverting it. These sections are divided by comments in a format the tool understands.

However, since the migrations are essentially just SQL scripts that are directly executed, it lacks the ability to, for example, run a schema-altering migration on multiple database schemas. This feature is necessary to update all independent company schemas. As such, this tool is only used in Cloudware's smaller databases that do not have separate individual schemas (this includes the Central Database and the File Server, as will be shown later on).

```
1  -- migrate:up
2  ALTER TABLE public.users ADD COLUMN email text;
3
4  -- migrate:down
5  ALTER TABLE public.users DROP COLUMN IF EXISTS email;
```

Listing 2.1: Example of a dbmate migration for a schema change

To give a short example, Listing 2.1 shows how a migration that adds a column would be written in dbmate. As we can see, the `up` section contains the desired change and the `down` section contains the actions to revert said change, in this example, dropping the new column. As was previously explained, dbmate separates the `up` and `down` methods using specific comments that the tool is expecting.

```
1  -- migrate:up
2  DROP FUNCTION IF EXISTS public.get_user_info(integer);
3  CREATE OR REPLACE FUNCTION public.get_user_info (
4    IN p_id       integer
5  ) RETURNS SETOF record AS $BODY$
6  DECLARE
7    _user         record;
8  BEGIN
9    SELECT id, name, email        -- returning new field
10     FROM public.users
11    WHERE id = p_id
12     INTO _user;
13
14   RETURN NEXT _user;
15   RETURN;
```

[1] https://github.com/amacneil/dbmate

```
16 END;
17 $BODY$ LANGUAGE 'plpgsql';
18
19 -- migrate:down
20 DROP FUNCTION IF EXISTS public.get_user_info(integer);
21 CREATE OR REPLACE FUNCTION public.get_user_info (
22   IN p_id       integer
23 ) RETURNS SETOF record AS $BODY$
24 DECLARE
25   _user         record;
26 BEGIN
27   SELECT id, name
28     FROM public.users
29    WHERE id = p_id
30     INTO _user;
31
32   RETURN NEXT _user;
33   RETURN;
34 END;
35 $BODY$ LANGUAGE 'plpgsql';
```

Listing 2.2: Example of a dbmate migration for a user-defined function

To understand the differences between a migration that changes the schema and a migration that updates a function, Listing 2.2 represents an example of the latter, still using dbmate. In the `up` section, we can see the new version of the function, returning an extra parameter, and in the `down` section, we see the previous version of said function. The `DROP FUNCTION` commands are optional in this example since the input parameters for the function did not change. However, this is not always the case, meaning that when the signature does indeed change, a `DROP FUNCTION` of the previous version is mandatory.

2.2.2 ActiveRecord

ActiveRecord[2] is a part of Ruby on Rails[3]. Ruby on Rails is a framework for developing web applications that follow the Model-View-Controller (MVC) design pattern. ActiveRecord corresponds to the Model in this pattern, which is responsible for representing the business logic. Migrations are written in Ruby files that can be created using different templates. These migrations usually implement two methods, the `up` and `down`, but unlike dbmate these are actual Ruby methods and not simply sections in a SQL script. As such, complex logic can be written using Ruby that will then execute SQL commands, giving it more versatility. Cloudware expanded ActiveRecord and created custom methods to fit their needs. These methods are mainly used for running migrations on all independent company schemas.

[2] https://guides.rubyonrails.org/active_record_basics.html
[3] https://guides.rubyonrails.org

```
class AddEmailToUsersTable < ActiveRecord::MigrationWithoutTransaction
  def up
    migrate_companies do |schema_name, company_id, tablespace_name|
      execute <<-SQL
        ALTER TABLE #{schema_name}.users ADD COLUMN email text;
      SQL
    end
  end

  def down
    rollback_companies do |schema_name, company_id, tablespace_name|
      execute <<-SQL
        ALTER TABLE #{schema_name}.users DROP COLUMN IF EXISTS email;
      SQL
    end
  end
end
```

Listing 2.3: Example of an ActiveRecord migration for a schema change

We can see some of these custom methods in Listing 2.3, in particular `migrate_companies` and `rollback_companies`. In this example, we are assuming the *users* table exists once in every individual company schema, and as such, when we want to change it, we must do it in every company schema. For that, we use the custom methods `migrate_companies` and `rollback_companies`, that provide the information of every company in a given database, and we iterate over them to apply the desired changes. ActiveRecord provides a layer of scripting above the SQL commands that allow for this type of changes that iterate over several schemas, something that dbmate is not capable of since it runs SQL commands directly.

```
class AddEmailToGetUserInfoFunction < ActiveRecord::MigrationWithoutTransaction
  def up
    execute <<-SQL
      DROP FUNCTION IF EXISTS public.get_user_info(integer);
      CREATE OR REPLACE FUNCTION public.get_user_info (
        IN p_id       integer
      ) RETURNS SETOF record AS $BODY$
      DECLARE
        _user         record;
      BEGIN
        SELECT id, name, email        -- returning new field
          FROM public.users
         WHERE id = p_id
          INTO _user;

        RETURN NEXT _user;
        RETURN;
```

```
18        END;
19        $BODY$ LANGUAGE 'plpgsql';
20      SQL
21    end
22
23    def down
24      execute <<-SQL
25        DROP FUNCTION IF EXISTS public.get_user_info(integer);
26        CREATE OR REPLACE FUNCTION public.get_user_info (
27          IN p_id       integer
28        ) RETURNS SETOF record AS $BODY$
29        DECLARE
30          _user         record;
31        BEGIN
32          SELECT id, name
33            FROM public.users
34           WHERE id = p_id
35            INTO _user;
36
37          RETURN NEXT _user;
38          RETURN;
39        END;
40        $BODY$ LANGUAGE 'plpgsql';
41      SQL
42  end
```

Listing 2.4: Example of an ActiveRecord migration for a user-defined function

Finally, in Listing 2.4 we can see a migration to update a function using ActiveRecord. Similarly to Listing 2.2, the `up` method contains the new version of the function and the `down` method contains the old version.

These examples are simple and show that although this method works, it does not scale well, particularly when updating several functions. For example, if a developer wants to update ten functions, the size of the migration would grow ten times as well, since he would need to copy the new version of all functions and then go back to a previous version of the project to get the old versions. At Cloudware, to get the version of a function to place in the `down` method, developers usually go to the master branch of a project and copy it from the respective file.

2.3 Approaches Based on Schema Modification Operators

A common approach to streamline database evolution is the use of schema modification operators. These SMOs correspond to simple database operations, such as creating a table or dropping a column. In some cases, these SMOs are made atomic operations, such as with the PRISM workbench [5]; and in other cases, they are even reversible, such as with BiDEL [9]. These SMOs are later translated into actual database operations in the language at hand, such as SQL.

Frameworks and tools that use this technique usually create a language of SMOs that they support. Furthermore, by using combinations of these simple operations, they can perform more complex operations to allow for all of the intended actions. Moreover, by limiting all of the operations on the database to be written using these SMOs, it makes it easier to then automatically generate migration scripts, and in some cases even generate the script to revert a given migration, like with the PRISM workbench [5].

Using this approach to handle database evolution has one obvious drawback: it limits how a developer creates changes in a database. Simplifying and mapping operations into SMOs may allow for an easier understanding of the changes being employed and ease the automatic creation of migration scripts. However, it also restricts how a developer can interact with a database since developers are limited to using the available SMOs, not the full scope of features a language such as SQL provides.

2.3.1 CHiSEL

In the area of simplifying database evolution, we have CHiSEL (Compositional High-level Schema Evolution Language) [13, 14]. CHiSEL is a high-level, user-oriented, schema evolution framework that is built on a formal algebra of Schema Modification Operators (SMOs) [13]. The framework identifies a list of these schema modification operators, such as `SELECT` and `JOIN`, and then defines a set of composite SMOs that are defined by a combination of the simple SMOs [14]. Since these operations are defined relationally, CHiSEL is able to capitalize on established techniques like query evaluation in order to rewrite expressions into more efficient yet semantically equivalent ones [14]. This is done to avoid certain undesirable occurrences, such as redundant operations, for example [13]. The goal of rewriting the schema evolution expressions is to improve the time or space efficiency of the original expression [13].

However, CHiSEL is intended to be used by users that are not experts in database management and evolution [14], meaning that the operations possible with the framework are more limited than what an expert on the subject might require. For example, CHiSEL cannot partition a table into another following a given condition. Due to this reason, CHiSEL cannot be adopted by Cloudware as a solution to their problem.

2.3.2 PRISM Workbench

Another tool focused on aiding in database evolution is the PRISM workbench [5, 4]. This framework was built in an attempt to support graceful database schema evolution, a common problem in traditional information systems, and that is further exacerbated in the context of web applications [4], as seen in the problem at hand. PRISM was developed to tackle specific problems such as a lack of methods to predict and evaluate the effects of a given schema change, the need to rewrite queries and application code to operate on the new schema, and performing the migration of the database [5].

PRISM attempts to minimize these problems by providing a language of Schema Modification Operators to express schema changes of any complexity, as well as tools that allow the database administrator to evaluate the effects of such changes [5]. PRISM also provides an optimized translation of old queries to be compatible with a new schema version, as well as automatic data migration [5]. Similarly to CHiSEL [13], PRISM also is based on the concept of Schema Modification Operators (SMOs), and it has been tested over the real database schema evolution history of Wikipedia, more specifically, over 170 schema versions in a 4.5 year period [5].

PRISM is even more relevant to the problem at hand since it also generates an inverse data migration script based on the inverse sequence of SMOs used to create a schema change, meaning it supports a late rollback [5]. The framework also stores and supports the different schema version's history, with the ability to access any of these versions at any point. As such, PRISM stands as a very interesting tool due to solving a few of Cloudware's problems, mainly the automatic creation of migration scripts. However, it does not solve every problem since it is focused on schema and data altering modifications and cannot manage user-defined functions' versions, which is also an essential objective of the tool to be developed. The focus of PRISM on updating actual queries being performed in the application code is not as relevant to the problem at hand as well, since the aforementioned user-defined functions in Cloudware's context tend to be very dynamic, making it almost impossible to update them based on a given schema change automatically. Due to this reason, the tool to be developed should focus on handling changes to user-defined functions and not on automatically updating queries.

Despite all this, PRISM is still relevant as a starting point, by once again using Schema Modification Operators and using that approach to generate a migration script for implementing a given change, as well as a script for reverting it if the need for that arises.

PRISM was further expanded upon with the release of PRISM++, which was the first system that offered end-to-end support for query and update adaptation through both structural schema changes and integrity constraints evolution [6]. With PRISM++, given a specification of the desired schema and integrity constraints evolution, it automates the migration of the data and the mapping of legacy queries and updates from the old schema to the new schema [6]. Moreover, a further enhancement of PRISM++ was implemented in the form of CoDEL (Complete Database Evolution Language) [8]. CoDEL builds upon the set of PRISM++ SMOs to inherit its practical feasibility. However, CoDEL is relationally complete and equally expressive as any minimal language providing relational completeness [8].

2.3.3 BiDEL Language

Finally, some languages already deal with changing between different schema versions natively. An example of such a language is BiDEL [9, 10], which stands for Bidirectional Database Evolution Language. BiDEL provides simple but powerful Schema Modification Operators that define the evolution of both the schema and the data from one version to the next [9]. However, as opposed to previous tools, the list of SMOs provided by BiDEL is bidirectional, meaning that any

operation can be reverted easily [10]. In the paper where it is presented [9], BiDEL is used in conjunction with InVerDa (Integrated Versioning of Databases), which is an extension to relational DBMSes to enable true support for coexisting schema versions in the same database [9]. InVerDa provides developers a list of Database Evolution Operations that are then executed using BiDEL to evolve from the current schema to a new version. BiDEL is powerful in the sense that it allows any changes to be propagated between all coexisting schema versions, including regular read and write operations [9, 10].

Similarly to Ruby on Rails' migrations using ActiveRecord, InVerDa helps managing schema versions outside the DBMS and generates SQL scripts for migrating between different versions. However, ActiveRecord focuses only on schema evolution, while InVerDa aims to support coexisting schema versions in the same database [9]. This feature is not one that Cloudware requires, and it also has the significant drawback of increasing the size of the catalog in a database dramatically since it creates an instance of the schema for each change performed. For this reason, both InVerDa and BiDEL fall out of the scope of the problem at hand. Another reason for this is that to adopt these technologies, it would require a complete change in the database language used by Cloudware, and such a task would require tremendous work due to the complexity of its databases, and as such, it is not an option.

2.3.4 Application Example

Looking now at a practical example, Figure 2.1 shows two versions of a schema, and it will be migrated using the PRISM workbench [5]. This particular framework was chosen since CHiSEL is intended to be used by non-developers [13], and BiDEL focuses on maintaining multiple schema versions and allowing them to coexist in the same database [9]. Both of these features fall out of the scope of the problem at hand and could influence the analysis.

```
-- SCHEMA v41 --
old(oid, title, user, minor_edit, text, timestamp)
cur(cid, title, user, minor_edit, text, timestamp,
    is_new, is_redirect)
-- SCHEMA v42 --
page(pid, title, is_new, is_redirect, latest)
revision(rid, pageid, user,  minor_edit, timestamp)
text(tid, text)
```

Figure 2.1: Example of 2 Versions of a Schema

This example was extracted from [5], and it shows a simplified version of the code required to evolve from version 41 to 42 of a given schema. This sequence of changes is presented in two formats in Figure 2.2: on the left, by using the well-known relational algebra notation on an intuitive graph [5], and on the right employing the SMO language supported by PRISM [5]. Trivial steps such as column renaming have been omitted to simplify the Figure 2.2 [5].

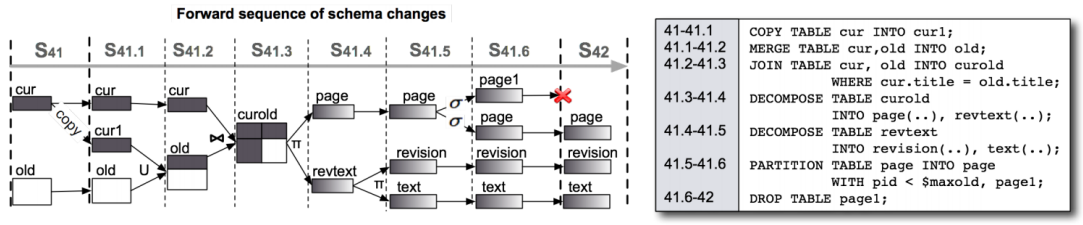

Figure 2.2: Schema Evolution using PRISM between 2 Schema Versions

This example represents a fairly complex operation. In schema version 41, current and previous revisions of an article have been stored in the separate tables `cur` and `old` respectively [5]. From schema version 42 on, the layout has been significantly changed: table `page` stores article metadata, table `revision` stores metadata of each article revision, and table `text` stores the actual textual content of each revision [5]. These representations are equivalent in terms of what information they maintain. Furthermore, due to being able to handle a migration of this complexity, it is safe to say that an SMO approach such as the one used by the PRISM framework provides enough flexibility for developers to perform all the necessary tasks for database evolution.

This is further confirmed by looking at Figure 2.3, as we can see the full extent of operations that PRISM is capable of. The list of usable SMOs appears to include all the operations a database administrator would require to evolve a database schema.

SMO Syntax	Input rel.	Output rel.	Forward DEDs	Backward DEDs
CREATE TABLE R($\bar{A}$)	-	R($\bar{A}$)	-	-
DROP TABLE R	R($\bar{A}$)	-	-	-
RENAME TABLE R INTO T	R($\bar{A}$)	T($\bar{A}$)	R($\bar{x}$) → T($\bar{x}$)	T($\bar{x}$) → R($\bar{x}$)
COPY TABLE R INTO T	R_{V_i}($\bar{A}$)	$R_{V_{i+1}}$($\bar{A}$), T($\bar{A}$)	R_{V_i}($\bar{x}$) → $R_{V_{i+1}}$($\bar{x}$) R_{V_i}($\bar{x}$) → T($\bar{x}$)	$R_{V_{i+1}}$($\bar{x}$) → R_{V_i}($\bar{x}$) T($\bar{x}$) → R_{V_i}($\bar{x}$)
MERGE TABLE R, S INTO T	R($\bar{A}$), S($\bar{A}$)	T($\bar{A}$)	R($\bar{x}$) → T($\bar{x}$); S($\bar{x}$) → T($\bar{x}$)	T($\bar{x}$) → R($\bar{x}$) ∨ S($\bar{x}$)
PARTITION TABLE R INTO S WITH *cond*, T	R($\bar{A}$)	S($\bar{A}$), T($\bar{A}$)	R($\bar{x}$), *cond* → S($\bar{x}$) R($\bar{x}$), ¬*cond* → T($\bar{x}$)	S($\bar{x}$) → R($\bar{x}$),*cond* T($\bar{x}$) → R($\bar{x}$),¬*cond*
DECOMPOSE TABLE R INTO S($\bar{A}$,$\bar{B}$), T($\bar{A}$,$\bar{C}$)	R($\bar{A}$,$\bar{B}$,$\bar{C}$)	S($\bar{A}$,$\bar{B}$), T($\bar{A}$,$\bar{C}$)	R($\bar{x}$,$\bar{y}$,$\bar{z}$) → S($\bar{x}$,$\bar{y}$) R($\bar{x}$,$\bar{y}$,$\bar{z}$) → T($\bar{x}$,$\bar{z}$)	S($\bar{x}$,$\bar{y}$) → ∃$\bar{z}$ R($\bar{x}$,$\bar{y}$,$\bar{z}$) T($\bar{x}$,$\bar{z}$) → ∃$\bar{y}$ R($\bar{x}$,$\bar{y}$,$\bar{z}$)
JOIN TABLE R, S INTO T WHERE *cond*	R($\bar{A}$,$\bar{B}$), S($\bar{A}$,$\bar{C}$)	T($\bar{A}$,$\bar{B}$,$\bar{C}$)	R($\bar{x}$,$\bar{y}$), S($\bar{x}$,$\bar{z}$), *cond* → T($\bar{x}$,$\bar{y}$,$\bar{z}$)	T($\bar{x}$,$\bar{y}$,$\bar{z}$) → R($\bar{x}$,$\bar{y}$),S($\bar{x}$,$\bar{z}$),*cond*
ADD COLUMN C [AS *const*|*func*($\bar{A}$)] INTO R	R($\bar{A}$)	R($\bar{A}$,C)	R($\bar{x}$) → R($\bar{x}$, *const*|*func*($\bar{x}$))	R($\bar{x}$,C) → R($\bar{x}$)
DROP COLUMN C FROM R	R($\bar{A}$,C)	R($\bar{A}$)	R($\bar{x}$,z) → R($\bar{x}$)	R($\bar{x}$) → ∃z R($\bar{x}$,z)
RENAME COLUMN B IN R TO C	R_{V_i}($\bar{A}$,B)	$R_{V_{i+1}}$($\bar{A}$,C)	R_{V_i}($\bar{x}$,y) → $R_{V_{i+1}}$($\bar{x}$,y)	$R_{V_{i+1}}$($\bar{x}$,y) → R_{V_i}($\bar{x}$,y)
NOP	-	-	-	-

Figure 2.3: Schema Modification Operators (SMOs) for PRISM

However, an SMO approach does imply that the developers would have to learn a new language (the respective SMO language, such as the one shown in Figure 2.2), just to be able to create database migrations. This constitutes a significant investment required to adopt a tool using this type of approach, and such a cost might not be viable to the solution being implemented. Also, an SMO approach does not handle function migrations, as has been previously stated.

Table 2.1: Summary of main functionalities of each tool analyzed

	Execute Migrations	Generate Migrations	Input Language	Extensibility / Schema Variability	Other Features
dbmate	Schema, Data, Functions	No	SQL Only	No	-
ActiveRecord	Schema, Data, Functions	No	Arbitrary Ruby Code	Yes (Manually)	-
CHiSEL	Schema, Data	Up (Based on SMOs)	DSL	No	Targeted for non-developers
PRISM	Schema, Data	Up & Down (Based on SMOs)	DSL	No	Update application queries
BiDEL	Schema, Data	Up & Down (Based on SMOs)	DSL	No	Co-existing schema versions

2.4 Approaches Comparison

Table 2.1 lists the tools currently used by Cloudware (dbmate and ActiveRecord) and the tools found during the research of the state of the art.

As we can see, both dbmate and ActiveRecord can execute migrations that change the schema, data, or user-defined functions; however, they cannot automatically generate migrations. Of the tools found, all of them can only execute migrations of schema or data. However, they can automatically generate migrations based on the definition of higher-level SMOs. The PRISM workbench [5] and BiDEL [9] can generate both the `up` and the `down` methods, while CHiSEL [13] only generates the `up` method automatically.

As for the input language, dbmate receives standard SQL code only; ActiveRecord receives arbitrary Ruby code that will, in turn, execute SQL commands, and the other three tools receive code in their respective domain-specific languages (DSLs).

Of all the tools analyzed, only ActiveRecord can be easily extended to handle new scenarios and is also the only one capable of handling migrations that need to be executed in multiple schemas. However, currently, this has to be done by manually writing code in the migration.

Each of the researched tools has some extra features that may prove relevant for this work. CHiSEL was developed for users that are not developers and lack knowledge in database evolution [13]. The PRISM workbench focuses on also updating queries in the application code when schema changes occur [5]. However, it cannot do this with user-defined functions due to their complexity and conditional logic. Finally, BiDEL allows for multiple versions of the schema to coexist in the same database [9].

As can be seen, and to the best of our knowledge, none of these tools completely solve the problem at hand, hence the need for a more complete solution.

Chapter 3

Tool Design and Implementation

3.1 Architecture

NANDO was developed as a Ruby Gem, with an entry point class `NandoMigrator`. This class is a singleton since only one instance should exist at any given time [11], as well as to facilitate access to its class variables from the other components. Figure 3.1 represents NANDO's architecture using a UML Component Diagram [3].

The components `Migration` and `MigrationGenerator` are covered in Section 3.2, and are responsible for the creation and execution of migrations. Section 3.3 will go over the `MigrationUpdater` component, responsible for automating the creation of migrations that add or update functions using the *update* command. Section 3.4 will go over the `SchemaDiff` component, responsible for comparing two database schemas using the *diff* command. Section 3.5 will go over Cloudware's custom methods, that allow migrations to be executed in multiple schemas. And finally, all the helper methods the tool provides are described in Section 3.6, and are represented in Figure 3.1 by the component `Helper Methods`.

3.2 Basic Migrator Features

The initial feature the tool had to implement in order to be considered a database migrator is, of course, the ability to create, apply and revert migrations on a database. NANDO migrations were implemented in a format identical to ActiveRecord since this was one of the migration tools already used by Cloudware. This decision minimizes the required learning curve for the tool and assists in preserving all of the functionality previously available.

As can be seen in Figure 3.1, the `MigrationGenerator` component is responsible for handling the creation of new migrations created using the *new* command. These migrations have internal methods, such as `execute` (used for executing SQL code on the database), that are implemented in the classes of the component `Migration`.

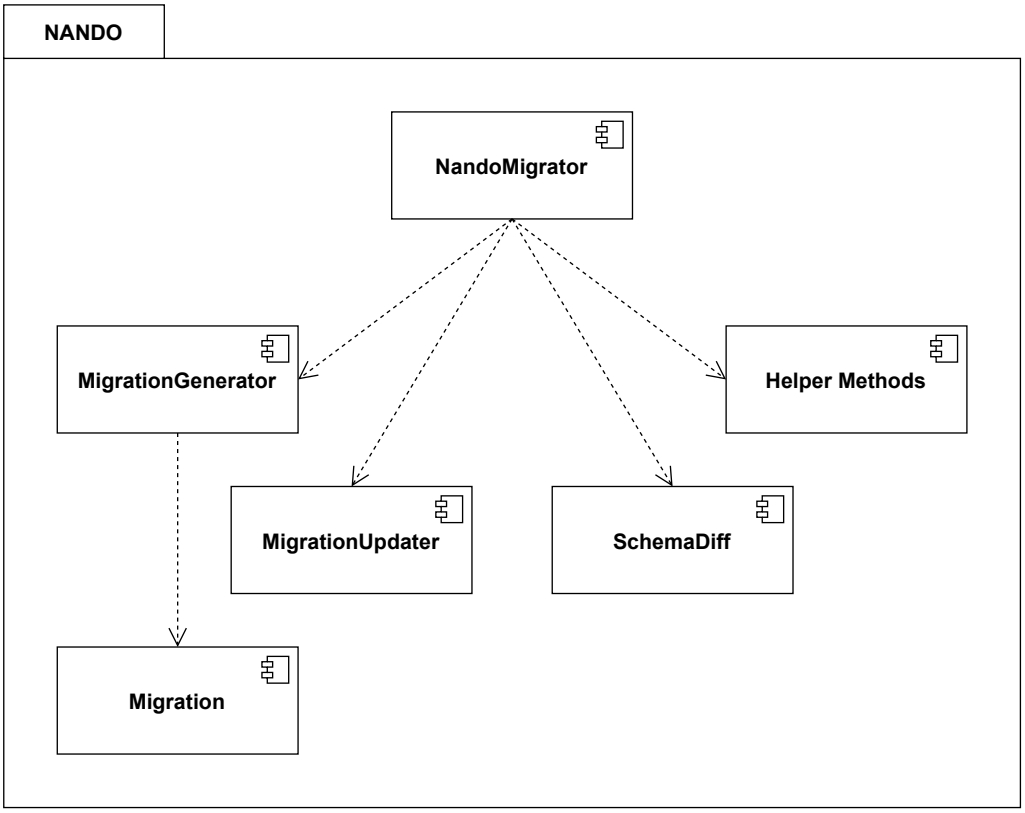

Figure 3.1: Overview of NANDO's Modular Architecture

Besides being the entry point of the program, `NandoMigrator` also handles the process of applying or reverting migrations when using the *up*, *down* and *apply* commands.

3.3 Database Functions Migration

After implementing the essential features, a migration tool requires, one of the following goals was to automate the process of creating migrations that add or update database functions. Creating a migration requires filling its `up` and `down` methods. The `up` method is called when applying a migration and should contain the code to migrate the database to a new version. In contrast, the `down` method is called when reverting a migration and should contain the code required to migrate the database to the previous version.

Traditionally, when a developer wants to update or add a function to the database, he would have to change the respective SQL file containing the function definition (or create one if the function is new). A standard practice followed at Cloudware is to have a separate SQL file for every database function. This file should also be versioned using a version control software such as Git[1].

[1] https://git-scm.com/doc

After this, a migration would have to be created, and the contents of the SQL file would have to be copied manually into it to fill the `up` method. Finally, to fill the `down` method, the developer would have to obtain the previous definition of the function, either by using Git or by searching the previous migrations for the latest redefinition of the function at hand. Creating these function migrations is currently a manual process and, as such, is very prone to errors, hence the need to automate it.

In order to do that, the tool will receive an indication of the files containing the functions we want to add or update, whose contents will fill the `up` method. Moreover, to determine the version of the function that will fill the `down` method, the tool will use the history of previous migrations to determine the last version of each function that was applied to the database.

In case a new function is being added to the database, this implies no previous definition of it exists. As such, in this scenario, the tool would still create the respective annotation in the `down` method; however, no other action is taken since there is no previous definition the function could revert to. Since only the annotation is created, the developer could potentially add a `DROP FUNCTION` command for new functions being added, if necessary.

This approach allows for several functions to be updated in the same migration, with the same amount of effort as updating a single function. In contrast, the manual approach requires significantly more effort and time when the number of functions the developer wants to update increases. The only cost in this new approach is listing the functions that need updating, while in the manual counterpart, there is the cost of opening and copying the contents of each function SQL file to fill the `up` method and the cost of determining which is the previous version of the function to fill the `down` method. The time to perform these tasks manually multiplies with the number of functions to update.

To perform this action with NANDO, the *update* command is used, where the developer only needs to specify the relative path to the migration to be expanded, and the tool will automatically fill the respective `up` and `down` methods, as will be explained in Sections 3.3.1 and 3.3.2.

Instead of using the `execute` function when executing the code that redefines a database function, another method was implemented to ensure that multiple functions are not being redefined in the same block. This method was named `update_function`, and it ensures only one database function is being redefined within the SQL code it executes. This is required to ensure the *update* command does not import a file that has multiple function definitions. The usage of this method also increases the performance for the `down` method, since the tool only has to look at the code inside each `update_function` block, and not all `execute` blocks, since these could contain any SQL code, not just database function definitions.

3.3.1 Up Method

In order to automatically fill the `up` method of a migration, the program requires some indication of the file from which it must obtain the SQL source code of each function. As such, the approach we considered to be ideal would be to create a custom annotation (in the format `"# NANDO: <path to file>"`) that would indicate to the program that during an *update* command, below

each of these annotations, the program should insert the contents of the respective file. To add annotations to a migration file, the developer can use the `-f/--function` flag or insert them manually. Then, after all the annotations have been added to the migration, running an *update* command will iterate over all the annotations, find the respective files and insert their contents on the migration inside an `update_function` directive, as was previously described. With these annotations, the *update* command can be executed as many times as needed since it also replaces any previous definition of the function before inserting the source code from the specified file.

Since this approach only depends on the path to the file containing the function definition, the only potential edge case is a change on the file path. If this change happens after the migration has been created, a warning is issued to the developer during the following *update* action, informing them that the specified file was not found. On the other hand, if the file path changes, and then we want to add that function to a migration, the process is the same as if the file has not changed. As long as the annotations include the correct file paths, the tool will always work correctly without requiring extra steps.

3.3.2 Down Method

As previously mentioned, to fill the `down` method of a migration automatically, we will use the previous migrations to determine the previous function definition that was applied. To do that, the program starts by creating a similar annotation as in the `up` method: `"# NANDO_DOWN: <path to file>"`. When running an *update* command, with each annotation in the `up` method, the program will search for an annotation in the `down` method with the same file path or create one if one does not exist. As such, the program will only use the file path in the annotations from the `down` method to match them to a respective annotation in the `up` method.

After finding or creating the necessary annotations on the `down` method, the program will then start to look at migrations with an inferior timestamp for the most recent redefinition of that function. This process begins by identifying the function present on the current source file, which requires that every SQL file contains a single function definition, a standard already followed by Cloudware. After identifying the current function, the program iterates over the previous NANDO migrations (starting from the timestamp of the current migration and moving backward) to find the most recent redefinition of the current function in an `up` method. After finding a previous definition, its contents are copied to the current migration, under the respective annotation in the `down` method. If no previous definition was found, the program issues a warning to the developer, and no code is inserted into the `down` method. This scenario occurs when a new function is added to the database or when a previous function has its name changed. In these scenarios, the developer has the freedom to add the SQL code he believes is necessary in each case, such as calling a `DROP FUNCTION` on the new function. Alternatively, he can also perform no action at all in these cases.

The heuristic used in identifying a function only uses its name and ignores the rest of the signature (namely, its arguments and return type). This approach was chosen since it is common to change a function's arguments or its return type. Ignoring them allows the program to find a previous definition of the function in a higher percentage of scenarios. The drawback of this approach is

that it prevents the usage of polymorphic functions since, when searching for a previous definition, the program has no way of determining if it is analyzing the correct function or a different function with the same name but different parameters. However, this cost is inconsequential since Cloudware does not use polymorphic database functions, since they often lead to unexpected behavior and more complicated logic to analyze. As such, the tool can afford to follow this approach.

When starting to use NANDO, the *baseline* command should be used to ensure the tool can find matches for the `down` method annotations. This command creates a single migration with all the functions currently in the database in separate `update_function` directives. A baseline migration can be created and applied at any time since they do not change the database state. Thus, all of the function definitions stay the same before and after applying these migrations. The purpose of these migrations is to ensure that all existing functions have at least one previous definition to be matched by the *update* command.

3.4 Database Schema Diff

After automating the creation of database function migrations, the next goal was to allow the tool to detect any changes to a database schema and provide the developer with suggestions to apply said changes to all database schemas.

As was previously mentioned, the traditional development method for schema changes at Cloudware involves the developer making the changes he needs in a specific database schema and then testing those changes in that specific company. When the feature is tested and finalized, the developer must create a migration to apply the schema changes to all necessary schemas in a database. In this particular use case, the developer would need to recreate the changes he applied during the development phase. Therefore, automating this process would be tremendously valuable since it would minimize human error.

The chosen approach for this problem involves the developer providing two schemas: the source schema, where he performed the changes, and a target schema used for comparison. The program will start by querying the database and obtaining all the relevant information for a comparison of schemas. The program uses the PostgreSQL system catalogs, a set of internal tables where a relational DBMS stores schema metadata, such as information about tables and columns[2].

These two schemas can be compared using the *diff* command. This command will compare both schemas and present the developer with suggestions on how to transform the source schema into the target schema and vice-versa. This comparison is made in both directions in order to generate commands for both the `up` and `down` methods since the changes required to transform the source schema into the target schema can be reverted using the changes required to transform the target schema into the source schema.

The program stores all relevant information about a schema's tables, views, columns, triggers, constraints, and indexes. The program starts by creating a hash map for the schema structure with two entries: one hash map for tables and one hash map for views. These are the highest level of

[2]https://www.postgresql.org/docs/11/catalogs.html

schema database objects considered, and all others mentioned will exist associated with either a table or a view. The program then creates an entry in these two hash maps for every view or table found in the system catalog. These new entries will have hash maps for the subsequent database objects, namely columns, triggers, constraints, and indexes. After this structure is created, the program will then query the system catalogs to fill in all information regarding these remaining four types of database objects.

When the program has completed all queries to both schemas, it will then compare them and register any differences. As such, the program classifies 3 possible scenarios for differences: *extra*, *missing* or *mismatching*. Using columns as an example, the program is prepared to handle:

- A column that is found on the source schema but not on the target schema (*extra*)
- A column that is not found on the source schema but exists on the target schema (*missing*)
- A column that is found on both schemas, but its definition does not match between them (*mismatching*)

The heuristic used to compare two database objects is as follows: every object has some field that identifies it uniquely, such as a column's name or an index identifier. This field will be used as the *key* in the respective hash map. If the source schema has a database object with a name that does not appear on the target schema, then it is classified as *extra*. If the reverse is true, then it is classified as *missing*. Finally, if both schemas have a database object with the same name, all of its remaining properties are compared, and if they do not match, then the object is classified as *mismatching*. The logic of obtaining and storing all the info about a schema was separated from the logic of comparing said schemas to decouple these two aspects of the process.

Although the program can always suggest commands to fix any *extra* or *missing* database objects (such as columns or triggers), it is not always possible to suggest to the developer a command to correct a *mismatching* scenario. For example, in the case of a trigger, index, or constraint, redefining them would involve destroying the existing database object and recreating it, a process that could have unexpected implications. In these scenarios, the program only warns the developer that there is a difference in the definition of the database object, leaving the responsibility of determining the best solution to the developer. In other cases of mismatching database objects, such as columns, the program suggests the commands to apply the necessary changes. However, a warning is still issued to the developer. This warning is necessary since, even though these modifications can be executed with the provided commands, they can also fail. An example of this would be attempting to change a column's data type from `text` to `integer`. This operation would raise an error since PostgreSQL cannot always automatically cast a `text` column into an `integer`.

A summary of the database objects and respective attributes the tool compares can be found in Table 3.1. As was previously mentioned, the tool can always suggest commands when a database object is listed as *missing* or *extra*, and as such, this information was omitted from Table 3.1.

The first column indicates the type of database object, while the second column lists the attributes that are compared by the tool. The third column lists the *mismatching* attributes the tool

can generate commands in order to correct, while the fourth column lists the *mismatching* attributes that, when detected, cause the tool to raise a warning. As we can see, the tool can only generate commands for differences in 3 of the attributes of columns, and a warning is still raised to the developer in these scenarios. For every other type of attribute that is listed as *mismatching*, only a warning is raised due to the reasons previously mentioned. The tables and views are not compared directly. Instead, their respective columns, triggers, constraints, and indexes are compared with each other.

Table 3.1: Attributes compared by NANDO and respective generated suggestion

Database Object	**Atributes Compared**	**Generates Command when Mismatching**	**Raises Warning when Mismatching**
Table	Respective columns, triggers, constraints and indexes	-	-
View	Respective columns, triggers, constraints and indexes	-	-
Column	Column's relative position in the table/view; Column's default value; Column is NOT NULL; Column's datatype;	Column's default value; Column is NOT NULL; Column's datatype;	Column's relative position in the table/view; Column's default value; Column is NOT NULL; Column's datatype;
Trigger	Trigger's definition;	-	Trigger's definition;
Constraint	Constraint's source code; Constraint's definition;	-	Constraint's source code; Constraint's definition;
Index	Index definition; Index affected columns;	-	Index definition; Index affected columns;

Some database objects store their own schema's name in their definition or in other attributes. To avoid classifying these instances as *mismatching*, the program replaces any references to its schema with a constant `__SCHEMA_NAME__` when storing the information about each schema. This replacement is done with this constant instead of just an empty string to allow the program to replace it again with the appropriate value when suggesting the schema-altering commands.

After both schemas have been compared and the differences have been stored, the program will show the differences found. This output is separated for each table that has differences in order to improve readability. The differences are also color-coded, allowing developers to understand the output easily, and a respective symbol is also shown at the beginning of each line.

Database objects marked as *extra* are shown with the color green and with a "+" symbol, those marked as *missing* are shown with the color red and with a "-" symbol, and finally, those marked as *mismatching* are shown with the color yellow and with a "?" symbol. So, for example, in case a table has an extra column in the source schema when compared to the target, then the output would show the table as *mismatching*, and then inside its context, it would show the *extra* column. As was previously mentioned, this comparison is performed in both directions, and as such, the output shows both comparisons.

An example of the output of a *diff* command can be seen in Figure 3.2. In this scenario, when comparing `schema_1` to `schema_2` the table `users` does not match, because the column `name` is *missing* from the source schema (in this scenario, `schema_1`). When performing the

comparison with the schemas switched, the same table still does not match, however now the column `name` is listed as *extra* in the source schema when compared to the target (`schema_2` and `schema_1`, respectively).

```
❯ nando diff schema_1 schema_2

Comparing 'schema_1' to 'schema_2'
? Table 'users'
-   Column 'name'

Comparing 'schema_2' to 'schema_1'
? Table 'users'
+   Column 'name'
```

Figure 3.2: Example Output of a *diff* Command

After this, the developer is asked if he wants to see the suggested commands to fix these differences. If so, the program will then generate these commands, replacing the `__SCHEMA_NAME__` placeholder with `#{schema_name}`, or the value indicated in the *.env* file for the `SCHEMA_VARIABLE` variable. The program uses this variable and not the target schema or source schema in the suggestions since this command intends to generate the code necessary to apply a change to all database schemas. In order to facilitate this, methods that iterate over groups of schemas and that use this syntax have been implemented, as will be further explained in Section 3.5.

For every table, the first types of commands printed are any `DROP` commands that do not require to be inside an `ALTER TABLE` command, such as `DROP TRIGGER` or `DROP INDEX`. After that, if any commands require an `ALTER TABLE`[3], then one is created, and all of the operations that need to be performed inside it are then printed and correctly indented for better readability. Finally, if there are any `CREATE` commands that do not require being inside an `ALTER TABLE`, those are printed last, such as creating an index or a trigger. This order of operations is suggested in order to minimize errors due to dependencies.

Since the schema comparison is executed in both directions, this allows the generated suggestions to fill both the `up` and `down` method, since the changes required to turn the source schema into the target schema are the reverse of the changes required to turn the target schema back into the source schema.

Finally, this *diff* command can also be used for other purposes more specific to Cloudware. For example, the process of creating new companies (and, as such, new schemas) can occasionally have inconsistencies due to human error, which leads to differences in the structure of these new schemas. A simple check that can now be performed with NANDO is to create a new company in the local development environment and then compare it with a pre-existing one. If no changes are found, both schemas should be equivalent, and the creation of new companies is working as intended.

[3] https://www.postgresql.org/docs/11/sql-altertable.html

3.5 Multiple Schema Iterators

As has been previously referenced, Cloudware divides all the information for each company in a separate database schema. This approach was chosen for several reasons, such as reducing the data size of tables (and thus increasing performance) and allowing many companies to share the same database, ensuring scalability. However, this approach also comes with certain drawbacks, such as forcing migrations that change the database schema to be executed on all independent company schemas instead of just in one "central" schema. Because of this, it is also essential to have an understanding of Cloudware's structure for each schema to determine what is the best way to approach the problem.

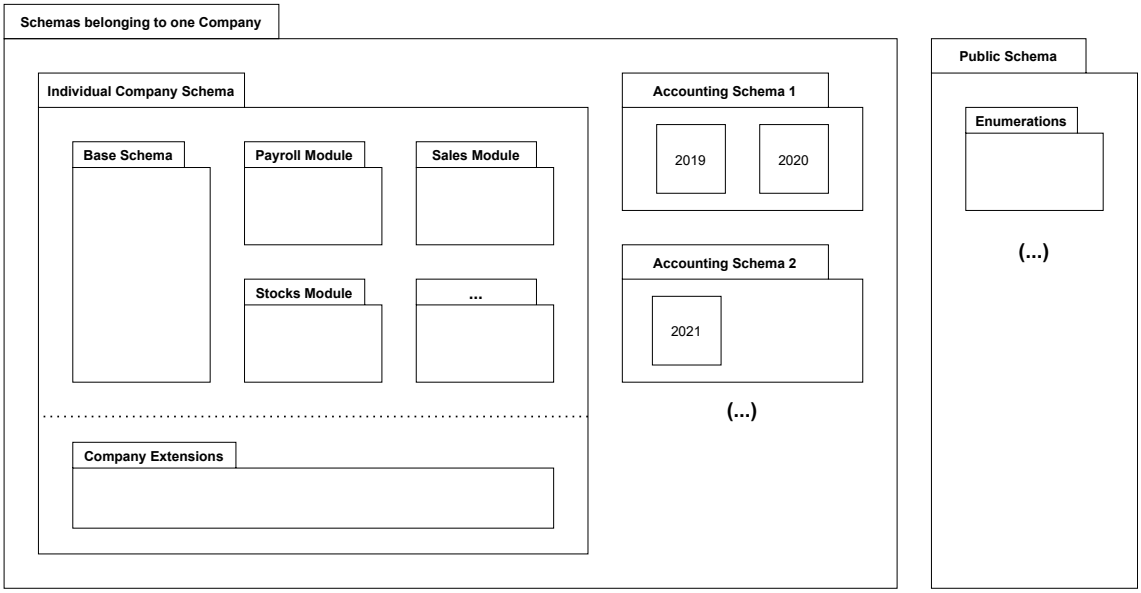

Figure 3.3: Structural View of Database Schemas

As can be seen in Figure 3.3, each company possesses one individual company schema, that contains most of the tables that belong to it, some of which are then separated into modules (this will be explained later). However, besides the individual schema, a company can also possess multiple accounting schemas. These schemas are a bit different from the individual ones since a company can have multiple of them (or none at all), and they are further subdivided into fiscal years, as can be seen in Figure 3.3. The details of these schemas are out of the scope of this problem. However, it is crucial to keep in mind that a company has one individual schema and 0 or more accounting schemas, since certain migrations will have to run only on all individual schemas, and others will need to run only on all accounting schemas, or in other groups of schemas.

The structure previously described is repeated for each company; however, specific tables are common to all companies (such as certain enumerations or the table that contains a list of all companies, for example). This type of information is stored in the *public* schema, meaning certain migrations will not iterate over all schemas but will instead just be executed in one schema, like a traditional migration.

Finally, the user-defined functions are not replicated in every schema; they are shared by the entire database. This means that any migration that creates or updates a function only has to run once in each database, never once for each company schema or accounting schema.

As was previously mentioned, the individual company schema contains some tables that all companies have (in Figure 3.3, it is referred to as "Base Schema"), and then the rest of the tables are subdivided into modules (some of which are also represented in Figure 3.3). These modules represent tables that are only created when a user activates them for a specific company. This means that two companies may have a different amount of tables in their individual schemas, depending on which modules they have active. This approach is helpful to reduce the number of tables in each schema by only creating them when they start being necessary.

As such, one requirement of this tool was that it needed to preserve all functionality that was previously possible using either dbmate or ActiveRecord, which includes the helper methods that Cloudware had previously added to ActiveRecord, that allowed migrations to be executed in certain types of schemas dynamically. These methods were initially implemented in Ruby and were called directly in the migration code.

Since NANDO is also implemented using Ruby, porting these methods was relatively simple. The only code that had to be adapted was the way the tool interacts with the database since the previous version used ActiveRecord specific methods. In contrast, NANDO has to make use of the PG library[4].

These helpers are specific to Cloudware's context and include methods that iterate over different groups of schemas, such as all company schemas or all company accounting schemas. These iterators provide different variables such as the current schema name or the current schema's tablespace value, which are necessary to dynamically run a migration on multiple schemas. These variables are interpolated within the SQL code that will be executed, as can be seen in Listing 3.1. Since these methods iterate over multiple schemas in a database to perform changes, they can only be called in migrations without transaction to prevent creating giant transactions that would significantly slow down the database.

```
1  class AddEmailToUsersTable < NANDO::MigrationWithoutTransaction
2    def up
3      migrate_companies do |schema_name, company_id, tablespace_name|
4        execute <<-SQL
5          ALTER TABLE #{schema_name}.users ADD COLUMN email text;
6        SQL
7      end
8    end
9
10   def down
11     rollback_companies do |schema_name, company_id, tablespace_name|
12       execute <<-SQL
13         ALTER TABLE #{schema_name}.users DROP COLUMN IF EXISTS email;
```

[4] https://www.rubydoc.info/gems/pg/PG

```
14        SQL
15      end
16    end
17  end
```

Listing 3.1: Example of a NANDO migration using Cloudware's custom methods

As we can see in Listing 3.1, Cloudware's custom methods act like iterators over a specific list of schemas while providing information of each one. In this example, the `migrate_companies` method provides information about each schema's name, the `id` of the company it belongs to, and the tablespace name where the schema is stored. Tablespaces in PostgreSQL allow the database administrators to define locations in the file system where the files representing database objects will be stored[5]. In this example, we can also see how these variables are interpolated into the SQL code, as is the case with the `schema_name` variable.

As a final note, even though it is not the focus of the work being developed, it is essential to keep in mind that Cloudware has different projects, and some of them are distributed over several databases. The fact that some migrations have to be executed in separate databases could add another restriction to the tool.

As shown in Figure 3.4, Cloudware currently has 12 clusters, each containing one database and several web servers that interact with it. These cluster databases contain most of Cloudware's data, including all of the independent company schemas previously mentioned. In addition, some databases interact with all clusters, such as the Central Database and File Server. Since these last databases do not have multiple clusters, migrations that will be applied to them never use the custom Cloudware methods.

All of Cloudware's databases were previously migrated using either dbmate or ActiveRecord. However, these tools were not aware of how many different databases Cloudware has. This responsibility belonged to Cloudware's deployment scripts, which would run the required commands on the necessary machines, depending on the action the developer specified upon deploying. This level of separation was kept for NANDO, meaning that the tool also lacks awareness of the multiple existing databases, relying on the deployment scripts to handle that responsibility.

When deploying to either the Central Database or the File Server, the deployment scripts send the appropriate command to the specified machine. However, in the case of the cluster databases, since the 12 of them need to be migrated simultaneously, the script sends the appropriate command to all machines in parallel. If any errors arise, only the database where it occurred is affected and can be analyzed without any concern that the remaining databases were affected.

In case an error occurs while a migration that iterates over several database schemas is being executed, the schemas the tool has already iterated over successfully are skipped when attempting to apply said migration again (with either a *up* or *apply* command). This behavior prevents the execution of duplicate code in schemas where the migration was already executed and is only

[5] https://www.postgresql.org/docs/11/manage-ag-tablespaces.html

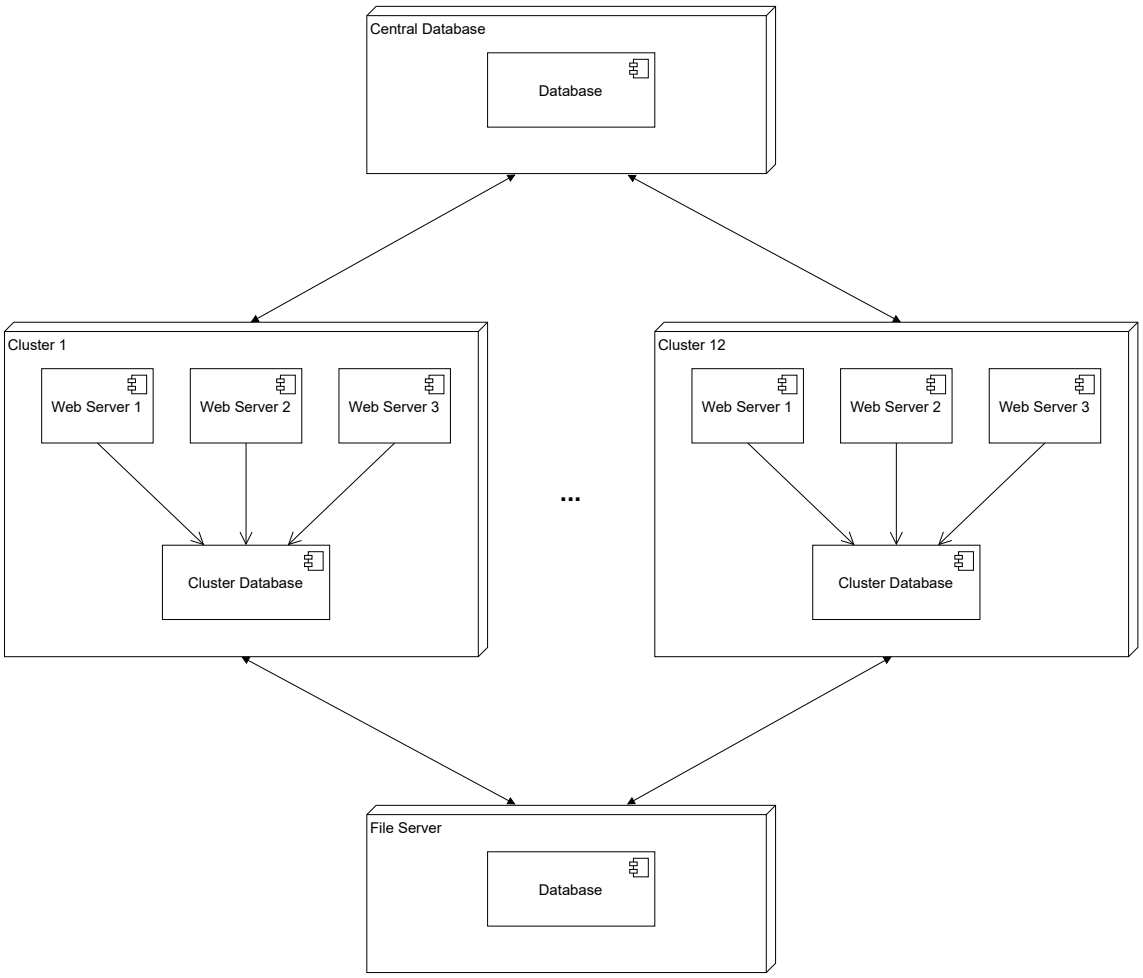

Figure 3.4: Deployment View of Cloudware's Infrastructure

possible because all schemas also possess a table that stores the timestamps of all migrations that have been applied to it.

3.6 Helper Methods

NANDO provides a series of other commands that assist the developer in performing the full scope of tasks necessary to migrate a database. These commands include the basic functionalities any database migrator requires, such as applying or reverting migrations and other tasks related to the initial setup of the tool. The syntactic details of these commands are described in Appendix A.

3.6.1 nando parse

When starting to use the tool, if the developer wants to migrate previous dbmate migrations to the NANDO format, the *parse* command can be used. First, the developer specifies the source and destination folders. The program then iterates over the source files converting them into NANDO

migrations, and maps them correctly between regular migrations and migrations without a transaction. This operation must be performed in separate source and destination folders since the program starts by clearing the destination folder, allowing this command to be executed multiple times.

3.6.2 nando baseline

To ensure that all functions that already exist in the database have a definition in a previous NANDO migration, the *baseline* command can be used. This command generates a migration with a separate `update_function` directive for each database function. This baseline migration is required in order for the *update* command to find a previous definition of any function in a previous migration to fill in the `down` method.

3.6.3 nando new

This command is used to create a new migration file for NANDO. This migration is a Ruby file with a class representing the migration, with two methods already created: the `up` and `down`. Using the `-t/--type` flag, the developer is able to specify the type of migration, between *Migration* or *MigrationWithoutTransaction*. The first type is a migration executed entirely inside a single database transaction (meaning that if an error occurs, the entire transaction is rolled back). A *MigrationWithoutTransaction* is executed outside of a transaction, meaning that any modifications that have already been executed will not be rolled back automatically if an error occurs. An example of a migration created by the *new* command can be seen in Listing 3.2.

```
1 class TestMigration < Nando::Migration
2   def up
3
4   end
5
6   def down
7
8   end
9 end
```

Listing 3.2: Example of a NANDO migration created by the *new* command

3.6.4 nando up

This command is used to apply all migrations that have not yet been executed. First, the program gets the list of valid migration files in the respective migration folder. A valid migration must have a name that starts with 14 digits (representing the migration's timestamp), followed by the migration's name. Finally, the ".rb" extension indicates it is a Ruby file. After getting this list of files, the program creates a table to store the applied migrations if such a table does not yet exist

(by default, named `schema_migrations`). If such a table already exists, the program gets the list of applied migrations and proceeds to iterate over the files, applying the migrations not yet in the `schema_migrations` table. Applying a migration consists of executing the code in the `up` method, and if no errors occur, then the migration version is added to the `schema_migrations` table.

When executing this command, the tool will look for all migrations that have not yet been applied, order them by timestamp and apply them in that sequence. Even if migrations with more recent timestamps have already been applied, any migration that is not yet present in the `schema_migrations` table will be applied when executing the *up* command.

3.6.5 nando apply

This command is used to apply a single migration, regardless of whether or not it has already been executed. It is intended only to be used during development due to how common it is to re-apply a migration while testing it. The command receives the migration timestamp as a parameter and then attempts to find a migration file with that exact value in its file name. If one is not found, an exception is raised. If the migration exists, it is executed, and the timestamp is added to the `schema_migrations` table if it was not present already.

3.6.6 nando down

This command is used to revert the latest migration that has been executed. The program starts by getting the most recent migration timestamp from the `schema_migrations` table. If there are no migrations recorded, the program terminates. However, if there are migration timestamps recorded but no migration file is found for the most recent recorded timestamp, an exception is raised. Finally, if there is a migration file for the most recent recorded timestamp, the migration is reverted. Reverting a migration consists of executing the code in the `down` method, and if no errors occur, then the migration timestamp is removed from the `schema_migrations` table.

3.7 Migration Process

The commands provided by NANDO are usually used in three main flows. These flows are mostly independent but can occasionally intercept each other. For example, when fixing a migration during the flow represented in Figure 3.7, the developer might need to perform the tasks present in Figure 3.6, such as using the *update* or *diff* commands.

The initial flow of setting up the tool (Figure 3.5) includes two commands: *parse* and *baseline*. When using the tool for the first time, if the developer is switching from dbmate to NANDO and wishes to preserve his history of migrations in some way, he can use use the *parse* command to transform his previous dbmate migrations into the equivalent NANDO migrations. If the table that keeps track of migrations (by default, it is the table `schema_migrations`) does not change, these parsed migrations will not be executed again. If the developer does not wish to keep the

history of migrations created before switching to NANDO, or if he is switching from another tool such as ActiveRecord to NANDO, then this setup step can be skipped. However, regardless of what tool was previously used, a step that is mandatory when starting to use NANDO is the creation of a baseline migration using the *baseline* command. This migration includes all the functions already present in the database and is necessary to ensure that future *update* commands find previous definitions of all existing functions.

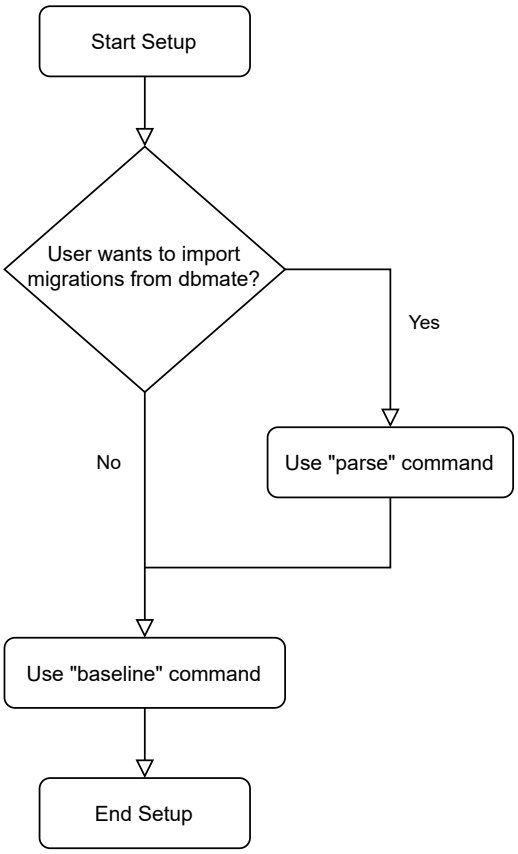

Figure 3.5: Setup Flow

The other main flow of the tool is the process of creating a migration (Figure 3.6). The developer starts by using the *new* command to create a new migration, specifying its type as *Migration* or *MigrationWithoutTransaction*. After the migration is created, the developer can then add any database functions he wants to update using the *update* command. This command can also receive an extra `-f/--function` flag to facilitate the addition of new functions. Suppose the developer has performed any structural changes to a specific schema during development and wants to propagate those changes to all schemas in this migration. In that case, he can run a *diff* command to get a list of changes detected by NANDO as well as the respective SQL command suggestions to apply those changes.

After the migration definition is completed, the developer can then apply it to the database, using either the *up* or *apply* commands (Figure 3.7). The *up* command applies all migrations that

have not yet been applied. In contrast, the *apply* command applies only the migration passed as a parameter, regardless of whether or not it has been applied before. Finally, if the developer wants to revert a migration, he can use the *down* command.

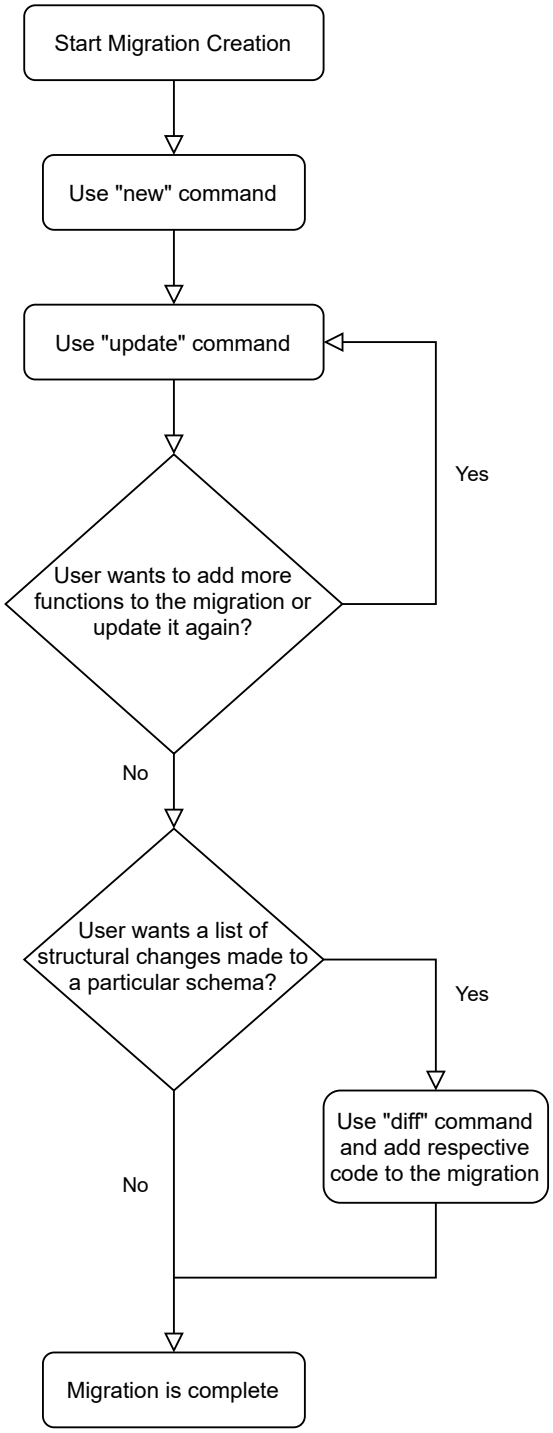

Figure 3.6: Migration Creation Flow

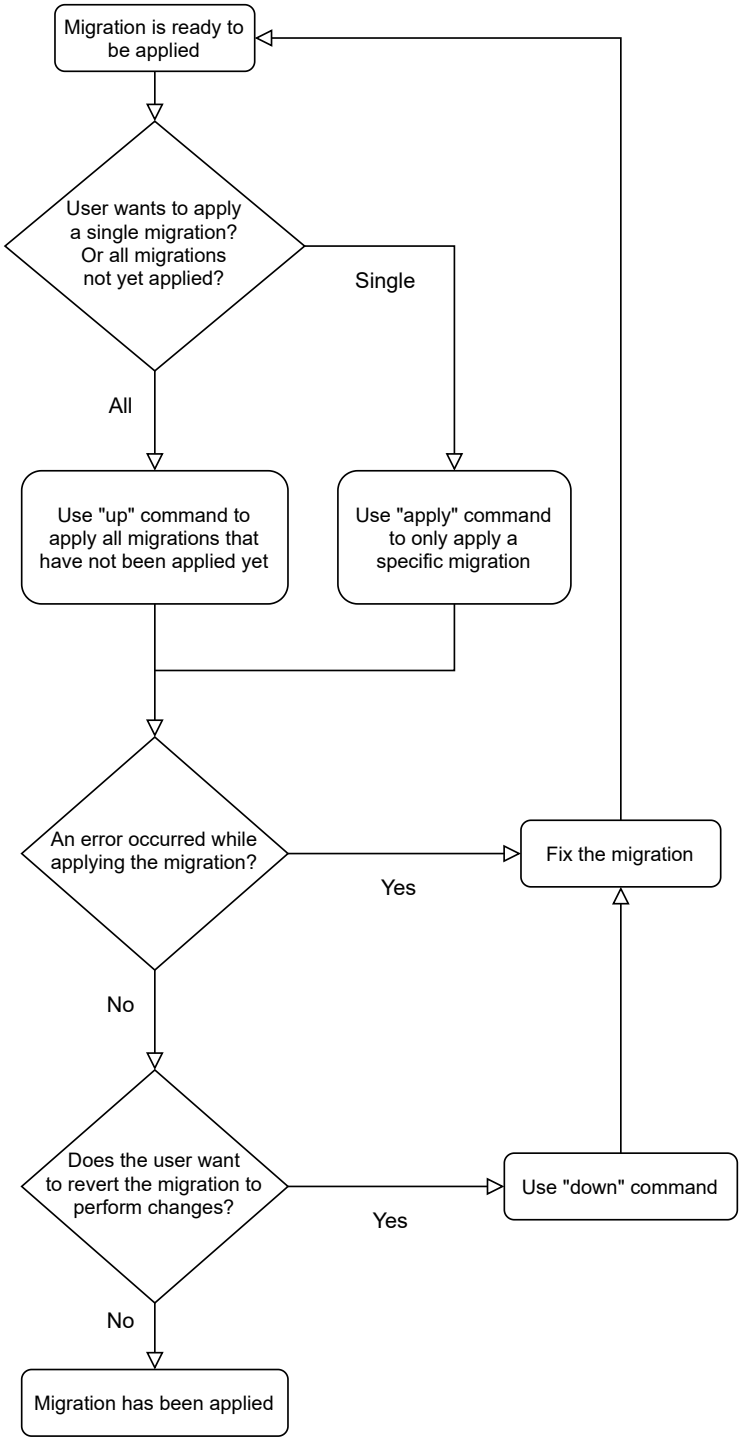

Figure 3.7: Migration Application Flow

Chapter 4

Validation

4.1 Tool Releases

In order to incrementally test the tool and receive continuous feedback from the developers at Cloudware, several releases of NANDO were performed. Some of these releases were only minor bug fixes found during testing, while others marked essential points in the development phase of the tool. All of these releases can be seen in https://rubygems.org/gems/nando/versions. RubyGems was chosen since it was the gem hosting service already used by Cloudware.

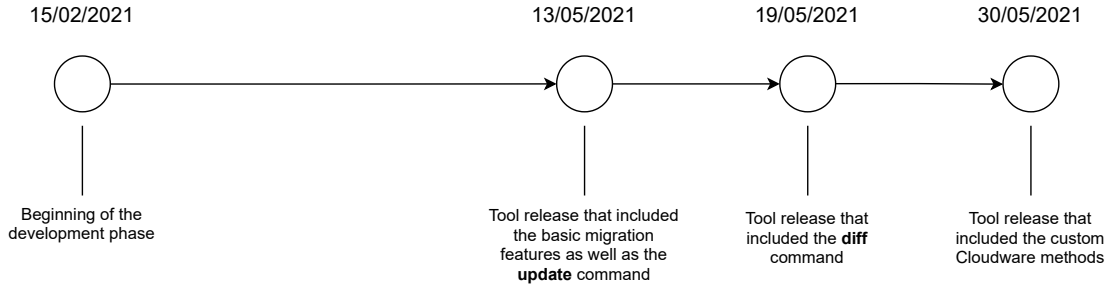

Figure 4.1: Tool Release Timeline

As can be seen in Figure 4.1, the development phase started on the 15th of February 2021. After that, the first release of the tool took place on the 13th of May 2021 and included the basic features a database migrator must have, as well as the implementation of the *update* command. This first release is described in Section 4.1.1, and includes versions 1.0.1 to 1.0.4. After this, on the 19th of May 2021, a new version of the tool (1.0.5) was released that included the *diff* command, as described in Section 4.1.2. Finally, the most recent release of the tool took place on the 30th of May 2021. This release included Cloudware's custom methods, used to iterate over groups of database schemas inside migrations, as is described in Section 4.1.3.

4.1.1 Versions 1.0.1 to 1.0.4

These releases all took place on the 13th of May 2021 and were the first versions of NANDO made publicly available. All these versions were grouped together since the only differences between them were fixes to common problems faced when deploying a Ruby gem for the first time, such as misused relative paths and packages being loaded with the wrong versions. The last of these releases, Version 1.0.4, represents the first iteration of NANDO that the developers could adequately use without dealing with problems unrelated to the tool's main features.

Version 1.0.4 provided the essential features a database migrator requires, such as creating, applying, or reverting a migration. Other than these basic functionalities, the tool also provided the necessary setup commands: *parse* and *baseline*, necessary to use the tool to its full extent. Furthermore, this release included the first iteration of the *update* command, one of the focal points of the tool. This version of *update* already performed the core operations necessary and would later on only receive minor fixes to bugs that appeared during testing.

4.1.2 Version 1.0.5

Version 1.0.5 was released on the 19th of May 2021 and included the first iteration of the *diff* command, another focal point of the tool. In addition to this feature, this release also included several fixes to most bugs that appeared during the testing performed with the previous versions.

Other than the *diff* command, the most relevant change is that the tool could now connect to password-protected databases, and developers could now specify a database using a hostname instead of requiring a static IP.

Several changes concerning code quality and readability were also performed, such as better error and exception handling and better error messages being presented to the developers.

4.1.3 Version 1.0.6

At the time of writing, Version 1.0.6 is the latest NANDO release. It was released on the 30th of May 2021. It included the last crucial feature expected of the tool: implementing Cloudware's custom methods to iterate over specific collections of schemas to apply migrations.

Other than this significant feature, this release also included fixes to bugs detected during the previous testing phases. The most relevant bug fixed in this version was affecting the *update* command, which was looking at all migrations when analyzing migrations to find a previous definition instead of just looking at past migrations.

This release also included changes to the architecture of NANDO, such as turning the main `NandoMigrator` class into a singleton, since there can only ever exist one instance, and this allowed for the class variables to be readable from anywhere in the program. Improvements to the code's quality were also performed, following this change into a singleton approach.

4.2 Tool Testing

Before releasing any of the tool's main features to the developers for testing, internal testing was performed to determine if NANDO was performing as expected. All of the tool's features were tested, from the essential functionalities a migrator must have, to the setup commands and finally, the tool's most critical features. These three main features had specific tests to determine if they were working correctly.

4.2.1 Basic Migrator Features

In the early development phase, the initial features implemented were the basic functionalities any database migrator requires. These functionalities had to be tested before progressing to the core features NANDO would implement.

The first feature was the ability to create database migrations (using the *new* command). This is the most straightforward operation the tool needed to perform since it essentially creates a file based on a template. The tool receives a migration name and an optional migration type, and it should create a file in the folder specified in the `MIGRATION_DIR` environment variable.

In order to test this behavior, manual tests were performed where the command would be executed with invalid parameters to check if all errors were being handled correctly. After this, valid parameters were passed to check that all available migration templates worked as intended. Finally, the `MIGRATION_DIR` environment variable was altered to test if the tool created files in the specified folders as well.

The second essential feature was the ability to apply and revert database migrations. Since this feature is critical to any database migrator, it was thoroughly tested. The expected behavior is to apply all migrations that have not yet been applied when running the *up* command and revert the most recent applied migration when running the *down* command. Furthermore, migrations that were already applied should be ignored by the *up* command, and migrations that have not been applied yet should be ignored by the *down* command.

To test the *up* command, multiple scenarios were considered:

- Applying migrations with timestamps more recent than the last applied migration
- Applying migrations with timestamps older than the last applied migration
- Attempting to apply a migration with an invalid name should raise a warning that the migration will be skipped by the tool
- Attempting to apply a migration that will produce an error should not mark the migration as applied. Furthermore, a migration of type `Migration` should roll back all of the changes executed before the error occurred, while a migration of type `MigrationWithoutTransaction` should not.
- Changing the value of the `MIGRATION_DIR` environment variable to determine if the correct folder of migrations is being considered

- Changing the value of the `MIGRATION_TABLE_NAME` or `MIGRATION_TABLE_FIELD` environment variables to determine if the correct table and respective field are being used to store applied migrations

To test the *down* command, multiple scenarios were also considered:

- Reverting migrations always reverts the last migration version applied
- When the migration to be reverted has a corresponding migration file, the migration is correctly reverted
- When the migration to be reverted no longer has a corresponding migration file (e.g., the file has been deleted, the file has an invalid name), an error is raised
- When there are no migrations to revert, an error is raised
- Attempting to revert a migration that will produce an error should not mark the migration as reverted. Furthermore, a migration of type `Migration` should roll back all of the changes applied before the error occurred, while a migration of type `MigrationWithoutTransaction` should not.
- Changing the value of the `MIGRATION_TABLE_NAME` or `MIGRATION_TABLE_FIELD` environment variables to determine if the correct table and respective field are being used to store applied migrations

4.2.2 Tool Setup Features

After the basic migration functionalities have been tested, some methods to assist in setting up the tool were developed. These commands were more formally tested since they were more complex in logic than the features previously mentioned. The two setup commands are *parse* and *baseline*.

The *parse* command expects two folder paths as parameters, and converts the dbmate migrations in the source folder into NANDO migrations in the destination folder. In order to test this behavior, a database from one of Cloudware's projects that was created using only dbmate was chosen, and a dump of the entire database schema was stored. Due to its low schema complexity but relatively large number of database migrations (around 200 at the time of writing), the Central Database was chosen. After that, the *parse* command was executed, creating a NANDO migration equivalent to each dbmate migration. These new migrations were then applied to a new database, and a dump of its schema was also stored. Finally, the dump of the original database was compared with the dump of the database created using the parsed migrations. These two dumps needed to be equivalent since only the schema was being stored, and the tool was successful in this test. The contents of the two dumps were compared using the Unix "diff" command[1], and no differences were found.

[1] https://ss64.com/osx/diff.html

The *baseline* command creates a migration with a separate `update_function` directive for every function currently in the database. This migration should make no changes to the database when applied since the tool should only re-assign every function's definition to its current definition. In order to test this, two of Cloudware's biggest projects in terms of the number of database functions were chosen. A dump for each database was created, containing every function definition in that particular database. Then, the *baseline* command was executed for each project, and the created migrations were applied. Following this, a second dump for each database was created and compared with the original dumps. For the tool to pass this test, both dumps for each project had to be exact matches. The dumps were, once again, compared using the Unix "diff" command and no differences were found, meaning the tool was successful in this test as well.

4.2.3 Update Command Testing

The *update* command was the first core feature of NANDO to be developed. When running an *update* command, the developer specifies a migration. The tool will then iterate over all the NANDO annotations in the `up` method, updating them with the code in the specified SQL file. Afterward, it will iterate over all the annotations in the `down` method and search for a previous definition of that function in all previous migrations.

In order to test the logic to fill the `up` method, multiple scenarios were considered:

- Trying to update annotations for files that exist should result in the individual file's content being inserted into the migration inside a respective `update_function` directive. Furthermore, if a respective annotation in the `down` method does not exist, one is created
- Trying to update annotations for files that do not exist should result in a warning being issued and no file contents being inserted
- Trying to update annotations for files that were already inserted in this migration should result in a warning being issued and no file contents being imported

Moreover, in order to test the logic to update the annotations in the `down` method, similar scenarios were considered:

- When updating an annotation, if a previous definition of the respective function exists in a previous migration, that definition (in the closest previous migration) is imported to the migration being updated
- When updating an annotation, if a previous definition of the respective function does not exist in any previous migration, a warning should be issued, and no other action should be taken

4.2.4 Diff Command Testing

After the *update* command, the *diff* command was the next core feature of NANDO developed. The command receives two database schemas and then proceeds to compare them, provides the developer with a summary of the differences detected and finally provides a suggestion of the necessary SQL commands to transform the source schema into the target schema and the changes required to transform the target schema into the source schema. These suggestions represent the code that should be included in the `up` and `down` methods, respectively. Any change detected leads to at least one respective SQL command to fix it, and when this is not possible, a warning is issued to the developer. Changes in tables, views, columns, triggers, indexes, and constraints were considered for this command, as can be seen in Table 3.1.

In order to test this functionality, we created multiple scenarios of similar schemas with minor differences to ensure the tool could handle them all correctly, such as differences in column types or mismatching trigger definitions. After this, we created a migration that performed all of the suggested code in the respective `up` and `down` methods. After applying this migration, the *diff* command would be executed again, and no differences should remain since the provided suggestions would have corrected them. Any remaining differences should have a corresponding warning to indicate they cannot be handled normally. Finally, to test the command suggestions of the `down` method, the migration is reverted, and the database should return to its original state. This was confirmed using both the *diff* command and by comparing database schema dumps generated before and after the test.

4.2.5 Testing Cloudware's Custom Methods

The final core feature that required testing was the custom methods previously developed by Cloudware that had to be ported over to *NANDO* to give developers all of the functionality they previously had, as well as all of the enhancements previously described. These custom methods iterate over specific groups of schemas in a database, such as all independent company schemas or all accounting schemas, and apply the code passed into them. Applying a migration using these methods should iterate over the necessary database schemas and insert the migration timestamp into the `schema_migrations` table of each schema. Reverting a migration is similar, except it removes the migration timestamp from the `schema_migrations` table of each schema.

Cloudware currently has 11 custom methods, and their functionality is described in Table 4.1. As we can see, all methods that iterate over a group of schemas come in pairs, with one method being used in the `up` method and the other being used in the `down` method. These iterating methods perform a query to the database to obtain the list of schemas they will iterate over and then execute the code present within their block. There is also the `table_exists` method that is used in conjunction with the other methods to allow some logic to be executed only if a given table exists or not.

To test all of the custom methods, we used Cloudware's main project since it already had the schema infrastructure required to test all of the methods. A migration was created to test every

Table 4.1: List of Cloudware's custom methods

Method Name	Method Description
migrate_companies	Used in the "up" method to iterate over all company schemas and apply a migration
rollback_companies	Used in the "down" method to iterate over all company schemas and revert a migration
migrate_accounting_companies	Used in the "up" method to iterate over all accounting schemas and apply a migration
rollback_accounting_companies	Used in the "down" method to iterate over all accounting schemas and revert a migration
migrate_fiscal_years	Used in the "up" method to iterate over all the fiscal years inside all accounting schemas and apply a migration
rollback_fiscal_years	Used in the "down" method to iterate over all the fiscal years inside all accounting schemas and revert a migration
migrate_user_schemas	Used in the "up" method to iterate over all the user schemas and apply a migration
rollback_user_schemas	Used in the "down" method to iterate over all the user schemas and revert a migration
migrate_user_templates	Used in the "up" method to iterate over all the user templates and apply a migration
rollback_user_templates	Used in the "down" method to iterate over all the user templates and revert a migration
table_exists	Used to determine if a specific table exists in a given schema

method, and inside the block of code that would be executed on every schema, a print message was included that lists all the properties that the method is providing to confirm the tool was applying/reverting the migration on the correct collection of schemas, and if it was providing all the information correctly.

Finally, in order to test the methods that come in pairs, migrations were created where the `up` and `down` methods were filled with the respective methods of each pair. Afterward, these migrations were applied and reverted similarly to as was described above to ensure the tool was behaving correctly.

4.3 User Testing

In order to compare NANDO to the tools previously used by Cloudware, we asked several of its developers to perform a series of tasks that covered the most common use cases for a database migrator since they will be the end-users of this tool. All tests involved the developers performing the same task using the previous migration tools (ActiveRecord or dbmate) and using NANDO, and

afterward, the respective times were recorded. Each test was performed once by each developer, and the results shown in Table 4.2 are the average times it took to complete each task.

All tests were performed in Cloudware's main project, where the previous migration tool was ActiveRecord. Four of Cloudware's developers were chosen to perform these tests. All of them had extensive experience using ActiveRecord and had between 5 and 23 years of experience in software development. They were given a script for each task before the time started to count to plan out how to perform each task optimally. Their results can be seen in Table 4.3; however, the developers' names were omitted.

4.3.1 Task 1 - Migration Updating 1 Function

This first task was created to demonstrate the difference in performance when performing one of the most common tasks that require a database migration: updating a single database function.

The developers were asked to start the test in a *git* branch that was previously created to simulate a real development scenario. The change to the function had already been performed to the respective SQL file, and the developer now has to create the corresponding migration.

Creating this migration using ActiveRecord involves:

- Generating a migration file, of type `Migration`
- Creating an `execute` directive in the `up` and `down` methods, that will contain the code to be executed
- To fill the `up` method, the developer must:
  - Open the SQL file containing the function that is being updated
  - Copy the file's content into the `execute` directive in the `up` method
- To fill the `down` method, the developer must:
  - Change to a *git* branch containing the previous definition of the function, traditionally the *master* branch
  - Open the SQL file containing the function that is being updated, since in this branch that file will contain the function's previous definition
  - Copy the file contents to the `execute` directive in the `down` method
  - Change back to the *git* branch that contains the change to the SQL file we are handling

In comparison, creating an equivalent migration using NANDO only requires the developer generating a migration of type `Migration` and running the *update* command, specifying the function he wishes to update using the `-f/--function` flag.

As can be seen in Table 4.2, completing this task took on average 185 seconds using ActiveRecord, while with NANDO it took only 24 seconds on average. This represents a decrease of around 87% in time spent performing the same task, a rather significant improvement. The

fact that NANDO automates the process also minimizes any potential human error and ensures the process can be repeated as many times as needed if further changes to the SQL file are required. During this test, no errors were committed by the developers while creating the migrations manually. However, errors while creating migrations manually occur frequently during a regular development cycle.

NANDO also ensures that the `down` method is filled properly. This was not always the case with the manual approach in current practice since developers often skipped this step because it was inconvenient and tedious.

4.3.2 Task 2 - Migration Updating 3 Functions

This second task was created to demonstrate that the time required for the manual approach increases almost linearly with the number of functions to update. However, using NANDO the time required stays almost the same. In this test, the task is very similar to Task 1; however, instead of updating just one database function, the developer must update three.

Similar to the previous task, the developers were asked to start in a *git* branch that was previously created to simulate a real development scenario. The changes to the three functions had already been performed to the respective SQL files, and the developer now has to create the corresponding migration.

The steps to create this migration using ActiveRecord are very similar to the one in Task 1. The only difference being that instead of creating one `execute` directive and adding the source code of one file, now these tasks had to be done once for every function to update.

In the case of NANDO, the process is also very similar to the previous task. However, when calling the *update* command, the user can pass all SQL functions to the *update* command in bulk, meaning that the number of operations the developer must perform stays the same.

As can be seen in Table 4.2, completing this task took on average 244 seconds using ActiveRecord, while with NANDO it took only 28 seconds on average. This represents a decrease of around 89% in time spent performing the same task, an even greater improvement than in the previous test.

When comparing these results with the previous test, we can see that the ActiveRecord approach had an increase of around 32% in time spent, while using NANDO only had an increase of around 16%. Moreover, this difference is primarily due to the time the tool takes to perform the required tasks automatically since the number of actions required by the user stayed the same. In the case of ActiveRecord, the increase in time required is due entirely to tasks that have to be performed manually, making the increase in time even more relevant.

4.3.3 Task 3 - Schema Changing Migration

This final task was created to compare the performance of ActiveRecord and NANDO when creating a migration that performed a change in multiple schemas.

The steps to create this migration using ActiveRecord and NANDO are very similar. With both tools, the developer must generate a migration file of type `MigrationWithoutTransaction` and call the appropriate Cloudware custom method to iterate the necessary schemas (such as `migrate_companies`, for example) in both the `up` and `down` methods. The difference is that in the case of ActiveRecord, the developer must write all the schema changes manually. In contrast, in the case of NANDO, if the developer has already performed the changes on a specific schema in the database, the *diff* command can be used to provide the necessary SQL commands to apply those changes to all schemas.

As can be seen in Table 4.2, completing this task took on average 166 seconds using ActiveRecord, while with NANDO it took only 92 seconds on average. This represents a decrease of around 44% in time spent performing the same task. This decrease is not as drastic as in the other two tests, which is to be expected since NANDO only automates the process of writing the necessary SQL commands. The developer still has to specify the custom method he wants to use, just like with ActiveRecord. However, this increase is still relevant, and the tool also adds a layer of confidence to the migrations created since no other schema differences were found, other than the ones presented to the user, thus reducing any potential human error caused by forgetting to include certain schema changes migration.

4.3.4 Results Analysis

Looking at Table 4.2, it is evident that NANDO had a significant decrease in time spent performing the same tasks. Task 1 had a decrease of 87%, while Task 2 had a decrease of 89%. This shows that increasing the number of functions to update in a single migration does not affect the relative performance increase of NANDO when compared with the manual approach. In fact, the relative performance gained is even higher when the number of functions increases, which is to be expected since NANDO makes this process almost entirely automatic.

In regards to Task 3, using NANDO proved to be 44% faster than following the manual approach. As was previously mentioned, this increase is not as significant as with the previous two Tasks, since NANDO only makes part of the process automatic, leaving the programmer to still specify things such as the custom Cloudware method to use. However, while the time gained is not as significant as with the other two Tasks, using NANDO does provide the developer extra confidence in the migration being created.

Table 4.2: Average time spent performing each task, in seconds

	ActiveRecord	**NANDO**
Task 1 - Migration Updating 1 Function	185	24
Task 2 - Migration Updating 3 Functions	244	28
Task 3 - Schema Changing Migration	166	92

Table 4.3: Time spent by each developer performing each task, in seconds

	Task 1 (ActiveRecord)	**Task 1 (NANDO)**	**Task 2 (ActiveRecord)**	**Task 2 (NANDO)**	**Task 3 (ActiveRecord)**	**Task 3 (NANDO)**
Developer 1	152	23	238	29	164	86
Developer 2	172	25	312	21	173	104
Developer 3	166	20	283	24	185	99
Developer 4	250	28	142	38	141	80

Another important metric to consider is the number of lines of code written manually when compared to using NANDO. As can be seen in Table 4.4, for Task 1 and Task 2, NANDO increases the percentage of automatic lines from 3% and 1%, respectively, to 100%. While using ActiveRecord, the only lines generated automatically come from the template used when creating a new Migration. In contrast, NANDO fully automates this process, only requiring the developer to list the files containing the functions to update.

When looking at Task 3, the increase in automatic lines when comparing ActiveRecord with NANDO is not as drastic as with the previous Tasks, going from 31% to 50%. As was previously explained, this is because NANDO only automatically generates the SQL code that will be executed. With either tool, the developer still has the responsibility of writing the loop using one of Cloudware's custom methods. Similarly to the previous two Tasks, the only lines automatically written with ActiveRecord come from the template used when creating a new Migration. While the increase in automatic lines in Task 3 is not as drastic as with the previous tasks, if the migration were to have a large number of schema changes to apply, NANDO's performance in this aspect would increase since it automatically generates the SQL commands necessary to perform said changes. With ActiveRecord, the developer would still have to write this SQL code manually.

Table 4.4: Comparison of manual and automatic lines using NANDO and ActiveRecord

	ActiveRecord			**NANDO**		
	Manual Lines	**Automatic Lines**	**% of Automatic Lines**	**Manual Lines**	**Automatic Lines**	**% of Automatic Lines**
Task 1	292	10	3%	0	305	100%
Task 2	719	10	1%	0	736	100%
Task 3	22	10	31%	16	16	50%

To get a better understanding of how much this reduction is relevant to Cloudware, we determined how often migrations are created to estimate the overall gain the company will get from using NANDO. We looked at Cloudware's main project since most migrations are created to update it, and as such, it represents a very significant example.

The project is around ten years old, and in that time, 4547 migrations have been created for it, at the time to writing. The project used ActiveRecord as its database migrator. If we assume an average work year of 252 days, we would have around 2520 workdays over ten years. Dividing the number of migrations by the number of workdays, we reach an average of 1.8 migrations created per workday, or around nine migrations per week.

We started by estimating the average number of lines a migration tends to have. Around

16% of all migrations are of type `MigrationWithoutTransaction`, while the rest are of type `Migration`. Assuming that migrations without transaction represent all of the migrations that make changes to the schema, and assuming the remaining 84% only affect database functions, we can estimate that, on average, migrations that affect the database schema have around 323 lines. In comparison, migrations that update database functions have an average of 214 lines. Since developers tended to ignore filling the `down` method in a large portion of these migrations, the average number of lines should, in theory, be even greater.

With this information, migrations that change the database schema represent around 235 000 lines of code, while migrations that update database functions represent around 820 000 lines of code.

Taking a look at the results shown in Table 4.4, we can see that only around 1% to 3% of the 820 000 lines in migrations that update database functions were automatically written, meaning that the remaining lines were filled in manually. Considering the results from Task 1 and Task 2, since in these scenarios no manual lines were required, NANDO would completely automate these types of migrations, representing a massive improvement over the previous manual approach. However, it is relevant to keep in mind that in migrations that update functions, a large amount of the lines classified as "manual" when using ActiveRecord were actually copied and pasted from the respective SQL files into the migration and not actually manually written.

While estimating the overall gain in migrations that update the database schema is harder, based on the results of Task 3, we can observe that the number of automatic lines in ActiveRecord migrations is constant, the ten lines that come from the migration template. In the case of NANDO, as was previously mentioned, the manual lines are also relatively constant since they represent the calls to Cloudware's custom methods. With this information and the average of 323 lines in these types of migrations, we can estimate that with ActiveRecord 10 lines out of 323 are automatic, representing only around 3% of their content. With NANDO however, assuming on average 16 manual lines are required, this means that around 307 lines are automatic. This represents an increase of up to 95% of automatic lines, a much higher value than the 50% recorded in Task 3.

In order to estimate the overall time gained if NANDO had been used to create the previous migrations, we need to determine the average relative gain from both types of migrations. For migrations that update database functions, by observing the time decrease in Table 4.2 for Task 1 and Task 2, the relative time decrease was 88% on average. Moreover, looking at Task 3, we can see the relative time decrease was of around 44%; however, in this scenario, Task 3 is not representative of the time gained with using NANDO, since the required migration for this task only needed 32 lines, while the average migration that updates the database schema has around 323 lines, as seen above. However, even considering the time gained in these migrations is potentially much more significant, the average time decrease if NANDO had been used to write all of the previous migrations is still around 81%.

In summary, and as can be seen in Table 4.5, of the 820 000 lines of code from migrations that update functions, an estimated 100% of their lines could have been automatically written if NANDO would have been used, while ActiveRecord only had around 1% to 3%. When looking

Table 4.5: Comparison of performance between ActiveRecord and NANDO

	Function Migrations	**Database Schema Migrations**
% of total migrations	84%	16%
Average number of lines per migration	214	323
Total number of lines across all migrations	820 000	235 000
Average % of automatic lines using ActiveRecord	1%-3%	3%-31%
Average % of automatic lines using NANDO	100%	50%-95%
Average % of time reduction if NANDO had been used to create all previous migrations	88%	44%
Overall % of time saved in the process of creating database migrations	81%	

at the 235 000 lines of code from migrations that update the database schema, 50% to 95% would be automatically written if NANDO was used, compared to 3% to 31% when using ActiveRecord. This large discrepancy in these types of migrations stems from the fact that the size of these migrations heavily influences the relative improvement provided by NANDO.

When looking at the overall time gained in the process of creating database migrations, using NANDO would have decreased time spent by around 81%. However, this estimate is reasonably conservative since we are considering that migrations that update the database schema only take 44% less time being created. This value could be much higher, as was explained above. In addition to these improvements, there are also other advantages of using a tool to automate the process of creating migrations that are not as quantifiable. These include reducing many human errors that would have previously occurred with a manual approach and giving developers more confidence in the migrations being created.

4.3.5 Usability Survey

In order to assess the usability of the tool, a small usability survey was developed, based on the System Usability Scale (SUS) [1]. This survey was presented to the developers that performed the tests previously described.

The survey had the following five questions:

1. I think that I would like to use this tool frequently.
2. I thought the tool was easy to use.
3. I would imagine that most people would learn to use this tool very quickly.

4. I felt very confident using the tool.

5. In the future, I would prefer to use NANDO over the previous migration tools.

Each question was answered on a scale ranging from 1 to 5, where 1 represented "Strongly disagree" and 5 represented "Strongly agree". At the end of the survey, there was also an optional question for potential suggestions for the tool. The overall answers to the survey can be found in Table 4.6, and an average of the results can be seen in Table 4.7.

Table 4.6: Developers answers to the usability survey

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Suggestions for the tool**
Developer 1	5	5	4	5	5	It was amazing. Easy. Fast. Safe. Just keep improving the visual output.
Developer 2	5	4	5	5	5	-
Developer 3	5	5	5	4	5	-
Developer 4	5	4	5	4	5	-

As can be seen in Table 4.7, overall, the results were very positive in favor of NANDO. All developers believed that they would like to use the tool frequently (Question 1) and that, in the future, they would prefer to use NANDO over the previous migration tools (Question 5). The questions with the lowest score were Questions 2 and 4, and they questioned the developers about the degree of difficulty and confidence using the tool, respectively. Although the answers were still very positive, this shows there is potentially still room for improvement in these areas.

Table 4.7: Average of the developers' answers to the usability survey

Question 1	**Question 2**	**Question 3**	**Question 4**	**Question 5**
5.0	4.5	4.8	4.5	5.0

In regards to the optional questions related to potential suggestions, only one developer provided an answer. This answer suggested that the tool should continue to improve on its visual output. Enhancing this aspect could lead to an easier understanding of the tool, minimizing some of the less favorable results gathered from the survey.

Chapter 5

Conclusions

5.1 Achievements

As was mentioned in the introduction, the main goal of this dissertation was to develop an approach capable of streamlining and automating the process of creating database migrations. After analyzing the state of the art in Chapter 2, we determined that neither the tools being currently used by Cloudware (dbmate and ActiveRecord) nor any of the other tools examined had the complete set of features required to solve the problem at hand. This led to the decision to create a new tool that included all of the necessary functionalities, which was named NANDO. This tool's design and implementation is described in Chapter 3, covering its architecture and crucial functionalities, as well as the main workflows available.

Chapter 4 goes over the process of validation that was followed. It includes a list of major releases performed; a description of the tests performed before each release; a summary of the tests performed with actual users; an analysis of the performance of NANDO compared to previous tools; and finally, a review of the usability tests performed. A usage manual for NANDO was included in Appendix A. This manual covers all of the available commands and their respective flags and a list of all environment variables the tool provides.

Since ActiveRecord was one of the tools previously used by Cloudware and the one that offered the most functionality, several aspects from it were replicated when developing NANDO. These include:

- Migrations are written in Ruby code and follow a similar format in order to minimize the learning curve for developers that have previously used ActiveRecord;

- All functionality from ActiveRecord was preserved, including the ability to use Ruby variables interpolated with SQL code, and the ability to specify if a migration should be executed inside a transaction or not;

- All of the custom methods Cloudware had previously implemented to allow ActiveRecord to execute migrations in multiple schemas were also ported to NANDO;

Besides maintaining all of the previously available functionality, several improvements were also added to the tool in order to solve the problem at hand, making the process of creating database migrations more automatic, and thus reducing human error and time spent in these tasks.

In order to accomplish that goal, we identified the two most time-consuming tasks that were performed during the process of creating migration: the process of updating database functions and the process of writing SQL code for changes to the database schema. Besides being time-consuming, these tasks were also performed manually, which lead to occasional human errors that could be avoided if the process was automated.

To minimize the process of creating migrations that update database functions, the approach we elected uses the contents of the respective SQL files to fill the `up` method and uses the history of migrations to determine what version of each function will fill the `down` method. This approach was chosen since it minimizes the number of actions a developer has to take to create a migration and makes it easy to understand where the tool is getting the function definitions it is importing to the current migration. In addition, a system of annotations was implemented to facilitate the tracking of which functions are being updated in each migration.

In order to minimize the process of writing SQL code for changes to the database schema, we focused on the most common workflow during development. This workflow involves making these changes on a particular database schema to test them and then writing a migration to apply these changes to all schemas. As such, a way for the developers to determine which differences exist between two database schemas was developed to ensure all changes performed are included in the migration.

Both of these features can be used in a single migration. As such, if the developer wishes to update database functions and perform schema changes, he can do so without creating multiple migrations.

As was detailed in Chapter 4, when observing the overall time spent in the process of creating database migrations, NANDO would reduce this metric by an average of 81% compared to ActiveRecord. Furthermore, this estimate is reasonably conservative since we are considering that migrations that update the database schema would take 44% less time, a value that could be much higher, as explained in Section 4.3.4. Moreover, besides this reduction in time spent creating database migrations, using NANDO also reduces a large portion of human errors that would have previously occurred with a manual approach and gives developers more confidence in the migrations being created.

5.2 Future Work

There is potential future work achievable for NANDO. These possibilities include secondary "quality of life" functionalities and major features comparable to the modules developed for this dissertation.

The first of these improvements is to automate further the process of creating migrations that make changes to the database schema. As was observed in Chapter 4, this process had the lowest increase of performance when using NANDO, since a large portion of the migration code still has to be written by the developer. A potential solution to this would be introducing a method for developers to specify the custom method they want to apply. Then the tool would add it to the migration, potentially with the code suggestion from the *diff* command already imported. This feature should further decrease the time spent creating these types of migrations.

Another potential improvement that could be added to NANDO would be the ability to revert a specific migration, similarly to the *apply* command. This operation is occasionally helpful during development to test if the `down` method of a migration is working as intended. As such, the ability to test this with any migration without having to revert all applied migrations with a higher timestamp could be a convenient functionality.

The final feature that could benefit NANDO would be the ability to merge multiple migrations into a single one, removing any duplicate redefinition of functions or unnecessary calls to Cloudware's custom methods. This functionality could be helpful when a large development cycle occurs, leading to many intermediate migrations being created. When the cycle is over, instead of applying multiple migrations that likely redefine database functions multiple times and change database tables, a single migration that contained all of the changes performed would be preferred. It would allow for a better understanding of which aspects of the database are being changed and minimize the number of times the database schemas are iterated using Cloudware's custom methods. This feature would have to:

- Determine all functions that were updated in previous migrations in order to add them to the final migration while ignoring duplicates;
- Merge all actions that do not update database functions or schema, such as updating the data in a public table, shared by all companies;
- Merge all of the code inside the multiple calls to Cloudware's custom methods in order to prevent them from iterating multiple times over the same sets of the database schemas;

NANDO's performance will continue to be tracked by Cloudware. The need for other functionalities not listed here may arise, depending on potential new requirements that become apparent due to an increase in the tool's usage. In the short period of this dissertation, NANDO has reached a very interesting level of functionality and maturity that ensures the tool will be fully integrated into the production pipeline soon.

Appendix A

Usage Manual

In order to facilitate the learning process of using NANDO, a simple usage manual was created. It includes information about all the available commands, potential flags, and available environment variables the tool provides.

A.1 Installation

In order to install the tool, execute the following command:

```
gem install nando
```

At the time of writing, the most recent version of NANDO is 1.0.6 and includes all of the features described in this manual.

A.2 Tool Commands

The commands listed below use the values of the environment variables described in Section A.3. Of those, only `DATABASE_URL` is mandatory in order to specify which database the tool should interact with.

A.2.1 nando baseline

When starting to use NANDO on a project, the first command that is required is the *baseline* command. It creates a migration with a separate `update_function` directive for every function in the database. This migration is required in order for the *update* command to find previous definitions of every existing function when filling the `down` method.

This command receives no parameters and can be executed multiple times, creating a new migration each time. Applying these migrations makes no change to the database's state since every function is being redefined with its current definition.

Command syntax: `nando baseline`

Example: `nando baseline`

A.2.2 nando parse

Another optional command that could be executed when starting to use NANDO is the *parse* command. This command converts dbmate migrations into NANDO migrations if the developer wants to maintain the history of migrations files.

This command receives two parameters: the path to the source folder, where the original dbmate migrations exist, and the path to the target folder, where the parsed migrations will be created. However, the target folder will be emptied before parsing the migrations to allow this command to be executed multiple times without creating duplicates of each previous migration.

Command syntax: `nando parse <source folder> <target folder>`

Example: `nando parse db/old_migrations db/new_migrations`

A.2.3 nando new

After all the setup is done, to start working with the tool, a migration file is required, and to generate one, the *new* command is used. It creates an empty NANDO migration in the folder specified by the environment variable `MIGRATION_DIR`. The migration file name follows the format:

```
<migration creation timestamp>_<migration name>.rb
```

This command expects one parameter, the migration's name. This name can be in any format (snake case, camel case, or Pascal case) since the tool will then convert it to snake case when generating the file name and convert it to Pascal case when giving the migration class its name.

The `-t/--type` flag can be used to specify the type of migration to be created. By default, all migrations are created with type `Migration`. However, by specifying this flag, a migration can alternatively be created with type `MigrationWithoutTransaction`.

Command syntax: `nando new <migration name>`

Example: `nando new AddingColumnToUsersTable -t Migration`

A.2.4 nando up

This command is used to apply all migrations that have not yet been executed. To know which migrations have already been applied, the tool stores this information in a table within the database (see A.3.2 for more information). Applying a migration consists of executing the code within its `up` method, and registering its version on the table specified by the `MIGRATION_TABLE_NAME` environment variable.

This command expects no parameters. When it is executed, it applies all migrations in the folder specified in the environment variable `MIGRATION_DIR` that have not yet been applied. Migrations that have already been applied are ignored.

Command syntax: `nando up`

Example: `nando up`

A.2.5 nando down

This command reverts the latest migration that has been applied. The tool determines the latest migration by looking at the most recent version in the table specified by the `MIGRATION_TABLE_NAME` environment variable, since these versions represent the timestamps of the migrations. Reverting a migration consists of executing the code within its `down` method, and removing its version from the table specified by the `MIGRATION_TABLE_NAME` environment variable.

This command expects no parameters. If there are no migrations to revert, an error is thrown to warn the developer.

Command syntax: `nando down`

Example: `nando down`

A.2.6 nando apply

This command is similar to *up*; however, it only applies a single migration, and not all of the migrations yet to be executed. It applies the specified migration regardless of whether or not it has already been executed. It is intended to be used during development, where reapplying migrations to test them is a recurrent operation. Usage in a production environment is not advised.

This command expects one parameter, the version of the migration to be applied. This version is the timestamp present in the migration file name.

Command syntax: `nando apply <migration version>`

Example: `nando apply 20210604134225`

A.2.7 nando update

This command is used to update a NANDO migration by checking all annotations and inserting the corresponding source code for each function. This command updates both the `up` and `down` methods of the migration with the necessary code. It can be executed as many times as necessary without any consequences.

The format of annotations for the `up` method is `"# NANDO: <path to file>"`, and for the `down` method is `"# NANDO_DOWN: <path to file>"`.

This command expects one parameter, the path to the migration to update.

The `-f/--function` flag can be used to add more annotations to a migration, which will then be updated as well. This flag can either receive a single path to a file to add to the migration; or receive no value, which causes the tool to open an input in the interface, allowing the user to paste a list of files to add in bulk to the migration.

Command syntax: `nando update <path to migration>`

Example: `nando update db/migrate/20210604134225_test_migration.rb`

A.2.8 nando diff

This command compares two database schemas to determine if there are any structural differences between them. It also provides the developer with the necessary commands to turn the source schema into the target schema and the commands to turn the target schema into the source schema.

This command expects two parameters: the source schema and the target schema. The generated commands fills the target schema name using the value specified in the environmental variable `SCHEMA_VARIABLE`.

Command syntax: `nando diff <source schema> <target schema>`

Example: `nando diff schema_1 schema_2`

A.3 Environment Variables

In order to accommodate a large variety of projects, NANDO provides a series of environment variables to allow users to configure the tool for their specific setup. These variables must be specified in an `.env` file at the root of the project the tool is being used on. In Section A.3.8, an example of a NANDO `.env` file was included.

A.3.1 DATABASE_URL

This environment variable is the only one that is mandatory by the tool. Without it, NANDO would have no indication of which database it must apply or revert migrations, for example. This variable follows a format similar to dbmate, and expects a URL that follows this form:

```
<protocol>://<username>:<password>@<host>:<port>/<database_name>
```

Currently, the only supported protocol is PostgreSQL since this is the type of database used in all of Cloudware's projects.

A.3.2 MIGRATION_TABLE_NAME

As is common among database migrators, to know which migrations have been applied, the tool stores this information in a table inside the database being migrated. By default, this table is `schema_migrations`; however, if a developer wishes to store this information in a table with a different name, he can do so using this environment variable. If this table does not exist when the developer migrates a database, the tool creates it to store the versions of the migrations already applied.

A.3.3 MIGRATION_TABLE_FIELD

Similar to the previous environment variable, if the user wants to specify the column's name in the table that stores the applied migrations, he can do so using this variable. By default, its value is `version`.

A.3.4 MIGRATION_DIR

This environment variable is used to specify the relative path to the folder new migrations will be created, and the folder the tool will search for more migrations to apply. Its default value is `db/migrate`. This variable's value is also used in the *update* command to determine which folder will be used to search for previous definitions of the functions being updated.

A.3.5 WORKING_DIR

This variable specifies the path to the working directory the tool will use as a base when searching for the SQL files listed in the annotations of the migrations. By default, its value is the current directory the tool is being used in. However, if the developer wants to use NANDO commands from any folder of a project, he can specify the path to the root of said project. Doing so will ensure the tool will always look for SQL files in the current project, followed by the functions' file relative path inside said folder.

A.3.6 SCHEMA_VARIABLE

When generating the suggestions of SQL commands from the *diff* command, the tool uses this variable's value to fill any occurrences of the schema's name. This is done since the generated code is intended to be used in conjunction with Cloudware's custom methods, which use a variable to fill specific parameters such as the schema's name dynamically. By default, this environment variable's value is `#{schema_name}`, since this is the parameter name passed by Cloudware's methods. However, if different methods are implemented/used, a different variable name can be specified using `SCHEMA_VARIABLE`.

A.3.7 DEBUG

Finally, the last environment variable available is used for setting the tool to debug mode. While in this mode, when executing commands, the tool will also print a large amount of information to help developers understand what actions are being performed. In order to use debug mode, the variable's value must be set to `true`.

A.3.8 Practical Example

In Listing A.1 we can see an example of a NANDO `.env` file with all of its potential environment variables. As was previously explained, only the `DATABASE_URL` is mandatory. The remaining variables could be missing from the `.env` file and the tool would use the respective default values.

```
1 DATABASE_URL="postgres://toconline:toconline@localhost:1234/nando_testing"
2 MIGRATION_TABLE_NAME="schema_migrations"
3 MIGRATION_TABLE_FIELD="version"
4 MIGRATION_DIR="db/migrations"
```

```
WORKING_DIR="~/Users/projects"
SCHEMA_VARIABLE="#{schema_name}"
DEBUG=true
```

Listing A.1: Example of a NANDO `.env` file

References

[1] Assistant Secretary for Public Affairs. System usability scale (sus). https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html, Sep 2013.

[2] José Andany, Michel Leonard, and Carole Palisser. Management of schema evolution in databases. In *VLDB*, pages 161–170, 1991.

[3] Donald Bell. Uml basics: An introduction to the unified modeling language. *The Rational Edge*, 2003.

[4] Carlo Curino, Hyun J Moon, and Carlo Zaniolo. Automating database schema evolution in information system upgrades. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 1–5, 2009.

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

[6] Carlo A Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. *Proceedings of the VLDB Endowment*, 4(2):117–128, 2010.

[7] Julien Delplanque, Anne Etien, Nicolas Anquetil, and Olivier Auverlot. Relational database schema evolution: An industrial case study. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 635–644. IEEE, 2018.

[8] Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. Codel–a relationally complete language for database evolution. In *East European Conference on Advances in Databases and Information Systems*, pages 63–76. Springer, 2015.

[9] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1101–1116, 2017.

[10] Kai Herrmann, Hannes Voigt, Torben Bach Pedersen, and Wolfgang Lehner. Multi-schema-version data management: data independence in the twenty-first century. *The VLDB Journal*, 27(4):547–571, 2018.

[11] James E McDonough. Singleton design pattern. In *Object-Oriented Design with ABAP*, pages 137–145. Springer, 2017.

[12] Data Pilot. Function vs stored procedure in sql. https://kb.objectrocket.com/postgresql/function-vs-stored-procedure-602, Jul 2019.

[13] Robert Schuler and Carl Kesselman. Chisel: a user-oriented framework for simplifing database evolution. *Distributed and Parallel Databases*, pages 1–61, 2020.

[14] Robert E Schuler and Carl Kessleman. A high-level user-oriented framework for database evolution. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pages 157–168, 2019.