DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Automated Testing of Database Schema Migrations

PETER JONSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Automated Testing of Database Schema Migrations

PETER JONSSON

Master in Computer Science
Date: June 28, 2019
Supervisor: Johan Gustavsson
Examiner: Elena Troubitsyna
School of Electrical Engineering and Computer Science
Host company: The Swedish Police Authority
Swedish title: Automatiserad testning av databasschemaförändringar

Abstract

Modern applications use databases, and the majority of them are relational databases, which use schemas to impose data integrity constraints. As applications change, so do their databases. Database schemas are changed using migrations. Certain conditions can result in migrations failing in production environments, leading to a broken database state and testing can be problematic without accessing production data which can be sensitive.

Two migration validation methods were proposed and implemented to automatically reject invalid migrations that are not compatible with the database state. The methods were based on, and compared to, a default method that used Liquibase to structure and perform migrations. The assertion method used knowledge of what a valid state would look like to generate pre-conditions from assertions to verify that the database's state matched expectations and that the migrations were compatible with a database's state prior to migration. The schema method, used a copy of the production database's schema to perform migrations on an empty database in order to test the compatibility of the old and new schemas. 108 test cases consisting of a migration and a database state were used to test all methods. Both valid and invalid test cases that were not compatible with the database's state were used. The distribution of aborted, failed, and successful migrations were analyzed along with the automation, traceability, reliability, database interoperability, preservability, and scalability for each method.

Both the assertion method and the schema method could be used to stop most of the invalid migrations without access to the production data. A combination of the assertion method and the schema method would result in only 2/108 migrations failing and could reduce the failure rate even further through using a schema to reduce complexity for uniqueness constraints and to improve support for handling data type conversions.

Sammanfattning

Moderna applikationer använder ofta relationsdatabaser som har en strikt regeluppsättning för att garantera dataintegritet. När applikationerna förändras måste även deras regeluppsättningar göra det. Omständigheter kan leda till att databasförändringar inte går att genomföra i produktionsmiljön, vilket resulterar i ett trasigt databastillstånd. Testning av förändringar kan vara problematiskt utan tillgång till produktionsdata, men detta kan vara svårt då produktionsdatan i sig kan vara känslig.

Två valideringsmetoder föreslogs och implementerades för att automatiskt stoppa ogiltiga förändringar som inte är kompatibla med databasens tillstånd. Metoderna grundades i, och jämfördes med, en grundmetod som använde Liquibase för att strukturera och genomföra databasförändringar. Påståendemetoden använde kunskap kring vad som är ett giltigt tillstånd för att generera villkorssatser som verifierar att databasens tillstånd matchar förväntningarna innan förändringen genomförs. Regelmetoden använde en kopia av produktionsdatabasens regeluppsättning för att exekvera förändringen på en tom databas för att testa kompatibiliteten mellan den gamla regeluppsättningen och den nya. 108 testfall användes och bestod av förändringar samt tillstånd. Både giltiga och ogiltiga testfall som inte var kompatibla med databasens tillstånd användes. Distributionen av avbrutna, misslyckade och lyckade förändringar analyserades i faktorer som automation, spårbarhet, pålitlighet, databasinteroperabilitet, konserverbarhet samt skalbarhet för varje metod.

Både påståendemetoden och regelmetoden kunde användas för att stoppa de flesta ogiltiga förändringarna utan tillgång till produktionsdata. En kombination av påståendemetoden och regelmetoden skulle resultera i att enbart 2/108 förändringar misslyckades och kunna nå ännu lägre felfrekvens genom att analysera databasschemat för att reducera komplexiteten som krävs för vissa unikhetskrav och vidare öka stödet för konvertering av datatyper.

Contents

Chapter 1

Introduction

Modern applications typically handle some form of persistent state that is stored in a database. The most commonly used versions are relational databases [1] that enforce structural consistency through schemas. As an application evolves, there is eventually a need to alter its database schema through the use of migrations, i.e. instruction sets that modify the enforced data structure.

Practices such as continuous integration and continuous deployments have accelerated software development through build automation of tasks such as compiling, testing, and deploying code. Logically, testing takes place in a testing environment that should mimic the production environment. Conventional unit tests typically use pre-defined, faked, or otherwise randomized data for testing purposes [2]. Such data is generally authored for a specific software version and regularly altered without consideration for compatibility and with testing databases constantly being reset and migrated from scratch [3]. This poses a problem as migrations that are applied to databases in the testing environment will not deal with data that may have been corrupted or formatted in a different way than data generated in newer software versions. When performing the same migrations in the production environment, the data differences could mean that the migration cannot be correctly applied to the database. In addition to this, manual database administration could have altered the database schema in an undocumented way, i.e. different constraints or indices. When an incompatible migration is executed, the process may fail resulting in a state that neither the new nor old software version can interpret [4]. When this occurs, previous backups must be restored, but the restoration processes are slow, causing service downtime, and could result in data loss [3].

For applications dealing with sensitive data, it is of paramount importance that data from a production environment does not leave it. This raises an issue when testing code as there may not always be representative testing data. Thus, testing database schema migration for compatibility with data is vital, although problematic.

* * *

This work was carried out together with the Swedish Police Authority which provided technical expertise and virtual environments where experiments could be performed and their results collected.

1.1 Objective

The objective is to provide a means of testing the compatibility of database schema migrations in such a way that sensitive production data cannot be leaked and stricter access policies can be employed. This means that testing should not be done in a clone of the production environment. Moreover, the method of testing migrations should be accessible and promote a wide adoption by developers and database administrators.

Backup restorations and rollbacks should not be the preferred way of handling migrations that may fail. The time required for each software release should not be negatively impacted either and the reliability of each release should be increased. Finally, the entire process should be automated.

1.2 Requirements

The main requirement is that database schema migration compatibility must be easily verifiable without requiring user access to production data or leaking it from the production environment. All incompatible migrations should be stopped prior to performing a migration that could lead to a broken state in which the database cannot be used by the application to which it belongs.

The migration validation process should be easily incorporated into a modern software workflow of using automated pipelines for continuous integration in order to promote continuous deployment without the risk of performing failing migrations and having to rely on time and storage consuming backup restorations.

The following set of requirements is a baseline for what is required for a solution to the problem:

Autonomous

All parts of the process involving database changes must be automatable if no other specific requirement prevents it.

Traceable

All database changes in production can be traced back to someone who can be accountable for the change. Specifically, individual changes in database, tables or fields should be traceable back to the individual developer who authored the change.

Changes should also be traceable in regards to deployed version and the individual that authorized the deployment if this step is not completely automated in which case some other accountability principle may be enforced.

This requirement is more important than automation if accountability cannot be ensured through a fully automated chain of events.

Reliable

Database errors are caught before invalid database migrations are performed.

Interoperable

Database engine and version can change over time without affecting the build pipeline.

Preservable

Data integrity is preserved when performing database schema migrations.

Scalable

The solution can be applied to small and large databases alike.

* * *

The requirements are prioritized in the following order: reliability, preservability, automation, traceability, interoperability, and scalability.

1.3 Research Question

To what extent can relational database schema migrations be tested without data by using pre-conditional assertions about a state prior to migrating, or schemas from production databases, when considering the level of automation, traceability, reliability, interoperability, preservability, and scalability?

1.4 Scope and Constraints

Database schemas can be changed in different ways, such as through a command line interface, or through custom SQL (Structured Query Language) [5] instructions. Neither the operations that are performed through a command line tool nor those entered through the use of SQL are considered in this work as they can perform the same schema change in countless ways and thereby massively increasing the number of cases to consider. All changes must instead be performed through a well-defined instruction format that is specific to changing database schemas and which the same change cannot be performed in multiple ways. Extensible Markup Language (XML) [6] is used to achieve this. This constraint is added in order to reduce the number of input instruction sets that can perform the same change.

The scope is further reduced by not testing the internal functionality, such as version tagging or maintenance operations, of tools that can help with performing migrations. Instead, all assertions are made for the application specific database schema. Moreover, operations that do not alter the application specific database schema are considered out of scope.

This work is relevant for developers and users alike. By allowing production data to be isolated, while enabling automated testing of database schema migrations, testing coverage increases, bugs are reduced, and fewer administrators need to have access to sensitive production data.

It is time consuming for database administrator to restore backups when migrations fail. By increasing the migration reliability, administrators can focus more on infrastructure maintenance rather than being on constant stand-by for when things go wrong.

Chapter 2

Background

This chapter provides the foundations required to understand the problem, i.e. preventing database schema migrations from failing on a production database, and exposes relevant approaches to address it.

2.1 Migration Automation of Relational Databases

In 1970, Codd published the foundation of relational databases in an effort to "provide shared access to large banks of formatted data" in such a way that applications would "remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed". He reduced the coupling between the representation of application data and persisted data through a model based on finitary relations in the realm of mathematics [7].

Additionally, Codd proposed a model in which data is represented as sets $S_1, S_2, \ldots, S_n$ that may be indistinct, a relation R can be defined as a subset of the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$ with five properties:

- Each row [of data] represents an n-tuple of R.
- The ordering of rows is immaterial.
- All rows are distinct.
- The ordering of columns is significant – it corresponds to the ordering $S_1, S_2, \ldots, S_n$ of the domains on which R is defined.

- The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

It should be noted that only a few alterations could be made to the internal representation in Codd's model without affecting the logic in some applications. This means that application logic must also be tested whenever its database's schema is altered.

* * *

Databases with structured data are organized in two parts: a schema and an instance. While the instance varies over time, using the formal schema to store data, the schema itself is time independent and makes up the database's structure with formal semantics [8]. The schema is a "finite set of rules" reflecting the "structural relationship among the data" [9]. Different database management systems handle schemas and their formatting differently [10]. Available rule sets can also differ between systems. This means that data coming from one database management system is not necessarily compatible with another.

As an application evolves there may be a need to alter the database's integrity constraints – its schema. Databases are refactored using migrations that often consist of one or more scripts of SQL instructions. All scripts that are required for performing a migration should be placed under version control in order to be able to deploy previous software versions as data representation may not be the same for each release, meaning that logic will be incompatible with other representations [11].

Some migrations, i.e. adding uniqueness constraints, changing data types, and restructuring tables during zero-downtime deployments (see section 2.5.2), handle data records and not only the formal rules. Thus, a migration may perform operations that change both the database instance, i.e. the database's data records, and schema [11]. Other changes, i.e. inserting, updating, or dropping data, may specifically target data records without affecting the schema in some cases. This means that migrations containing such changes cannot be tested without access to representative data, and such data is hard to obtain.

In 1982, Shneiderman and Thomas proposed that small atomic operations should be used to change databases' structure in a way that is consistent and to some extent reversible [12]. Later work by Roddick in 1993 proposed taxonomy for schema evolution through temporal rollback techniques [13]. The book *Refactoring Databases: Evolutionary Database Design (paperback)* covers the database refactoring process and states that migrations should be verified before being deployed. There is, however, no proposition on how to do so in an automated fashion that achieves the goals laid out in section 1.2. A manual method for writing tests for databases was presented in 2008. The method used a coverage metric to guide testers to improved tests compared to previously [14]. In the same year, it was, however, shown that relational databases could be automatically normalized and have their primary keys generated. This was done using dependency graph diagrams and dependency matrices to indicate dependencies. An automated process could minimize data redundancy by building directed graphs of data dependencies. Primary keys were identified in the process [15]. This indicates that database automation has been considered even though manual methods were developed.

Migrations can be handled with some automation, and without requiring the termination of running services. A viable method of providing zero-downtime database schema migrations was presented by Jong, Deursen, and Cleve in 2017. Mixed-state between two simultaneous schema versions was solved by introducing ghost tables that provided a transparent mixed state. Forward and backward triggers on the ghost tables and data tables were used to maintain compatibility, and can be removed when an update has been completed. Non-blocking migrations could be used to achieve zero-downtime, but performance was negatively affected by the triggers [16]. This was also presented along with a tool, QuantumDB, in 2015 [17] which further automated migrations at the cost of using another database management system.

Migration automation does, therefore exist to some extent, but currently offers only a very limited static analysis of migration syntax. Thus, the reliability requirement (see section 1.2) is not fulfilled.

2.2 Assertions

In order to increase the reliability of performing migrations, they must be tested. The result of performing migrations can be tested implicitly by migrating from an empty database state and then testing application logic. It is, however, more important to test that migrations can be performed without failing due to incompatibilities between migration and database state.

Testing stems from assertions, which are statements regarding a software system's intended behavior. Assertions can be used as a verification technique in which statements are converted to Boolean questions with two possible answers: true and false. The question is asked with the expected answer being true. If the answer comes back as false, the assertion's statement does not hold. When this happens, the software system's behavior is considered incorrect [18].

In 1949, Alan Turing [19] stated:

> How can one check a large routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.

Following Turing's reasoning, it is possible to verify the entire behavior of a software system by verifying its smaller components using assertions [19].

Turing's work was later expanded by Tony Hoare in 1969 when he presented the Hoare triple $\{P\}C\{Q\}$ [20].

The Hoare triple is interpreted as "if the assertion P is true before initiation of a [command] C, then the assertion Q will be true on its completion" using the modern notation [20].

Standard Hoare logic and the Hoare triple cannot be used to verify software that does not terminate. If C runs forever, there is nothing after it. Thus, the post-condition Q cannot be asserted [20].

Post-conditions are widely used to test software by making assertions about a method's result against an expected result when the input in known.

* * *

Database changes are mostly deterministic, in execution and result, if the database state prior to performing the changes is well-known. Assertions can be used to verify that the database state is such that a specific proposed change can be applied.

2.3 Conventional Testing

Conventional software testing is based on assertions and typically revolves around testing an application's functionality from a fixed perspective by comparing results when using known inputs with expected results that mimic the intended behavior. Application logic is generally tested with a configuration that matches that of the production system as much as possible.

Models and methods are often mocked or stubbed in order to test things in increased isolation. When increasing automation, every single code commit should be tested [21, 22]. Databases are often overlooked and only implicitly tested through application logic after having migrated the database from scratch to its most recent state and populating it with fixed, or randomly generated, data.

Conventional testing is commonly centered around white box testing, i.e. tests that are performed against a known code structure [23], and black box testing, i.e. tests that are performed without knowing details about the code that is tested [23].

Unit Tests

Unit tests are typically small tests for white box testing isolated units of code, i.e. only a single method or class, using assertions [18, 21]. The unit test coverage is not always representative of the code functionality as it is easy to write tests that achieve complete code coverage while missing certain aspects of the software being tested [24].

Integration Tests

Integration tests are usually larger in scope than unit tests. They generally provide white box, or black box, testing for a higher-level code structure consisting of more than a single method or class [21]. The test coverage of integration integration tests is more representative of the code functionality than unit test coverage. If all interactions have complete coverage, there should seldom be errors due to faulty logic [24, 19].

Smoke Tests

Smoke tests can be seen as minimal "sanity checks". They are used to ensure that the software is working as expected before it is released [25]. A smoke test can for instance check if the software can be started or if it can be accessed

[25]. For a web service, this would be equivalent with accessing the root path “/”. The coverage is poor, but verifies that the deployment process works.

If software performs all database migrations from scratch and use other testing techniques for verifying the application logic, it can be considered a minimal smoke test for its database schema migrations. This does, however, only prevent malformed migrations from being executed on the production environment.

Test Chaining

The software testing process can be automated in build pipelines (see section 2.5.1). Multiple test types, i.e. unit tests and integration tests, are often run in a chain with non-parallel tests being performed in rising order of complexity, and execution time. If one of the tests in the chain fail, the entire chain fails. It is desirable to have tests fail fast if there is a problem anywhere in the testing chain. This reduces the overall verification time [26]. A typical build pipeline with chained tests is illustrated in figure 2.1.

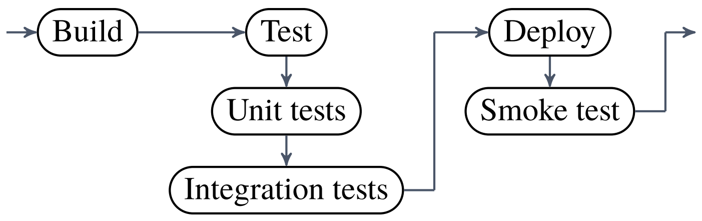

Figure 2.1: A simple build pipeline with continuous delivery and chained tests.

2.4 Database Testing

Conventional testing has failed to provide a means of preventing incompatible migrations from being performed on the production database. There are, however, a few techniques for performing some tests on databases and their schema migrations.

Query Validation

SQL instructions can be described using formal semantics that can be used to validate both individual instructions or migrations that are comprised of multiple SQL instructions [27]. Query validation through static analysis cannot be used detect compatibility issues with database migrations as the set of formal semantics used by the production database is unknown during validation.

ALGV

An alternative to query validation is ALGV (Automated Liquibase Generator and Validator), which is an automated Liquibase (see section 2.6) migration script generator with static input validation that was created by Manik Jain (2015). ALGV also abstracts away some aspects of handling database schema migrations by only requiring developers to enter "business logic" as stated by Jain [28].

ALGV can be used to ensure that the data format is valid, but it cannot be used to prevent incompatible migrations from being deployed. Thus, ALGV cannot be used to fulfill the requirements of section 1.2.

Schema Comparisons

A more complete method of testing migrations was used to test the functionality of Liquibase by Dziadosz et al. (2017) who performed a migration and compared its resulting schema with the production database's schema which he attempted to mimic [29].

Although schema comparisons are an improvement to ALGV and statical analysis, it misses migrations that either touch or depend on data records. The method can, however, be expanded by reducing the size of each comparison by using the reasoning of Turing [19] to form assertions as described in section 2.2.

* * *

Previous research has not presented a solution for viably testing database schema migrations without running migrations against a clone of the production environment or by performing manual tests for each migration. Static analysis of migrations can be used to some extent, but the problem of incompatible migrations being executed on a production database remains. An improved testing method requires more knowledge about the production database, which poses a problem when data is not allowed to leave the production environment.

The assertion method (see section 3.3) for validating migrations uses many smaller comparisons as per the schema comparisons in section 2.4 but moves comparisons to the production environment rather than the testing environment. This ensures that no production data is leaked in the process. Moreover, the schema method (see section 3.4) for validating migrations performs the comparisons by performing migrations using the production database's schema as a basis to make sure that migrations are compatible with the state, and comparing the resulting schema implicitly through the conventional testing (see section 2.3) of application logic.

2.5 Continuous Integration and Continuous Deployment

One of the requirements from section 1.2 states that the process of handling database changes should be automatable. By using an autonomous process, software releases should also become more reliable [30]. One way of automating software development is through the use of continuous integration, which was introduced by Fowler and Foemmel in the year 2000 as an extension of the ideas followed by the XP, "Extreme Programming", community in that software build processes should be automated and that builds should be run as often as possible. Furthermore, they presented a simple software build pipeline that starts when the codebase receives an update, has automatically built software, and automated tests for a deliverable application [22].

Fowler and Foemmel expanded the concept of continuous integration in 2006 when they stated that "every commit should build the mainline on an integration machine". They also claimed that the build process should take at most ten minutes to complete in order for the testing to provide rapid feedback. The concept was also expanded to cover software deployments in continuous deployment [22].

Continuous delivery is the continuation of continuous integration through the means of an automated delivery process. An even more automated process is that of continuous deployment in which deliverables are automatically deployed to production following a successful software build and delivery process.

The general goal with continuous integration and continuous deployment is to reduce time to delivery by shortening development cycles, increase software quality through testing, and that all of these subgoals should be done through means of automation [31]. It has been proven that companies adopting continuous integration, continuous delivery, and continuous deployment achieve these goals [31]. The rest of this section covers how such processes are often implemented with build pipelines (see section 2.5.1) stemming from test chaining (see section 2.3) and some techniques for delivering and deploying database migrations (see section 2.5.2).

2.5.1 Build Pipelines

Olsson, Alahyari, and Bosch (2012) wrote that “the transition towards continuous integration requires an automated test suite, a main branch to which code is continually delivered and modularized development” [26, p. 399]. The automation should start from the code on the main branch or from the result of a proposed change to it. A declarative build pipeline can be used to house and orchestrate everything that is needed to produce a new software version from some given source code configuration [22]. A build pipeline consists of individual steps, i.e. handling source code checkouts, fetching third party dependencies, producing a binary artifact, running tests, or mechanisms for delivering produced artifacts to a server, that are required for the build [22]. A simple build pipeline is illustrated in figure 2.2.

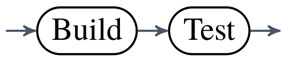

Figure 2.2: A simple build pipeline with continuous integration.

Some steps may also be run in parallel if this is supported by the software system. This vastly increases the scalability aspects and results in quicker feedback [22]. A simple build pipeline that utilizes parallelization is shown in figure 2.3.

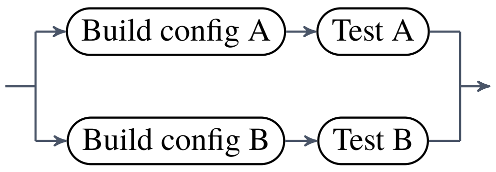

Figure 2.3: A simple build pipeline with continuous integration that builds, and tests, two different configurations in parallel.

In order to achieve continuous deployment, the automated build pipeline must include a step that deploys the built software into production. This step differs from product to product, but the main goal is to make the new version available.

Olsson, Alahyari, and Bosch (2012) noted that software development practices change over time, and that a there is a barrier between continuous integration and continuous delivery due to this. The study concludes that the following challenges exist [26]:

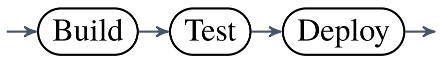

Figure 2.4: A simple build pipeline with continuous delivery.

1. Local software configurations cause inconsistencies which make the code hard to test reliably.
2. To produce builds for every code commit, the verification process needs to be quick.
3. Work processes must be transparent. Otherwise, code integrations become problematic.

Chen (2015) stated that "a robust, out-of-the-box, comprehensive, yet highly customizable solution for [continuous delivery] doesn't exist yet" [32, p. 54]. This is mainly because solutions for continuous integration, continuous delivery, and continuous deployment must generalize well – otherwise, new solutions have to be created for each individual software project [26, 32].

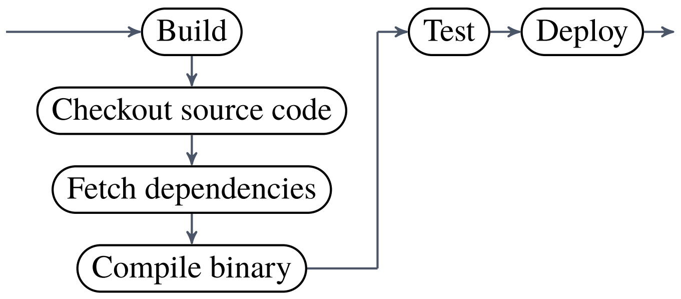

Figure 2.5: The build stage typically consists of smaller tasks.

Pipeline stages can also consist of smaller sub-tasks. An example of a more detailed pipeline can be seen in figure 2.5. The build stage may typically involve fetching the source code repository, download external dependencies, and compile some form of binary or otherwise readable file.

2.5.2 Delivery and Deployment

The ultimate build pipeline should automate the entire process starting with source code and ending with software running in a production environment. This is known as continuous deployment, and the final stage of the normal continuous delivery pipeline should result in the new software version being deployed, or delivered somewhere [22]. This section handles that process for database migrations.

Some services have high demands on availability. If the service is not allowed to be down for even a few seconds, they fall into a category of services with zero-downtime [16].

Web services that require zero-downtime utilize load balancers and multiple servers to redirect traffic to an available server while individual servers are updated in sequence using a method known as *Rolling Upgrade* [33, 16].

Rerouting traffic with load balancers is not always feasible when a service's database structure is changed. Many schema changes can lock certain database tables during the time that the change is occurring. These changes take varying time to carry out depending on the table size. For a large database table, this can take hours [16].

During the database schema change, the web service can fail due to issues with querying certain tables. This violates the requirement for zero-downtime, and thus software updates that change the database structure are considerably harder than what can be handled through routing traffic to servers with different versions of application logic [16].

Another issue is that the database structure could be different between the two concurrent versions. Both software versions should be able to independently manage the data without loss of service. This is referred to as being in a mixed-state [16].

One way to handle database schema changes is to place two different server pools behind a load balancer and use a method known as *Big-Flip*. Each pool should contain everything that is needed in order to run the service [33, 16]. The databases are synced between the two pools and schema changes can be carried out on the database in the idle pool [33, 16].

One downside of the Big-Flip method is that all costs for running the service are essentially doubled. Furthermore, the application exists in a mixed-state during the transition from one version to another [33, 16].

The server pools for Big-Flip can also be active at the same time when no update is being performed [33]. This scenario is, however, rather unlikely as the goal of continuous integration and continuous delivery is to constantly push out new updates [22]. This means that the system needs to be functional when only half of the maximum capacity is available [33].

Blue-Green Deployments can be used for managing database changes. It is essentially a Big-Flip approach that just handles the database. With it, two database instances are placed behind a load balancer. The two instances can be updated independently while being idle. Data can be synchronized between the two instances. With this method, the application may not have to deal with mixed-state as the database synchronization could handle such issues [3].

Another database specific method, *Expand-Contract*, can also be used. This method modifies the database schema using non-blocking instructions while staying compatible with older and newer software versions [3].

* * *

Previous work shows that the migration deployment process can be automated without causing service downtime. This means that the last stages of a complete build pipeline (see section 2.5.1) exist whereas the earlier stages, belonging to the realm of continuous integration, do not. By providing database-specific testing, the reliability could be increased to a level at which automated deployments become viable, thus promoting rapid releases [22].

2.6 Liquibase

If no tool is used to perform database schema migrations, SQL instructions are executed directly by the database management system. This process is illustrated in figure 2.6 and results in a database that is hard to version and a process that entirely depends on compatibility between SQL instructions and database engine. Furthermore, it is difficult to trace what changes are made to the production database's schema, when they were made, and by whom. This means that neither the traceability nor interoperability requirements from section 1.2 are met.

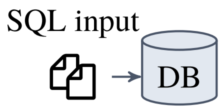

Figure 2.6: Migration process without a migration tool such as Liquibase.

Oertly and Schiller presented a basic approach for handling evolving databases by modeling migrations as lists of change entries that can be processed using one pointer for reading data, and one pointer for writing data [34] while change entries are available:

- [Move] the read pointer to [point to] the next change entry.
- Read the change entry.
- Analyze the entry and, if necessary:
- React to the change entry in an appropriate way.

The approach that Oertly and Schiller specified could provide data consistency throughout the migration process. It was, however, limited in terms of database interoperability as their solution was specific to the “Presto” environment for “evolutionary database and application design” [34]. An open-source project, Liquibase [35], was started in 2006 and worked in a similar way. The project focused on allowing database-independent schema migrations that could be source controlled. It has since been developed to support a multitude of migrations and input formats [29, 35].

13 different databases are supported by Liquibase version 3.6.3 without the use of any plugins [36]. These databases are listed in table 2.1.

MySQL	PostgreSQL
Oracle	Sql Server
Sybase_Enterprise	Sybase_Anywhere
DB2	Apache Derby
HSQL	H2
Informix	Firebird
SQLite	

Table 2.1: Liquibase-supported databases.
Liquibase version 3.6.3 supports 13 databases without the use of any plugins [36].

The changes that should be made to a database schema are stored in changelogs with one logical root file for each database. The root changelog contains an ordered collection of changesets which in turn contain the actual instructions for what changes should be made. Changelogs can also include other files in order to keep the file sizes manageable [37]. The migration process with Liquibase is shown in figure 2.7.

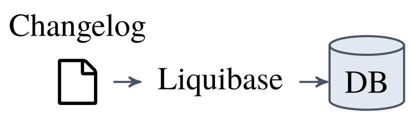

Figure 2.7: Migration process with Liquibase.

Two metadata tables, DATABASECHANGELOG [38] and DATABASE-CHANGELOGLOCK [39], are created by Liquibase in order to ensure that each changeset is only migrated once and that only a single Liquibase instance can access the database at any given time, thus preventing concurrency issues. Each changeset should also be given a unique combination of author and id. This is done by the author, and is stored in Liquibase's DATABASECHANGE-LOG metadata table subsequent to executing a changeset [38]. Changesets can also be configured to always execute when the changelog is handled [40].

Transaction locks are used around changesets to ensure atomicity. If a single change fails, the entire changeset fails, and the database is left unaltered. This could still pose a problem if multiple changesets should be executed in order to update a database before a new software release. Furthermore, database engines limit the size of transactions meaning that they cannot always be used with large databases [41, 42, 43]. A Liquibase changelog is thus not atomic [37, 40].

Liquibase has support for executing pre-conditions before performing any changes in order to ensure that the change is compatible with the current database state. Pre-conditions can be placed on changelogs, in which case they are checked before executing anything. Pre-conditions can also be placed on individual changesets, where they can prevent incompatible changes to be attempted and thus also time inefficient rollbacks [44]. Table 2.2 lists the available pre-conditions in Liquibase 3.6.3.

AND/OR/NOT Boolean logic.	Database type.
Running as user.	Changeset has been executed.
Column exists.	Table exists.
View exists.	Foreign key constraint exists.
Index exists.	Sequence exists.
Primary key exists.	Custom SQL or Java class.

Table 2.2: Liquibase-supported pre-conditions.
Version 3.6.3 of Liquibase allows testing to be performed through assertions (see section 2.2). Pre-conditions can also be formulated using custom SQL instructions.

The migration process with Liquibase has been verified multiple times, thereby increasing its reliability. Dziadosz et al. (2017) showed that an Oracle database could be migrated from scratch by Liquibase changelogs in such a way that the resulting database was identical to the reference [29]. Further verification came in 2018 when Mueller and Müller verified that Liquibase could be used to handle database versioning for CERN. They also concluded that all database changes must be synchronized between team members and that the same procedure should be used for all migrations. Furthermore, they declared that when a schema versioning tool is used, everyone in the development team must use it [45].

2.7 Research Design

Previous research has not presented an answer to the research question, but it offers some building blocks to build with. The primary focus of this work is to increase the reliability of database schema migrations and prevent incompatible migrations from failing in a production environment thereby causing a broken database state. In order to increase the interoperability and provide an easily interpretable migration format, Liquibase [35] is used in this research. The Liquibase formatted changelogs also lend themselves well to provide traceability through schema versioning and migration files that can also be placed under source code control through Git [46]. A default method using Liquibase should thus be able to provide a baseline result for the reliability of the migration process [29].

Many different testing techniques stem from small assertions [19] (see section 2.2) which Liquibase provides as pre-conditions [44] that are comparable to the pre-conditions of the Hoare triple [20]. Pre-conditions could be used to prevent incompatible changes from being attempted, but the current process is manual and therefore fail the automation requirements of a modern workflow (see section 2.5). There should be a way to automate pre-condition generation due to the well-structured changelog format [37] that is used by Liquibase. An assertion method of validating Liquibase changelogs against a database state should use the Liquibase format to produce pre-conditions that are executed on the production database without leaking data.

Although the assertion method could stop incompatible migrations from breaking the production database, it does so at a very late stage of traditional software build pipelines (see section 2.5.1). Another method that is based on schema comparisons [29] (see section 2.4) should export the production database's schema, insert it into a database in a testing environment, and execute Liquibase changelogs on it. A schema method would not be able to validate changes involving data records, but it may be able to provide an acceptable reliability for validating compatibility between migrations and database states.

A comparison between the assertion method and the schema method using the default Liquibase method as a reference would serve to answer the research question and lead to further research that could be used to eventually fulfill the objective described in section 1.1.

Chapter 3

Methodology

A new component, a *validator*, was introduced in the database schema migration process in order to validate migrations. The validator resides in a pipeline test stage that is designed to detect, and reject, incompatible migrations before changes are made to the production database.

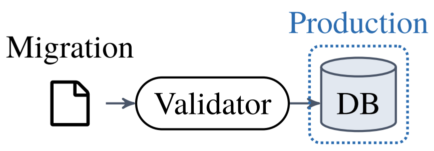

Figure 3.1: A validator component is inserted between a Liquibase changelog in XML format (labeled “Migration” in the figure) containing a set of changes [37] that should be performed on the production database’s schema and database (herein denoted as “DB”) in order to verify the migration’s validity. The production database holds a persistent state of an application’s data. The validator is responsible for preventing the execution of migrations that are not compatible with the database’s state. There are multiple ways of implementing a validator component.

When in use, all migrations must pass through a validator before being executed. This is illustrated in figure 3.1.

Two validation methods were considered for this project. The first method, the assertion method, uses pre-condition assertions to ask true/false questions about the database’s state for which the answer should always be true. An enhanced migration is acquired by injecting pre-conditions into a copy of the original Liquibase migration. If the answer is false, the migration is not deemed compatible with the database and the migration is aborted. Pre-conditions are automatically generated by a separate entity herein called a *generator* (see section 3.3), which uses the individual Liquibase changes [47]. The second method, the schema method, performs a migration against a copy of the production database's schema. If there are any issues while performing the migration on an isolated copy of the schema, the migration is rejected as invalid. The two proposed methods are shown in figure 3.2.

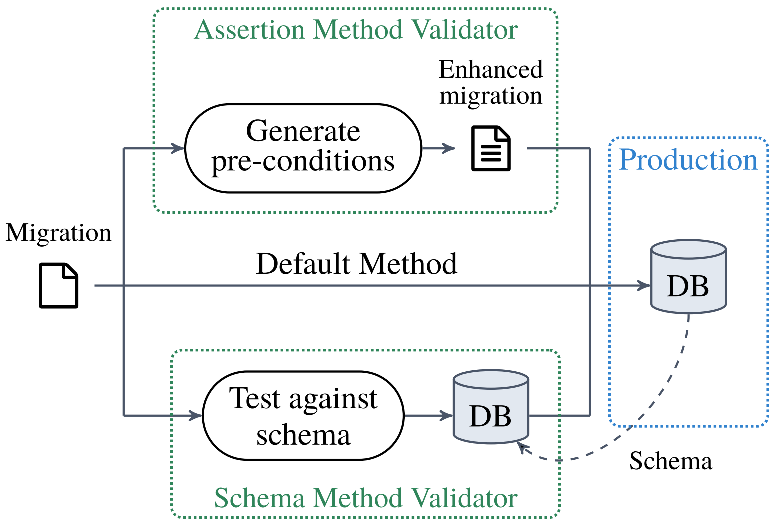

Figure 3.2: Overview of the two proposed, and tested, validation methods. Two validator implementations, one using the assertion method from section 3.3 and one using the schema method from section 3.4 are highlighted. The default method of using Liquibase without a validator component is also included in the overview.

3.1 Changelog Handling

In order to provide consistency across implementations and thereby increase the interoperability by reducing the training that is required to switch between projects and validation methods, all migrations should be handled in the same way. All schema changes should originate from the same root changelog file. This file is given the name *changelog-master.xml*. The root changelog should be placed in a resource folder such as *src/main/resources/liquibase* within its application's source code. This folder should be be placed under source control [48].

In order to maintain a good structure for migrations, the root changelog may not contain any changesets. Instead, it may only contain "include" instructions to import other changelogs as shown in listing 10.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/
      dbchangelog http://www.liquibase.org/xml/ns/
      dbchangelog/dbchangelog-3.6.xsd">
  <include file="./changelog-update-1.0.0.xml"/>
  <include file="./changelog-update-1.0.1.xml"/>
  <include file="./changelog-update-2.0.0.xml"/>
</databaseChangeLog>
```

Listing 3.1: Example of a *changelog-master.xml* file for the 3.6 minor release of Liquibase.

Every time that the database should be changed, a new changelog should be added to the *src/main/resources/liquibase* folder. Added changelogs should follow the same naming convention of using *changelog-update-VERSION.xml*. This makes it easy to get the change history for software versions. The application's version is also used when versioning the changelogs. The version numbering should typically follow that of semantic versioning [49]. A complete example of the file structure required for a few different versions is illustrated in figure 3.3.

Changelogs belonging to already released software versions may not be altered as this would prevent changes from being traceable, thus violating the traceability requirement from section 1.2. Furthermore, only the main changelog should contain "include" instructions in order to keep the folder structure clean. This also makes the structure easily greppable.

The SQL format should not be used for changelogs as it has a tighter cou-

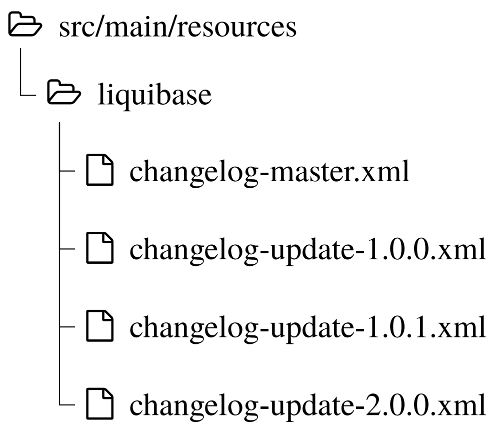
src/main/resources
liquibase
changelog-master.xml
changelog-update-1.0.0.xml
changelog-update-1.0.1.xml
changelog-update-2.0.0.xml

Figure 3.3: Changelog file structure. Adopted from the Liquibase documenation, but modified to align file names, and to provide better support for sorting [48].

pling with the database engine due to variations in allowed SQL syntax. Although the XML, YAML, and JSON formats offer the same functionality, the XML format was chosen for this work as it well supported on all platforms and programming languages. The same format should be used for all changelogs for the same database.

3.2 Default Method

The default method uses a typical approach for running schema migrations with Liquibase, which provides a database agnostic and structured way of organizing schema changes as described in section 2.6. No validator is used.

1. An XML-formatted Liquibase 3.6.3 changelog [35] is executed directly on the MySQL 8.0.11c database [50] server in production through a Connector/J 8.0.13 [51] driver that facilitates communication between the Liquibase software and the MySQL software.

   The migration is visualized in figure 3.4 where Liquibase is used to perform a migration "Migration" directly against a database "DB" in the production environment.

   The following command is used to perform the migration:
   `liquibase --changeLogFile=changelog.xml migrate`.

2. During the execution, Liquibase may attempt to change the schema in a way that is not allowed by the MySQL database. If this happens, Liquibase will detect it through an error message sent back through the Connector/J driver. This error will describe the illegal operation.

   If such an error is received, the operation is halted and the migration has failed. Otherwise, if no errors are received, a message will be sent back notifying that the changes have been carried out. This means that the migration has been successfully performed.

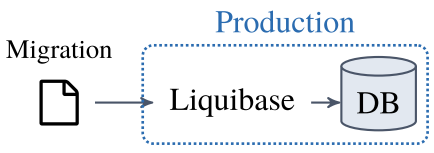

Figure 3.4: The default method. A migration, a Liquibase changelog in XML format, is performed the production database using Liquibase. The changelog consists of multiple changes that are made to the database's schema. No validator is used between the input migration and the production database.

3.3 Assertion Method

The first method, the assertion method, is based on the Hoare triple [20] and conventional testing through assertions [19]. If a command C is known and deterministic, its output should always be the same given consistent input following Hoare's reasoning. By adding pre-conditions P according to the Hoare triple $\{P\}C\{Q\}$, the result should also be consistent.

A validator formulates assertions about what the database state should, and should not be, prior to performing a certain migration. The assertions are executed as pre-conditions. A migration is only allowed to be performed if all pre-conditions pass.

The validator pairs Liquibase changes with a number of pre-conditions that are defined beforehand using a separate *generator* component.

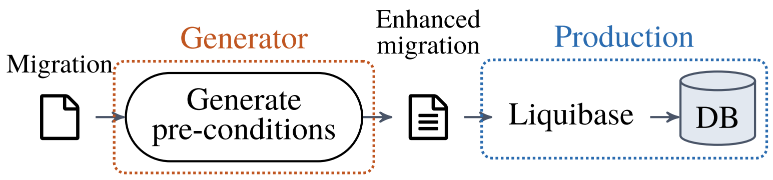

Figure 3.5: The assertion method. An inputted Liquibase XML changelog "Migration" is injected with Liquibase Pre-conditions [44] through a generator component and outputted as an enhanced migration in the same XML format. The enhanced migration is used by Liquibase to perform changes on the production database. Pre-conditions are true/false questions with an expected true answer. All pre-conditions are checked prior to making changes. If at least one of the pre-conditions fail by getting a false answer, the migration is aborted and no changes are made to the database.

3.3.1 Pre-condition Mappings

A mapping consists of a pairing between a single type of Liquibase change [47] and a number of assertions that should be true for the migration to be compatible with a database's state.

The purpose of using predetermined mappings is to provide a deterministic generation of pre-conditions and to promote code reuse. Empty assertions are provided as templates that are filled out by a pre-condition generator. All assertions ask true/false questions about the database's state with the expectation that all answers come back as true.

Mappings are resolved from a Liquibase change [47] by the generator using the change's type.

The following mappings were used to generate pre-conditions for each type of change. The table is sorted alphabetically.

Change	**Pre-condition Assertions**
Add Auto Increment	The column exists. The column is indexed.
Add Column	The table exists. The column does not exist
Add Default Value	The column exists.
Add Foreign Key Constraint	The base column exists. The referenced column exists.
Add Not Null Constraint	The column exists. The column is not nullable. There are no null values exist within the column (only if no default value is given).
Add Primary key	The columns exist. The primary key does not exist. The table does not have a primary key.
Add Unique Constraint	The columns exist. The column does not have a unique constraint. The column does not have a constraint with the same name. The columns are unique before adding the constraint.

Change	**Pre-condition Assertions**
Create Index	The columns exist. The table does not have an index with the same name.
Create Table	The table does not exist.
Create View	The view does not exist (if there is no declaration to replace it). The SQL instructions in the where condition is valid. The tables in the where condition exist. The columns in the where condition exist.
Delete Values	The custom SQL instructions are valid. The table exists. The columns exist.
Drop Column	The table exists. The column exists.
Drop Default Value	The column exists. The column has a default value.
Drop Foreign Key Constraint	The foreign key exists.
Drop Index	The index exists.
Drop Not Null Constraint	The column exists. The column is not nullable.
Drop Primary Key	The primary key exists. The table has a primary key.
Drop Table	The table exists.
Drop Unique Constraint	The unique constraint exists.
Drop View	The view exists.
Insert Values	The columns exist. Uniqueness constraints will not be violated for single columns.
Modify Data Type	The columns exist.

Change	**Pre-condition Assertions**
Rename Column	The old column exist. The new column does not exist.
Rename Table	The old table exists. The new table does not exist.
Rename View	The old view exist. The new view does not exist.
Update Values	The custom SQL instructions are valid. The table to update exists. The tables in the where condition exist. The columns in the where condition exist.

Table 3.1: A table of mappings between Liquibase changes and pre-condition assertions. Changes are listed in alphabetical order.

3.3.2 Procedure

1. The Liquibase XML changelog is interpreted and broken down into uniquely identifiable Liquibase changesets [37, 40].

```xml
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
 xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.6.xsd">
  <changeSet id="1" author="peter">
    <createTable tableName="people">
      <column name="name" type="varchar(255)" />
    </createTable>
  </changeSet>
  <changeSet id="2" author="peter">
    <addColumn tableName="people">
      <column name="nickname" type="varchar(255)" />
    </addColumn>
  </changeSet>
</databaseChangeLog>
```

Listing 3.2: Example of a Liquibase changelog XML file containing two changesets.

   The changelog in listing 3.2 would be broken down into one changeset containing a change that creates table “people” and one changeset containing a change that adds a column “nickname” to the table “people”.

   The changesets are further reduced into a list of changes.

   The changeset with id “1” in listing 3.2 consists of a single change for creating the table “people” and the changeset with id “2” consists of a single change that adds a column to it.

2. The *generator* pairs each change with previously defined assertion templates as specified by the pre-condition mappings in section 3.3.1.

   An assertion template is an assertion without details from the change that it was paired with.

   The changeset with id “1” in listing 3.2 would be mapped to the pre-condition assertions found under “Create Table” in section 3.3.1. In this particular case, the only such assertion is that the table should not already exist in the database.

3. The assertion templates are filled in by the generator using information from the single change that they were each paired with. The SQL parser JSqlParser [52] is used to extract tables and columns for changes that partially use SQL input. The filled out assertion templates result in a set of assertions.

   Assertions are translated into Liquibase pre-conditions [44] and added to their corresponding change's parent changeset.

   The process of generating pre-conditions from a Liquibase changelog is illustrated in figure 3.6.

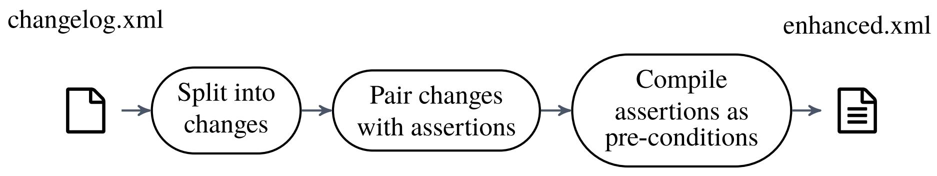
changelog.xml
Split into changes
Pair changes with assertions
Compile assertions as pre-conditions
enhanced.xml

Figure 3.6: The functionality of a pre-condition generator. A Liquibase changelog (changelog.xml) in XML format is injected with pre-conditions and outputted as a larger XML-formatted Liquibase changelog (enhanced.xml).

4. The changelog with the added pre-conditions is serialized back to the Liquibase XML format (enhanced.xml in figure 3.6). The reserialized changelog is archived.

   The changeset with id "1" in listing 3.2 results in the modified found in listing 3.3

```xml
<changeSet id="1" author="peter">
  <preConditions onError="HALT" onFail="HALT"
    onSqlOutput="IGNORE">
    <not>
      <tableExists tableName="people"/>
    </not>
  </preConditions>
  <createTable tableName="people">
    <column name="name" type="varchar(255)" />
  </createTable>
</changeSet>
```

Listing 3.3: A stub of changeset with id "1" in listing 3.2 containing pre-conditions generated using the mappings found in section 3.3.1. A pre-condition has been added that asserts the table "people" does not exist prior to

performing a change that creates the table “people”. This is done by wrapping the Liquibase pre-condition “tableExists” with a “not” element for inverting Boolean logic [44].

5. The modified Liquibase changelog (enhanced.xml in figure 3.6) is executed on the MySQL production database [50] through Connector/J [51].

6. Pre-conditions are executed before each change set in the changelog as the enhanced changelog is being handled by Liquibase [35]. If at least one of the pre-conditions fail, the entire migration process is aborted.

   Otherwise, the production database has been updated as intended by the initial Liquibase migration changelog. The migration is successful.

   The migration fails if the generator fails to provide adequate validation through pre-conditions for a requested change. In this case, errors are thrown during the migration process, but the database may have been left in a half-migrated state.

3.4 Schema Method

The second method, the schema method, revolves around pipeline integration through the use of an additional test environment that is separated from the production environment. This process is designed to fit into traditional build pipelines for continuous integration with incremental database changes [3, p. 327-331]. The schema method requires two databases of the same engine version: one database dedicated to production and one database for testing. The testing database does not contain any data records.

The production database's schema is extracted and injected into an empty database in an isolated testing environment. Liquibase changelogs are executed against the schema in order to validate them. This is shown in figure 3.7.

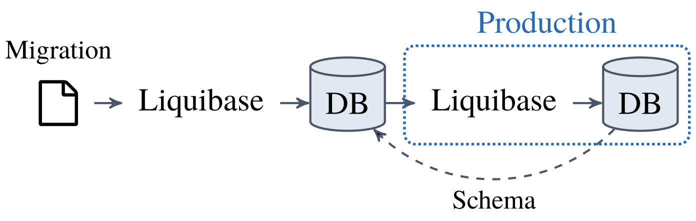

Figure 3.7: The schema method. The database schema is extracted from the production database, and used to test migrations. A Liquibase XML-formatted migration "Migration" is first executed on the empty schema in the database in the testing environment. If the migration is considered successful in the testing environment, the same migration is executed on the database in the production environment.

1. The database schema is extracted from the production MySQL database [50] and stored in a `schema.sql` file with SQL instructions for rebuilding the database schema from scratch.

   This is done with the following command:

```
#!/bin/sh
mysqldump -no-data DB --ignore-table={DB.
    DATABASECHANGELOG,DB.DATABASECHANGELOGLOCK} >
    schema.sql
mysqldump DB DATABASECHANGELOG DATABASECHANGELOGLOCK
     >> schema.sql
```

Listing 3.4: A script that can be used to extract the schema from a database that is handled by Liquibase. The tables that are used for Liquibase to handle changes are exported in their entirety in order to allow previously migrated

changes to be skipped. Production data is not stored in any of the Liquibase-specific tables. Modifications may be required in order to connect to the database using valid credentials.

2. The `schema.sql` file is moved from the production environment to the test environment.

3. The extracted production database schema is imported into an empty MySQL database [50] in the test environment.

   This is done with the following command:
   `mysql < schema.sql`.

4. The Liquibase changelog is executed on the MySQL test database [50] through a Connector/J [51] driver that facilitates communication between the Liquibase software and the MySQL software.

5. If an illegal operation is encountered by MySQL during the migration process, Liquibase will detect it through an error message from the Connector/J driver. If this happens, the migration is considered invalid and the process is aborted.

   Otherwise, the migration is considered valid by the schema method. The Liquibase changelog is allowed to be executed on the production database.

6. The Liquibase changelog is executed on the MySQL production database [50] through Connector/J [51] using the same method as when performing migrations in the test environment.

   If an illegal operation is encountered, the schema method has failed to find the incompatibility between the Liquibase changelog and the production database's state. The migration has therefore failed

   If no illegal operation is encountered, the changelog was indeed valid as determined by the schema method through the isolated execution against the production database's schema.

3.5 Evaluation

Migration validation methods were evaluated using a set of 108 unique test cases that were designed using the supported changes for Liquibase and MySQL. Valid migration-state pairs were first added as test cases before performing minor alterations to individual aspects of the database state that would make the migration incompatible with it. Every such changed variable became a separate test case. A test case consists of an SQL file depicting the production database state, and a Liquibase XML changelog to execute.

1. Create test cases for changes supported by Liquibase and MySQL:

    - Add auto increment
    - Add column
    - Add default value
    - Add foreign key constraint
    - Add not null constraint
    - Add primary key
    - Add unique constraint
    - Create index
    - Create table
    - Create view
    - Delete values
    - Drop column
    - Drop default value
    - Drop foreign key constraint
    - Drop index
    - Drop not null constraint
    - Drop primary key
    - Drop table
    - Drop unique constraint
    - Drop view
    - Insert values
    - Modify data type
    - Rename column
    - Rename table
    - Rename view
    - Update values

    Note that when creating a view, there is an option to replace existing views of the same name. This drastic change in behavior alters the validity of migrations. Thus, there should be test cases for both scenarios. The test cases that should be used for evaluation are listed in table 3.2.

2. Implement two validators:

    - One validator that uses the assertion method to generate pre-conditions.
    - One validator that performs a migration against an extracted schema as described in the schema method.

3. Execute all test cases using validator implementations of the assertion method and the schema method along with the default method. Classify each test case result for each implementation using the following criteria:

   Successful The migration was considered valid by the validator.
   No errors occurred while performing the migration on the production database.

   Aborted The migration was considered invalid by the validator.
   The migration was not performed on the production database.

   Failed The migration was considered valid by the validator.
   Errors occurred while performing the migration on the production database.

The test cases followed a structure that eliminated repetition. One Liquibase migration changelog in XML format was paired with multiple database states. The test cases for table creation are illustrated in figure 3.8. The three files make up two unique test cases: one with a valid database state because it is empty and one with a state that is invalid because the table already exists.

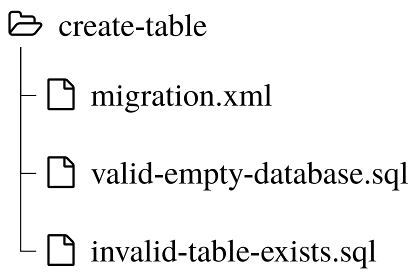

Figure 3.8: Test case structure for a migration that creates a table. The same migration is paired with two states, resulting in two separate test cases.

Test Cases

The following 108 test cases were used when evaluating the three implemented methods.

Liquibase Change	**Test Cases/Database States**
Create table	Empty database Table exists
Drop table	Table does not exist Table exists
Rename table	Empty database New table exists Old table does not exist
Add column	Column does not exist Column exists Table does not exist
Drop column	Column does not exist Column exists Table does not exist
Rename column	Column does not exist New column exists Old column missing Table does not exist
Create view	Invalid SQL
Create view with replacing	Column in view does not exist Column in where clause does not exist Columns exist Table in view does not exist View exists
Create view without replacing	Column in view does not exist Column in where clause does not exist Columns exist Table in view does not exist View exists
Drop view	View does not exist View exists

Liquibase Change	**Test Cases/Database States**
Rename view	New view exists Old view does not exist Old view exists, new view does not exist
Create index	Column does not exist Index does not exist Index exists Index exists in other table No indices exist
Drop index	Index does not exist Index exists Index exists in other table No indices exist
Add auto increment	Column does not exist Column exists with primary key Column exists without primary key Column is auto incrementing Column is of type “datetime” Column is of type “varchar”
Add primary key	Column does not exist Index exists but is not a primary key No primary key exists Primary key exists Primary key exists in other table
Drop primary key	Column does not exist Index exists but is not a primary key No primary key exists Primary key exists Primary key exists in other table
Add foreign key	Base column does not exist Referenced column does not exist Referenced column is not primary key Has unnamed constraint Invalid reference Named constraint exists Valid references

Liquibase Change	**Test Cases/Database States**
Drop foreign key	Base table does not exist Constraint does not exist Constraint exists Constraint is in another table
Add not null constraint with default value	Column does not exist Column exists with null data Column exists with null data and default value Column exists without null data Column is not nullable
Add not null constraint without default value	Column does not exist Column exists with null data Column exists with null data and default value Column exists without null data Column is not nullable
Drop not null constraint	Column does not exist Column exists Column is nullable
Add default value	Column does not exist Column exists Column exists and has default value Column exists with other data type
Drop default value	Column does not exist Column exists and has default value Column exists without default value
Delete values	Column exists without null data Column in where clause does not exist
Insert values	1-column unique constraint and duplicates 1-column unique constraint no duplicates 2-column unique constraint and duplicates 2-column unique constraint no duplicates Column does not exist
Modify data type	Column does not exist Compatible data Incompatible data

Liquibase Change	**Test Cases/Database States**
Update values	Column exists Column in where clause does not exist Updated column does not exist
Add unique constraint	Column exists with non unique data Column exists with unique data Column name does not exist Constraint exists
Drop unique constraint	Constraint does not exist Constraint exists

Table 3.2: A table of Liquibase changes and test cases for each change.

Chapter 4

Results

All results in this chapter were obtained using the methods described in chapter 3: the default method performs Liquibase migrations without any validation, the assertion method performs Liquibase migrations after inserting preconditions that verify that the database state matches expectations, and the schema method performs Liquibase migrations on a copy of the production database's schema to check for errors. Test results adhere to the evaluation classifications as defined in section 3.5.

The results are presented in logical groups after tested change with a table that lists test results for each test case. The logical groups are: tables, columns, views, indices, auto incrementation, primary keys, foreign keys, nullability, default values, handling individual values, and unique constraints. A summary, section 4.12, is also included at the end of the chapter. The table is preceded by a diagram that shows the numeric distribution of result classifications. A total of 108 unique test cases were used.

4.1 Tables

This section covers results for the seven test cases that are related to creating, dropping, and renaming tables.

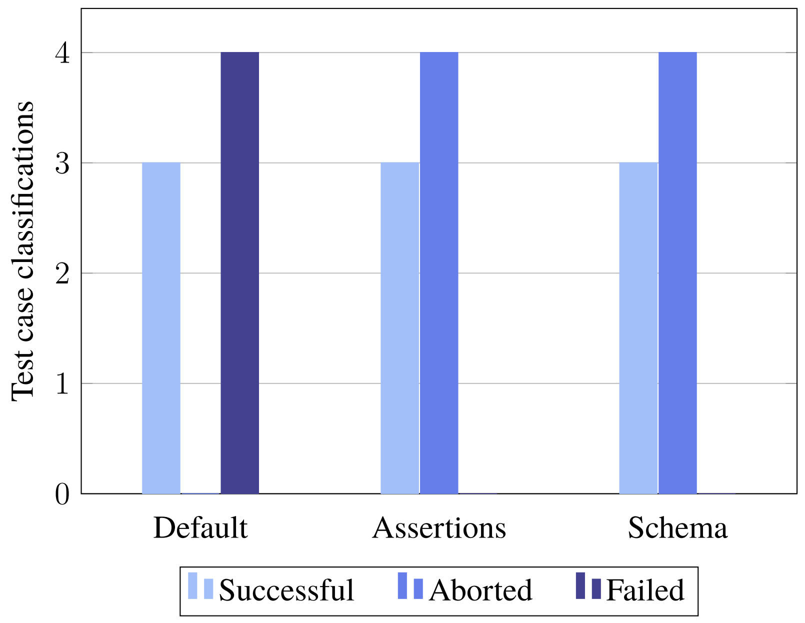

Figure 4.1: Table test case results. 7 test cases regarding the creation, deletion, and renaming of database tables.

Test Case	**Default**	**Assertions**	**Schema**
Create table: Empty database	Successful	Successful	Successful
Create table: Table exists	Failed	Aborted	Aborted
Drop table: Table does not exist	Failed	Aborted	Aborted
Drop table: Table exists	Successful	Successful	Successful
Rename table: Empty database	Successful	Successful	Successful
Rename table: New table exists	Failed	Aborted	Aborted
Rename table: Old table does not exist	Failed	Aborted	Aborted

Table 4.1: Detailed table of table test case results. The results are identical for the assertion method and the schema method.

4.2 Columns

This section covers results for the ten test cases that are related to adding, dropping, and renaming columns.

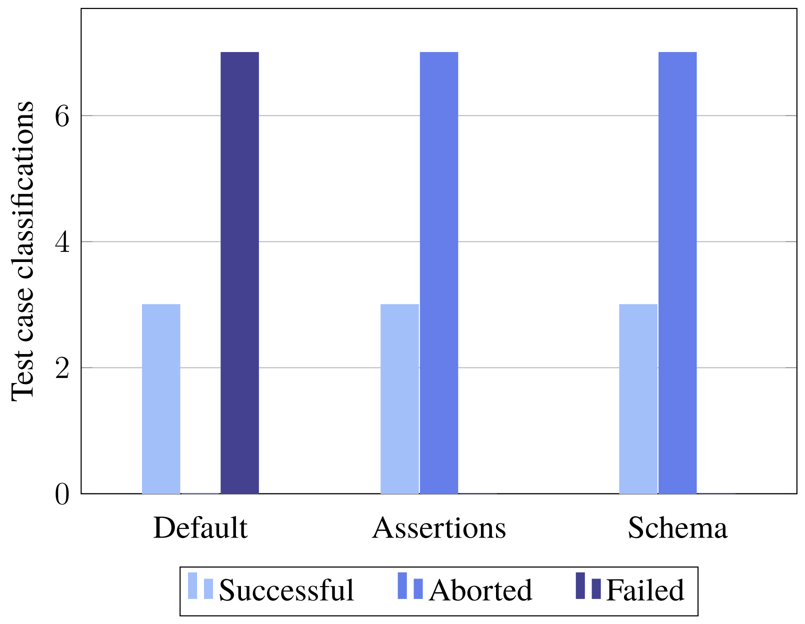

Figure 4.2: Column test case results. 10 test cases regarding the creation, deletion, and renaming of table columns.

Test Case	**Default**	**Assertions**	**Schema**
Add column: Column does not exist	Successful	Successful	Successful
Add column: Column exists	Failed	Aborted	Aborted
Add column: Table does not exist	Failed	Aborted	Aborted
Drop column: Column does not exist	Failed	Aborted	Aborted
Drop column: Column exists	Successful	Successful	Successful
Drop column: Table does not exist	Failed	Aborted	Aborted
Rename column: Column does not exist	Successful	Successful	Successful
Rename column: New column exists	Failed	Aborted	Aborted
Rename column: Old column missing	Failed	Aborted	Aborted
Rename column: Table does not exist	Failed	Aborted	Aborted

Table 4.2: Detailed table of column test case results. The results are identical for the assertion method and the schema method.

4.3 Views

This section covers results for the 16 test cases that are related to creating, dropping, and renaming views.

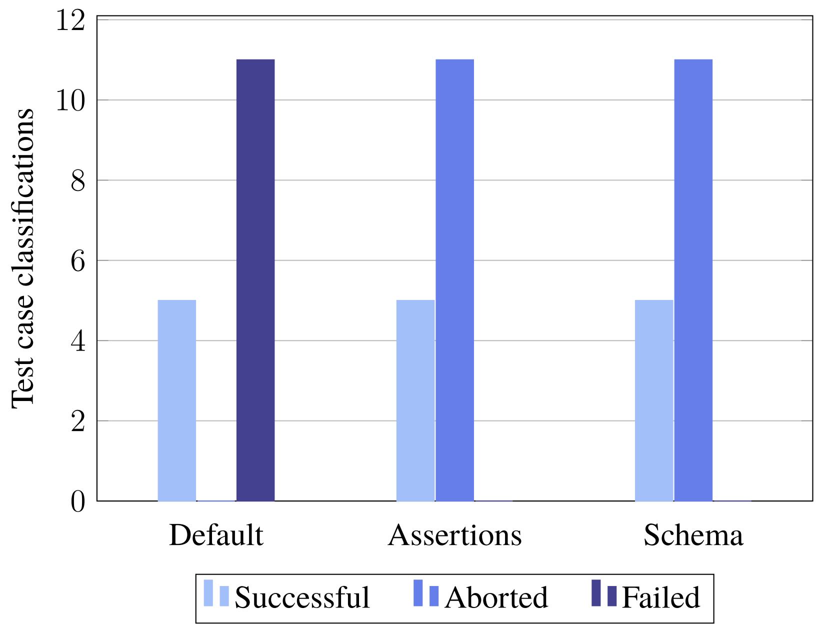

Figure 4.3: View test case results. 16 test cases regarding the creation, deletion, and renaming of database views.

Test Case	**Default**	**Assertions**	**Schema**
Create view with invalid SQL	Failed	Aborted	Aborted
Create view with replacing: Column in view does not exist	Failed	Aborted	Aborted
Create view with replacing: Column in where clause does not exist	Failed	Aborted	Aborted
Create view with replacing: Columns exist	Successful	Successful	Successful
Create view with replacing: Table in view does not exist	Failed	Aborted	Aborted
Create view with replacing: View exists	Successful	Successful	Successful
Create view without replacing: Column in view does not exist	Failed	Aborted	Aborted
Create view without replacing: Column in where clause does not exist	Failed	Aborted	Aborted
Create view without replacing: Columns exist	Successful	Successful	Successful
Create view without replacing: Table in view does not exist	Failed	Aborted	Aborted
Create view without replacing: View exists	Failed	Aborted	Aborted
Drop view: View does not exist	Failed	Aborted	Aborted
Drop view: View exists	Successful	Successful	Successful
Rename view: New view exists	Failed	Aborted	Aborted
Rename view: Old view does not exist	Failed	Aborted	Aborted
Rename view: Old view exists, new view does not exist	Successful	Successful	Successful

Table 4.3: Detailed table of view test case results. The results are identical for the assertion method and the schema method.

4.4 Indices

This section covers results for the nine test cases that are related to creating, and dropping indices.

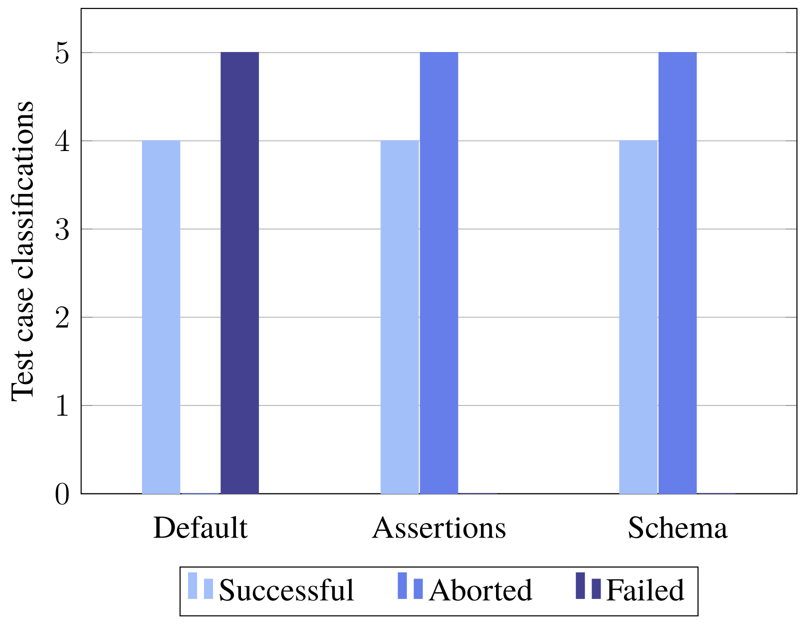

Figure 4.4: Index test case results. 9 test cases regarding the creation, and deletion of table indices.

Test Case	**Default**	**Assertions**	**Schema**
Create index: Column does not exist	Failed	Aborted	Aborted
Create index: Index does not exist	Successful	Successful	Successful
Create index: Index exists	Failed	Aborted	Aborted
Create index: Index exists in other table	Successful	Successful	Successful
Create index: No indices exist	Successful	Successful	Successful
Drop index: Index does not exist	Failed	Aborted	Aborted
Drop index: Index exists	Successful	Successful	Successful
Drop index: Index exists in other table	Failed	Aborted	Aborted
Drop index: No indices exist	Failed	Aborted	Aborted

Table 4.4: Detailed table of index test case results. The results are identical for the assertion method and the schema method.

4.5 Auto Incrementation

This section covers results for the six test cases that are related to adding auto incrementation.

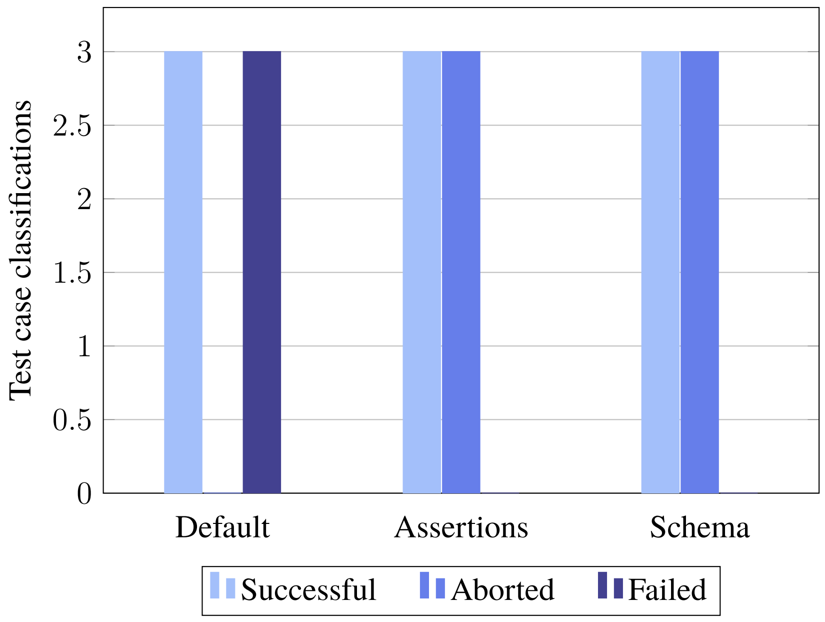

Figure 4.5: Auto increments test case results. 6 test cases regarding the addition of auto incrementing fields.

Test Case	Default	Assertions	Schema
Add auto increment: Column does not exist	Failed	Aborted	Aborted
Add auto increment: Column exists with primary key	Successful	Successful	Successful
Add auto increment: Column exists without primary key	Failed	Aborted	Aborted
Add auto increment: Column is auto incrementing	Successful	Successful	Successful
Add auto increment: Column is of type datetime	Failed	Aborted	Aborted
Add auto increment: Column is of type varchar	Successful	Successful	Successful

Table 4.5: Detailed table of auto incrementation test case results. The results are identical for the assertion method and the schema method.

4.6 Primary Keys

This section covers results for the ten test cases that are related to adding, and dropping primary keys.

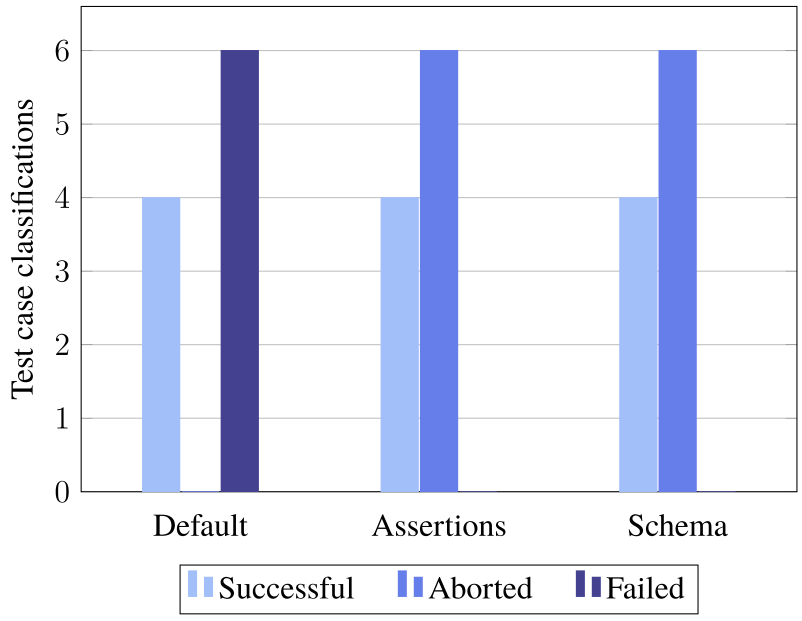

Figure 4.6: Primary key test case results. 10 test cases regarding the addition and deletion of primary keys.

Test Case	**Default**	**Assertions**	**Schema**
Add primary key: Column does not exist	Failed	Aborted	Aborted
Add primary key: Index exists but is not a primary key	Successful	Successful	Successful
Add primary key: No primary key exists	Successful	Successful	Successful
Add primary key: Primary key exists	Failed	Aborted	Aborted
Add primary key: Primary key exists in other table	Successful	Successful	Successful
Drop primary key: Column does not exist	Failed	Aborted	Aborted
Drop primary key: Index exists but is not a primary key	Failed	Aborted	Aborted
Drop primary key: No primary key exists	Failed	Aborted	Aborted
Drop primary key: Primary key exists	Successful	Successful	Successful
Drop primary key: Primary key exists in other table	Failed	Aborted	Aborted

Table 4.6: Detailed table of primary key test case results. The results are identical for the assertion method and the schema method.

4.7 Foreign Keys

This section covers results for the 11 test cases that are related to adding, and dropping foreign keys.

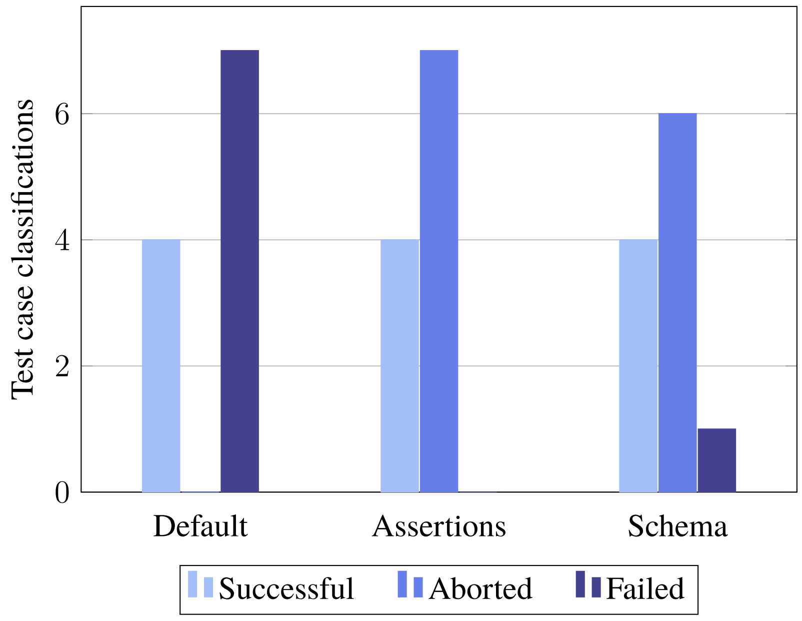

Figure 4.7: Foreign key test case results. 11 test cases regarding the addition and deletion of foreign key constraints.

Test Case	**Default**	**Assertions**	**Schema**
Add foreign key: Base column does not exist	Failed	Aborted	Aborted
Add foreign key: Referenced column does not exist	Failed	Aborted	Aborted
Add foreign key: Referenced column is not primary key	Successful	Successful	Successful
Add foreign key: Has unnamed constraint	Successful	Successful	Successful
Add foreign key: Invalid reference	Failed	Aborted	Failed
Add foreign key: Named constraint exists	Failed	Aborted	Aborted
Add foreign key: Valid references	Successful	Successful	Successful
Drop foreign key: Base table does not exist	Failed	Aborted	Aborted
Drop foreign key: Constraint does not exist	Failed	Aborted	Aborted
Drop foreign key: Constraint exists	Successful	Successful	Successful
Drop foreign key: Constraint is in another table	Failed	Aborted	Aborted

Table 4.7: Detailed table of foreign key test case results. The schema method fails when a foreign key constraint is added to a column containing data that does not reference another record after imposing the constraint. The assertion method aborts the same migration.

4.8 Nullability

This section covers results for the 13 test cases that are related to adding, and dropping not null constraints.

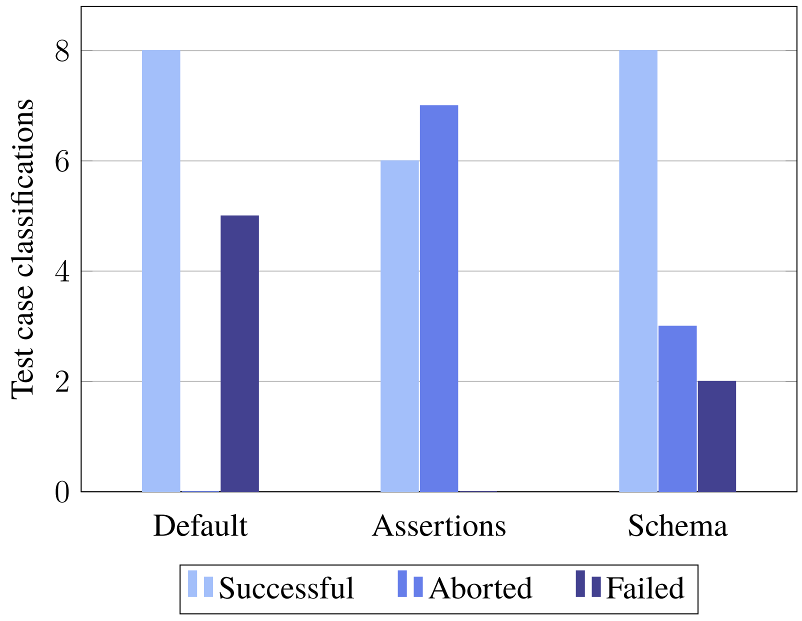

Figure 4.8: Nullability test case results. 13 test cases related to allowing and disallowing null values in tables.

Test Case	**Default**	**Assertions**	**Schema**
Add not null constraint with default value: Column does not exist	Failed	Aborted	Aborted
Add not null constraint with default value: Column exists with null data	Successful	Successful	Successful
Add not null constraint with default value: Column exists with null data and default value	Successful	Successful	Successful
Add not null constraint with default value: Column exists without null data	Successful	Successful	Successful
Add not null constraint with default value: Column is not nullable	Successful	Aborted	Successful
Add not null constraint without default value: Column does not exist	Failed	Aborted	Aborted
Add not null constraint without default value: Column exists with null data	Failed	Aborted	Failed
Add not null constraint without default value: Column exists with null data and default value	Failed	Aborted	Failed
Add not null constraint without default value: Column exists without null data	Successful	Successful	Successful
Add not null constraint without default value: Column is not nullable	Successful	Aborted	Successful
Drop not null constraint: Column does not exist	Failed	Aborted	Aborted
Drop not null constraint: Column exists	Successful	Successful	Successful
Drop not null constraint: Column is nullable	Successful	Successful	Successful

Table 4.8: Detailed table of nullability test case results. The assertion method aborts migrations that add a "not null" constraint that does not allow null values. The assertion method uses business logic described in section 5.3.2. The schema method does not abort the same migration.

4.9 Default Values

This section covers results for the seven test cases that are related to adding, and dropping default values.

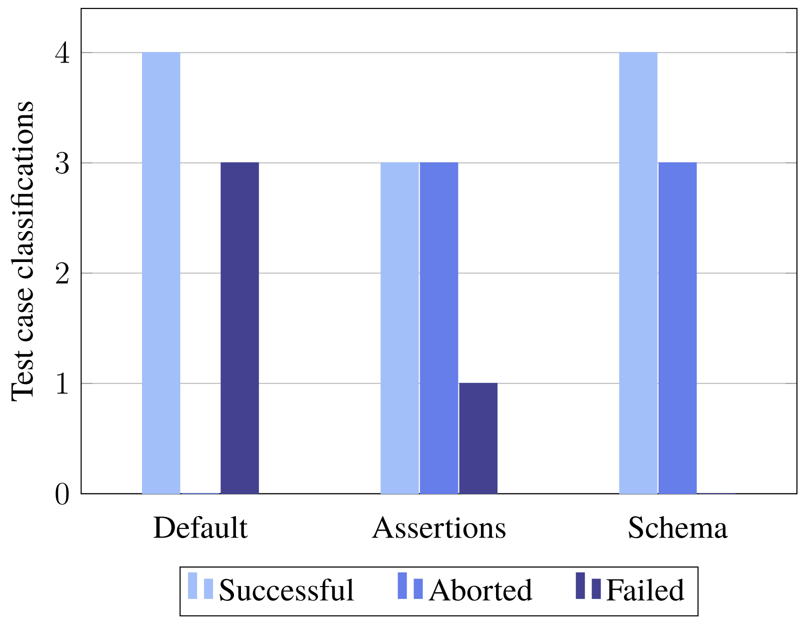

Figure 4.9: Value-related test case results. 7 test cases related to assigning and deleting default values for columns.

Test Case	**Default**	**Assertions**	**Schema**
Add default value: Column does not exist	Failed	Aborted	Aborted
Add default value: Column exists	Successful	Successful	Successful
Add default value: Column exists and has default value	Successful	Successful	Successful
Add default value: Column exists with other data type	Failed	Failed	Aborted
Drop default value: Column does not exist	Failed	Aborted	Aborted
Drop default value: Column exists and has default value	Successful	Successful	Successful
Drop default value: Column exists without default value	Successful	Aborted	Successful

Table 4.9: Detailed table of default value test case results. The assertion method fails to validate a migration that adds a default value to a column with the default value being incompatible with the column's data type. The schema method aborts the same migration.

4.10 Values

This section covers results for the 13 test cases that are related to deleting, inserting, and updating values; and modifying data types.

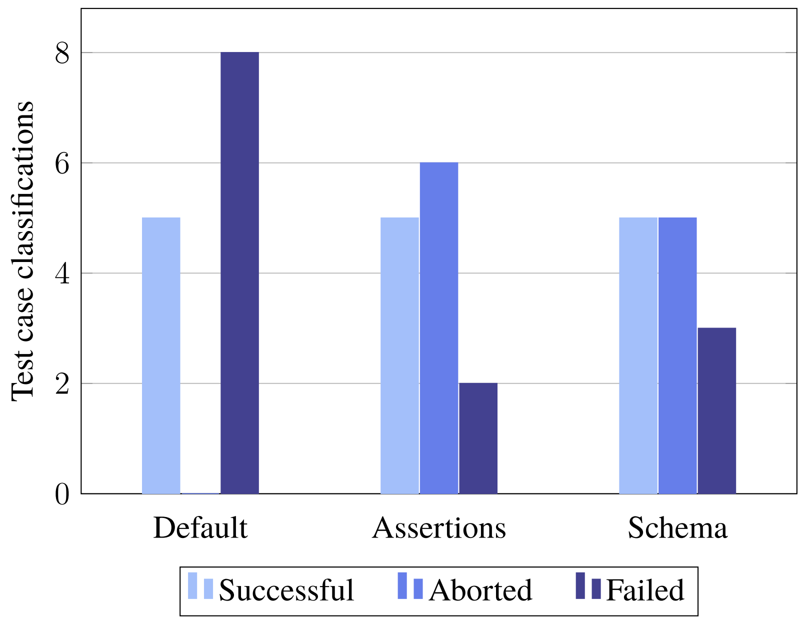

Figure 4.10: Value-related test case results. 13 test cases regarding the deletion, insertion, modification, and updating of records.

Test Case	**Default**	**Assertions**	**Schema**
Delete values: Column exists without null data	Successful	Successful	Successful
Delete values: Column in where clause does not exist	Failed	Aborted	Aborted
Insert values: 1-column unique constraint and duplicates	Failed	Aborted	Failed
Insert values: 1-column unique constraint no duplicates	Successful	Successful	Successful
Insert values: 2-column unique constraint and duplicates	Failed	Failed	Failed
Insert values: 2-column unique constraint no duplicates	Successful	Successful	Successful
Insert values: Column does not exist	Failed	Aborted	Aborted
Modify data type: Column does not exist	Failed	Aborted	Aborted
Modify data type: Compatible data	Successful	Successful	Successful
Modify data type: Incompatible data	Failed	Failed	Failed
Update values: Column exists	Successful	Successful	Successful
Update values: Column in where clause does not exist	Failed	Aborted	Aborted
Update values: Updated column does not exist	Failed	Aborted	Aborted

Table 4.10: Detailed table of value test case results. All of the implemented methods fail to verify unique constraints over two columns when inserting records and when records are modified in such a way that values become incompatible with their respective columns' data types. The assertion method is successful in aborting migrations that insert a record that violates a one column unique constraint. The schema method fails for the same test case.

4.11 Unique Constraints

This section covers results for the six test cases that are related to adding, and dropping unique constraints.

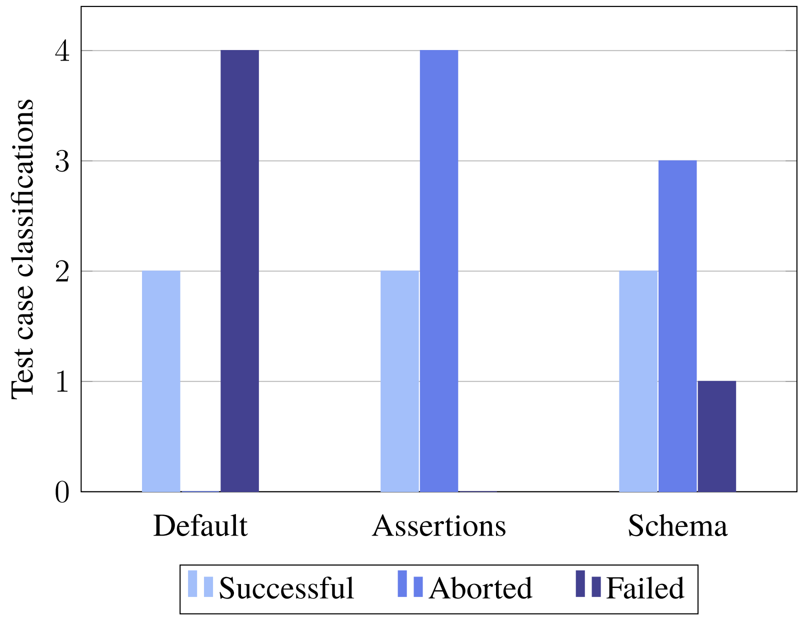

Figure 4.11: Unique constraint test case results. 6 test cases regarding the addition and deletion of unique constraints.

Test Case	**Default**	**Assertions**	**Schema**
Add unique constraint: Column exists with non unique data	Failed	Aborted	Failed
Add unique constraint: Column exists with unique data	Successful	Successful	Successful
Add unique constraint: Column name does not exist	Failed	Aborted	Aborted
Add unique constraint: Constraint exists	Failed	Aborted	Aborted
Drop unique constraint: Constraint does not exist	Failed	Aborted	Aborted
Drop unique constraint: Constraint exists	Successful	Successful	Successful

Table 4.11: Detailed table of unique constraint test case results. The assertion method aborts migrations where a unique constraint is imposed over a column that does not contain unique values. The schema method fails to validate the same type of migrations.

4.12 Summary

The default method did not abort any migration. The schema method achieves the same number of "successful" classifications as the default method. The assertion method includes business logic that affects classification so that more migrations are aborted.

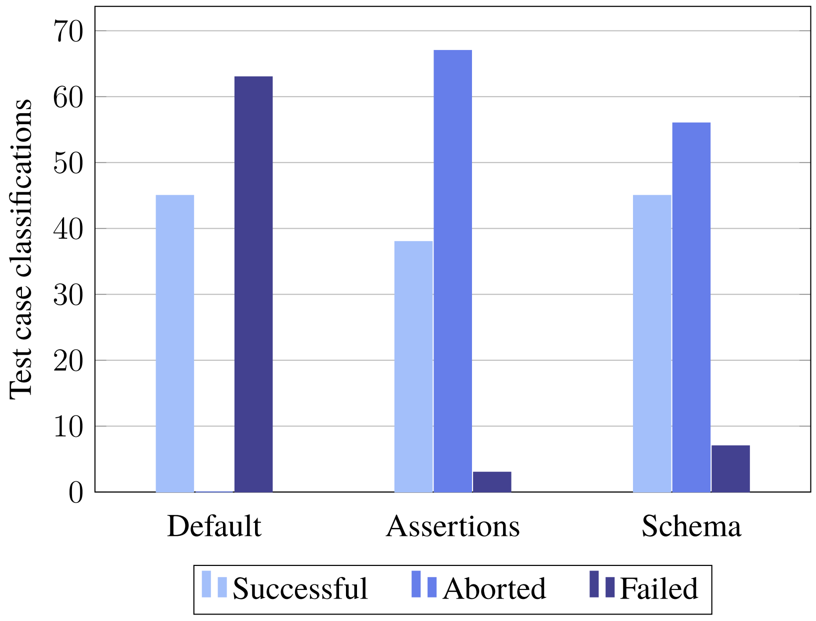

Figure 4.12: A summary of all test results. 108 test cases were used in total. The assertion method aborts the most migration, but fails the fewest. The schema method has the same number of "successful" classifications as the the default method.

Chapter 5

Discussion

This chapter provides some topics for discussion surrounding the requirements in section 1.2. Section 5.1 presents an optimal method for validating and performing migrations as suggested by the results. The other sections in this chapter follow the same format as the requirements, with some ethics, sustainability, and societal aspects in section 5.8 and suggestions for future research in section 5.9. A small summary is included as section 5.10 at the end of this chapter.

5.1 Combined Assertion and Schema Method

The optimal method, as suggested by the results, is a combination of both methods as there are some gaps in the achieved reliability, i.e. adding a default value for a column that exists with another data type, null constraints and migrations that are affected by data records.

A combined assertion and schema method, henceforth denoted as the *CAS method*, can be implemented in order to achieve the best results from both methods without considerably increasing the complexity. Such a combination is shown in figure 5.1. This would combine the results and only allow changelogs that are considered valid by both methods. By doing so, most of the validation, i.e. validating changes that are not affected by data records, can be moved to the testing stages rather than the deployment stages.

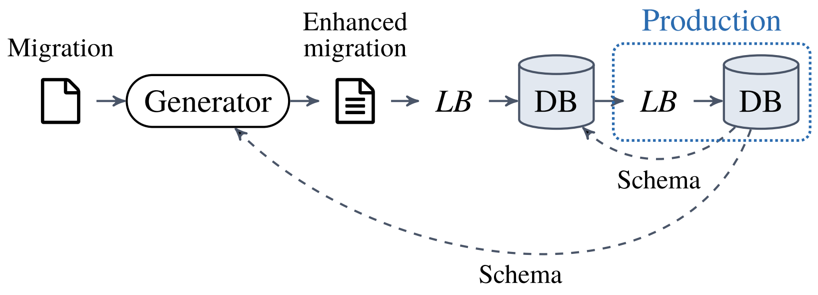

Figure 5.1: The CAS method: a combination of the assertion method and the schema method. The production database's schema is used to get test results earlier in the process. Pre-conditions are used to assert that the complex areas of the production database state are valid. Liquibase, here annotated as *LB* is used to perform all migrations.

The CAS method does not require any additional components compared to the assertion method and the schema method, but the assertion generator can be enhanced by using the exported production database's schema.

5.2 Automation

One of the requirements from section 1.2 is that “all parts of the process involving database changes must be automatable if no other specific requirement prevents it”. By fulfilling this goal, more resources can be allocated to further development rather than on repeating the same steps endlessly. This is absolutely necessary in order to further shorten the time between software releases. It has been shown that database schema migrations can be automated through pipelines similar to those covered in section 2.5.1 as the migrations can already be deployed using a method such as those in section 2.5.2 and now also tested through the use of the assertion method or the schema method.

By implementing means of validating migrations and using one of the existing methods for deploying them, an entirely automated continuous deployment pipeline can be achieved [22, 26, 32, 3]. Both the assertion method and the schema method are entirely automated and performed equally well, and can therefore be used interchangeably, in terms of the requirement for automation. This was expected as both methods were designed to fit within automated build pipelines.

The missing component in previous work is automated migration testing with which testers can focus on automating complex testing scenarios and increasing test coverage of such tests. This further increases the responsibility given to the developers. Both the assertion method and the schema method can be used to test most migrations as indicated by the results in chapter 4. The main difference between them, however, is that the schema method can be used to detect errors in earlier stages than the assertion method.

The CAS method allows early error detection, i.e. that of the schema method, and the reliability of both the assertion method and the schema method without sacrificing automation. The entire migration process can thus be automated.

5.3 Reliability

The requirements stated that the reliability of migrations should be such that "database errors are caught before invalid database migrations are performed". Well-tested software contains fewer bugs than untested software, and the fewer bugs a project has, the more maintainable it is. Fewer bugs also mean greater trust in digital systems [53]. Additionally, the users' productivity is improved which increases revenues, and reduces energy consumption downstream. Any method that adds meaningful tests should increase the project's maintainability. Thus, both the assertion method and the schema method increase software maintainability compared to the default method [54].

This section describes some potential error sources, i.e. failing test cases, for the assertion method and the schema method. It is discussed how such error sources would affect the results and how they have been mitigated.

5.3.1 Data Types

The results from sections 4.9 and 4.10 indicate that it is problematic to validate changes involving different data types.

The schema method can successfully validate changes that modify the default value with an incompatible data type. This is because default values do not rely on data. The assertion method fails to classify the case test case due to the difficulty in generating pre-conditions that cover data type compatibility.

In order to check if two data types are compatible, data must either be typecasted into another data type and compared, or matched against a regular expression. These operations are complex, and may differ between database engines [55]. Followingly, is seems infeasible to achieve complete coverage for changes handling data types in a way that does not affect the interoperability requirement.

5.3.2 Design Decisions

During the implementation of the validation methods, there were some design decisions that had to be made. This is especially relevant for the assertion method as it offers the most flexibility in how pre-conditions are generated.

Design decisions regarding non-altering changes and SQL input had to be considered for the implementation of the assertion method. Rollbacks had to be considered when integrating Liquibase. The rest of this section discusses how the design decisions were made.

Non-altering Changes

The results from chapter 4 show that there are cases where migrations may be successful even though the state is such that nothing is changed.

The changes that may succeed despite being dubious by not altering the schema are:

- Deleting default values for columns without default values.
- Allowing null values in a column that already allows null values.
- Making an automatically incrementing column automatically incrementing.

All changes noted above have an end goal that is already achieved. Although this does not cause any issues on the production database, it highlights a lack of knowledge about the database schema. Furthermore, it could compromise database engine interoperability. After performing the migration, the test and production database will be synced. In this regard, the changes can be allowed to run.

It is, however, good practice to maintain predictability in what changes are allowed to pass pre-condition checks. It is more predictable to only allow actual changes to be performed. Otherwise, the changelog does not provide a complete history of what has happened to the production database. Moreover, this further ensures that the test environment is in sync with the production environment as changes cannot be made more than once.

This design decision means that more migrations are rejected by the method that uses pre-conditions to check if the database can be altered. This increased the consistency between what pre-conditions were used for each change. Removing something from the schema, generally produces an assertion that checks if there is something to remove. Adding something to the schema should thus also require a check to confirm that it does not already exist.

There are no design decisions that have to be taken into consideration with tests that operate solely on a database schema. Therefore, results from the default method and the schema method merely reflect if a change is possible or not. Both the assertion method and the proposed CAS method are affected by design decisions when implementing the pre-condition generator.

SQL Input

Some schema changes in Liquibase, such as creating views [56], and deleting data [57], use raw SQL in its input. Although SQL instructions are out of scope, partial support is required in order to generate pre-conditions for those changes as supporting them greatly increases the reliability. Furthermore, the schema method was expected to perform much worse than the assertion method when dealing with data records. Performing a test on deleting values when a column in the where clause does not exist showed that the schema method aborted the migration just as the assertion method did. This insight is valuable.

Liquibase was selected as it is content-aware, whereas SQL is not. Some tasks are infeasible to perform with change-specific language tags.

Only minimal support of SQL instructions is needed in most cases. The SQL should be validated either through execution in the schema method or through statical analysis in the assertion method. This prevents migrations from breaking once executed in production.

The pre-condition generator implementation used JSqlParser as a simple SQL parser to allow minimal support of changes that are partially SQL instructions. This resulted in some limitations [52]. Mainly, columns are not mapped to the correct tables, and thus some assumptions have to be made. In order to provide some validation, generated pre-conditions check that there exists a pair of mentioned table and column for each column. This means that generated pre-conditions are not entirely correct. Ultimately, this reduces the usability to some extent.

SQL instructions as part of change instructions in Liquibase XML tags can be further researched. This means dropping the constraint of not handling migrations with SQL.

Rollbacks

Some changes include mechanisms for reverting them once they have been made. This is known as a rollback mechanism. Typically, a change that creates a table will generate a rollback that drops it – as will most easily undoable Liquibase changes [58].

If all changes in a changeset includes changes with rollback mechanisms, the changeset can be undone. This means that updates can be undone when including rollback instructions. Thus, it is easy to revert to previous schema versions if the application must be replaced with an older version due to a regression.

Some changes, typically more complex ones, cannot be used to automatically generate rollback code. Liquibase does, however, include a “rollback” tag for manually specifying how changes can be reverted [58]. Downsides of using manual rollback tags are: larger file sizes, increased complexity, and increased development time. Furthermore, the use case is limited as database schemas should typically be rebuilt from scratch during development, rather than being rolled back.

It is not always possible to find accurate rollbacks for some changes. If a change deletes all records matching some condition, there is no viable way of reinserting those records again in a rollback [3]. Thus, rolling back such changes is infeasible.

Rollback mechanisms could be enforced for software projects, but the benefits seem small. The exception is for bug-fixes, and minor performance enhancements. If such updates fail, it could be useful to smoothly revert to a previous version. This is typically a case where the software version is also backwards compatible.

5.3.3 Data Samples

The actual data used when testing is also a potential error source.

Test Data

The main reason for adding tests to database schema migrations is that data differs between test and production environments. This is, however, also an error source for the results from testing the two methods.

Test data is typically much more well-formed than production data that can even be partially corrupt if written by older application versions [3]. All test cases in this work use data that is both well-formed, and known. This could pose an issue as tests could run into the same problem that it is supposed to solve. The risk is lowered by providing positive, and negative samples for each tested change. Multiple samples of failing migrations are also provided where possible.

Randomized test data could be used in order to provide somewhat less known test cases [3]. There is, however, an issue with it: some structure is needed in the generation in order to make it compatible with the database schema, meaning that the added benefit is minuscule.

Database Size

The general size of data that is modified could alter the outcome of testing. Large data collections may cause timeouts, or loss of service that could halt the migration process and leave the database in an invalid state. This would typically occur on the production database server as it would store more data than that in the test environment. Migrations that would otherwise fail due to high complexity are not altered by any methods. Hence, the main concern is the generated pre-conditions that add some extra queries. Most pre-conditions mostly look through smaller internal tables that describe the database structure. Such pre-conditions would result in highly scalable queries.

Pre-conditions such as ensuring uniqueness pose more of an issue as the resulting query would look through all rows in a table. If this table is too large, this process may fail. This would, however, also halt the migration process.

Database size is not a considered factor when producing the results for this work. This means that the scalability may be lower than expected. Additional tests with large amounts of generated data can be performed in order to mitigate this.

5.3.4 Environment

The environment in which migrations are performed can also affect the test outcome, and validity, of migrations. This also applies to validation methods.

Database Engine

There are many different types of relational databases, and they do not all use the same SQL instructions. Furthermore, they may not always offer the same functionality. Thus, generated pre-conditions may not produce the same result for all database engines, or versions. This risk is minimized by versioning the test generation, and by allowing specialized SQL instructions to be generated for different databases.

Unknown Database Implementation Details

The schema method will always use the database engine's own rules and will therefore not be affected by unknown details about the inner workings of a certain database engine. This is not the case with the assertion method as it relies purely on the generated pre-conditions. If there are any validation rules that are not known when implementing the assertion method, the consequences may be terrible. Some rules within the schema may be stored differently although they could have the same impact on migrations. A primary key may not be stored in the same way as a uniqueness constraint even though it enforces uniqueness. If such implementation details are overlooked when designing a pre-condition mapping such as the one in section 3.3.1, the coverage will decrease and the assertion method's reliability will suffer.

Testing Environment

Database servers in the test environments should be of the same type and version as those in the production environments. This ensures that nothing can vary in the execution of deterministic SQL queries. If there is a difference, test results from the two methods cannot be trusted.

Another varying factor is what programs are installed on the database servers. A migration may utilize programs, files, or directories during execution. If are present in the testing environment, but missing from the production environment, migrations may fail. On the other hand, if they are missing from the testing environment, it will be detected by both test methods.

The testing environment should mimic the production environment in its entirety. The only difference should be the data that is stored in their databases.

Concurrency

If the production database is utilized by a running application during migration, some issues may arise due to varying data. This is mostly applicable to the method of generating pre-conditions as the other method cannot test data. It may take some time between checking pre-conditions, and completing the migration. This can cause issues.

A potential scenario:

1. State: The table “items” has a field called “name”. It is unique in all records, but there is no unique constraint.

2. Migration: It is desirable to make “items.name” unique by imposing a constraint.

3. Migration: Check if “items.name” is unique. It is.

4. Application: Add a new record to “items”. Its name already exists. It is not unique, but there is no constraint. It is ok.

5. Migration: Add unique constraint to “items.name”. Failure, records are not unique.

In the scenario, some changes may have been performed in the migration, leaving the database in a broken state. The pre-condition allowed the migration to be performed, but the process failed due to concurrent modification [16]. This issue can be completely avoided by not utilizing zero-downtime database deployments. Another way of avoiding it is by guaranteeing uniqueness through logic prior to performing the migration. This does, however, require some additional planning, development time, and an extra software release.

It is infeasible to lock large databases between checking pre-conditions and completing the migration since there transaction locks are limited [41, 42, 43]. A desired solution is scalable. Thus, this method of locking cannot be used to solve issues related to concurrent database access.

5.4 Traceability

All changes in the production database must be traceable back to the author according to the requirements laid out in section 1.2.

There already exists a mechanism for doing so in Liquibase. All “changeSet” tags must specify an “author” attribute denoting what developer was in charge of the changes [40]. This is helpful, but it is not enough if the requirement is to be taken seriously.

In order to provide accurate traceability down to specific software versions, the changelog and its contents must be checked in under source control. This can also provide some additional information about code reviews, and pull request comments, if applicable.

The “author” attribute can be arbitrary, which is not good for traceability. This can be fixed by coupling it to the code committer by using an email address, or a cryptographic signing key. Before merging commits to the master branch, this coupling has to be verified. If the “author” information does not match the committer, the merge should be rejected. This can be done by an automatic tool, and ensures that author information remains valid even if commits are squashed.

For git, the committer’s email can be obtained using the command `git show -s --format=%ce COMMIT_ID`. Note that the committer is not always the original author. The original author’s email can be obtained using the command `git show -s --format=%ae COMMIT_ID` [59].

Database changes without the use of Liquibase, or the single source of truth, the changelog, should not be allowed. This ensures a completely traceable chain from commit to change in production.

Each software project’s owners must then only decide who is responsible for each change: the author, or the committer? This is only applicable if the author and committer is not the same developer. Thus, accountability is also inferred.

5.5 Interoperability

One of the requirements stated that the solution works even if database engine and version are changed. This section discusses how well that requirement holds.

There are many different types of databases in use. These types include relational databases, graph databases, object-oriented databases, and document stores to name a few. Only relational databases were considered in this work as they were the most common in February 2019 [1]. This drastically reduces the compatibility. It would, however, be a lot more difficult to find a good solution for managing migrations in a NoSQL database that does not use a schema to impose integrity constraints but instead require data to be reformatted [60].

5.5.1 Compatibility

The most important component of all tested methods is Liquibase. Thus, the interoperability of the validation methods is dependent on the interoperability of Liquibase.

The amount of supported databases should be the same for the validation methods as for normal Liquibase usage. 13 different databases are supported by default as is listed in table 2.1.

In addition to the 13 supported databases, Liquibase can be extended with custom plugins in order to further increase database support. Furthermore, due to Liquibase being open source, the code can be modified in order to provide more functionality.

The syntax for authoring changes has already been decoupled from database-specific SQL dialects. Apart from missing fields that are mandatory only for some database engines, the few custom SQL fields in some changes may require revision when changing database engine, or version.

Some pre-conditions may differ between database engines. The interoperability can be increased by allowing pre-conditions to generate different SQL queries for different databases. This way, assertions can produce the intended results for both MySQL, Oracle RDBMS, and PostgreSQL. By letting generated pre-condition queries depend on the database engines, further developments in relational databases can be accommodated. The result is a more flexible system for testing schema migrations.

In conclusion, the database engine support should be considered more than enough. The time required to switch database engine, or version, should be considered minuscule.

5.5.2 Liquibase Interaction

Liquibase is required for all of the mentioned validation methods. It can, however, be integrated in a few different ways, each having its own pros and cons.

It is desirable to use the same approach when testing, and when deploying to production, to minimize differences in order to increase proper test coverage. Local development is allowed to be different as bugs will still be detected by the pipeline.

Furthermore, developers do not need to know about specific implementation details in a centralized build environment. The same applies to the precondition generator used in the assertion method. It is, however, beneficial to know about the validation process – knowledge of packaging is not as important.

Installed

One way of interacting with Liquibase is to have it installed through a package manager. This makes it easy to select a specific version to use.

The downside of installing Liquibase is that multiple versions cannot be installed at once. This means that every software project requires a separate machine, which can be problematic when automating build pipelines as hardware redundancy is greatly increased.

Another downside of installing any package is trust. Since Liquibase is open source, it is possible to set up an automated process for building the software in-house, thus avoiding packages from other manufacturers. For applications handling sensitive data, this may be the best option as the packaged version of Liquibase will eventually receive credentials for managing the production database.

Both issues with trust, and version management, can be solved through build slave virtualization, or containerization. Every build can set up the required environment for performing migrations. Thus, Liquibase versions can be selected on a per project basis.

Maven Plugin

Maven is a widely used build tool for Java applications. It can be used to compile Java bytecode from source files. There is also support for custom plugins.

A plugin called *liquibase-maven-plugin* is provided, and can be used to integrate a desired version of Liquibase in the software development process.

It can be included in an application's *pom.xml* configuration file [61]. This is illustrated in listing 11.

```
<plugin>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-maven-plugin</artifactId>
  <version>3.6.3</version>
  <configuration>
    <propertyFile>
      /var/datical/liquibase/config.properties
    </propertyFile>
  </configuration>
</plugin>
```

Listing 5.1: Configuration excerpt for including Liquibase as a Maven plugin.

The plugin requires a JDBC (Java Database Connectivity) [62] driver to be included in order to communicate with the database. This dependency should also be included in the *pom.xml* file. The database engine type determines which dependency to include.

Database credentials, and other machine-specific details, are stored in a properties file such as */var/datical/liquibase/config.properties*.

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.13</version>
</dependency>
```

Listing 5.2: Dependency for a JDBC driver that works with MySQL 8.

Once the plugin is installed, the master changelog can be executed with the command `mvn liquibase:update` [61].

The main drawback of using Liquibase as a Maven plugin is that Maven needs to be installed, or otherwise usable on the system from which migrations are performed. This can potentially cause issues with version management of Maven if multiple versions are used for different software that is used on the same machine – which may be the case in a build pipeline.

Due to this, using Liquibase as a Maven plugin is best for local development where version control is project specific, and where Maven already is already used for dependency management.

Manually Handled Java Archive

Another way to use Liquibase is through a manually uploaded Java Archive with a ".jar" file extension. By doing so, applications become dependent on the filesystem. Versions can be handled through unique file names.

There is no need for a separate process for managing archives, but automation can be employed to automatically upload all new versions. Java archives can be stored in a shared filesystem in order to simplify maintenance.

The most negative aspect of this approach is that database testing servers need to have all archive versions available. Depending on the way that this is handled, it ultimately leads to less control over what Liquibase versions that are used in software – most available versions are likely not used.

Pre-condition Generator

The pre-condition generator required for the assertion method is also needed in the build pipeline. It is not required in local development.

Versioning can also be required as differences in input formats, and allowed changes, may differ over time. Major changes in Liquibase's behavior could break the generator's backward compatibility.

Ultimately, the generator should be packaged, or made available using the same method as Liquibase.

5.5.3 Legacy Systems

It is simple to use Liquibase when starting new software projects. It is, however, harder to use Liquibase in older, aging, projects. This section discusses possible ways of solving this issue.

Liquibase Without History

Although it is possible to start using Liquibase without a history, and only letting it handle future updates, it makes it hard to set up a viable test environment. Thus, adoption methods that cannot be used with an application right away are rejected. It should be noted that it would still work with tests that use the production schema.

This method is quick and only requires an empty changelog to be written. Further database changes should be performed by appending changesets to the changelog.

Development with this method is not recommended as there is no way to reset the database to a certain state. If the database is lost, there is no way to rebuild its schema.

Export Raw SQL Schema

A better approach is to dump the schema from the production database. This SQL can then be wrapped in an “sql” tag that is placed as the sole item in the first changeset of the changelog.

Liquibase is then able to create a fresh database from scratch. This allows both methods to be used.

This approach does not allow maximum portability, and could make it hard to change database engine in the future due to heavy reliance on SQL.

Fake Liquibase History

The third, and most time consuming method, is to fake a complete Liquibase history within a changeset. Normal Liquibase tags are used when setting up the initial state.

In order for this method to work, the result of executing the changelog must match the production database exactly. Liquibase changes are more general than SQL, meaning that it should be easier to swap database engine in the future.

If there is no need for the database engine to be easily replaceable, the additional time required for this method may not be justifiable.

5.6 Data Integrity

If a migration fails halfway through execution, data may be lost. Backups can mitigate the risk of data loss at the cost of time, and storage capacity [3].

Continuous backups can be taken in order to reduce the time that it takes to run a migration using any method [3]. Restoring the backup if a migration fails, however, is a process that can cannot be shortened.

By reducing the need for using production data during testing, less people need access to it. This increases privacy, and ultimately makes the data more tamper-proof.

Although not noticeable from the outside, trust within corporations may increase as colleagues no longer have access to their coworker's data. This allows employees to use software that they develop with others, and where sensitive data is stored.

The assertion method is the best method when it comes to guarantees about data integrity after performing a migration. The schema method falls short when there are uniqueness constraints as can be seen in section 4.10.

The default method leaves no guarantees for maintaining data integrity as is made evident by the results throughout chapter 4.

5.7 Scalability

One requirement from section 1.2 is that the validation methods should be applicable to “small and large databases alike”. Although not an initial requirement, the automation should preferably be scalable.

5.7.1 Changelog Filesize

One factor that may limit scalability is that of filesizes. The assertion method produces an enhanced changelog through its pre-condition generator. The produced changelog should be archived along with the built software.

The added pre-conditions result in a file output that is larger than the file input. This means that the assertion method requires more storage than the other proposed validation methods.

Most changes result in a constant amount of pre-conditions being generated. There are, however, changes that deal with data that will map to a non-constant amount of assertions. One example is when data is inserted and uniqueness constraints are tested. When single column examining uniqueness for n records of data in k columns are inserted, k assertions are generated for each record. Thus, $k \cdot n$ additional pre-conditions are added to the output, resulting in a linear growth.

The same logic can be applied to examine up to two column uniqueness constraints for n records of data in k columns: $n(k + \binom{k}{2})$ additional pre-conditions are required.

Even further, in order to check uniqueness constraints for all possible uniqueness, the additional pre-conditions make it infeasible as it follows the expression in figure 5.2.

$$n(\binom{k}{0} + \binom{k}{1} + \binom{k}{2} + \cdots + 1) = n \cdot 2^k$$

Figure 5.2: Expression for the number of assertions needed to check if uniqueness constraints are violated for n records in k columns. This number includes the pre-condition required for exiting when there is no uniqueness constraint.

The changelog’s filesize is not affected when using the default method or the schema method, as the original changelog file is used, and can to some extend be controlled by improving the generator component. Duplicate pre-conditions can be removed. Furthermore, redundant checks can be left out as some assertions may completely cover others. Checking a table exists is a subset of checking that a column exists.

Moreover, the generator could be optimized if it had knowledge about the production database's schema. Previous migrations in the changelog should not be relied upon as they may differ from the schema in production. If uniqueness constraints are known when making assertions, only n pre-conditions will be generated. This drastic change in complexity would make it feasible to check uniqueness constraints even with many records, and many columns. Finally, such knowledge would reduce the need for authoring redundant uniqueness evaluations when no such constraints exist.

The CAS method could improve scalability by only generating pre-conditions for changes that can be affected by data records.

5.7.2 Execution Time

Both the assertion method and the schema method take more time than the default method (see section 3.2).

For the schema method, this time is mainly performing migrations an additional time in a testing environment. The migration is performed on a database without data. This is a very quick procedure, and the time taken to do so is therefore negligible.

The additional time taken for the assertion method to complete will depend on the pre-conditions that are generated. The time taken to generate pre-conditions is negligible compared to actually executing them. While most pre-conditions are fairly quick, some are not. The most time consuming pre-conditions are those that query data. Examples of such pre-conditions are: checking for uniqueness, checking if a value exists, and validating foreign key references.

The most time consuming pre-conditions can also come in multiples, as is the case when validating uniqueness when inserting data. Hence, the reasoning in section 5.7.1 can be applied, resulting in an even more time-expensive operation.

It should be noted that most pre-conditions may not always be executed even though they exist in the migration changelog. Logic using "AND" or "OR" operations may terminate prior to checking all child pre-conditions. For uniqueness constraints, the first check is typically checking if the constraint exists – something that can be done well ahead of performing the much more expensive check that actually looks at all the records.

It should be noted, however, that the damages caused by not catching a faulty migration takes much longer to restore than any of the additional time that is introduced by running any of the methods.

5.7.3 Pipeline Workers

One way to scale up pipelines is to use multiple workers to process concurrent jobs. This increases throughput at the expense of hardware [63]. Underutilization of hardware is likely as the number of build jobs vary over the course of a workday. On the other hand, the waiting time between commit and verification is reduced. This should make sense economically.

All workers must be set up with labels indicating what their capabilities are. This means that migration-performing workers should be given a *liquibase* tag.

Containerization, and virtualization could be used in a testing environment to level utilization, and promote resource sharing. Resource sharing may not always be feasible in high-security production environments, but from a testing standpoint, it should not affect the outcome of test results.

5.7.4 Database Allocation

A problem arises due to running migrations in parallel: workers cannot have write access the same database as they would lock each other out [39]. This means that each worker must be assigned to a unique database instance if it is to use database operations.

The database configuration must be done as a part of the build job. The process can be inserted as an automated step of running the job, or it can be hard-coded for each worker. All jobs should make sure that accessed databases are being reset after use as uncleaned databases will cause issues in other builds.

Even though all software installations may be standardized, build jobs must be able to use varying credentials. If one machine – virtual, or physical – has multiple workers, it should provide configuration files for each one. This is demonstrated in figure 5.3.

There is also a possibility to allocate databases dynamically. Each build job could be responsible for creating, migrating, and finally destroying a database instance. In this case, credentials can be generated inside the build job, distributed to a target server, and used to create an empty database.

Dynamic allocation offers greater flexibility at the price of some added complexity. Dynamically allocated database instances must be destroyed following the termination of jobs. Otherwise, resources will leak until the pipeline crashes.

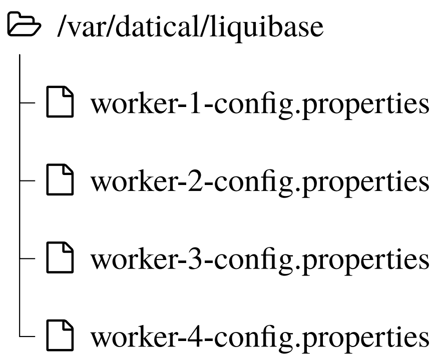
/var/datical/liquibase
worker-1-config.properties
worker-2-config.properties
worker-3-config.properties
worker-4-config.properties

Figure 5.3: Typical per-worker configuration on a system with four workers.

5.8 Ethics, Sustainability & Societal Aspects

Penzenstadler (2013) stated that the maintainability of a software project affects the general sustainability of it [54]. One of the goals with continuous integration (see section 2.5) and its derivatives is to increase development speed and maintainability through the use of automated testing [22]. This study brings maintainability to database schema migrations through those very same techniques.

Although the added testing phases result in more computation being performed for each software update, the increased energy usage is justified if the added methods can prevent loss of service due to undetected errors. Resource utilization, and service functionality, should maximized according to the "system usage aspect" of Penzenstadler's work [54]. Any downtime that results in under-utilization of production systems is significantly worse than if development systems are affected. This is mainly due to resource sharing between developers in a virtualized or containerized testing environment. Resource sharing may, however, not be feasible in high-security production environments due to the risk of vulnerabilities such as container escapes and weak local access control [64].

Increased traceability, and accountability, translates to increased developer responsibilities. Database administrators have less need for verifying that database changes are valid prior to migrating changes if the migrations have already been validated through a validator that uses a method such as the assertion method or the schema method. This frees up resources that can instead be allocated to increase application performance by improved database indexing. The time between releases should also be reduced by the increased digitalization resulting in an overall increase of the rate at which new changes are made to servers and bugs are fixed.

Automation in software building, and testing, also means that fewer manual tests are required. Instead, testers can focus on automating complex testing scenarios and increasing automated test coverage. This further increases the responsibility given to the developers while increasing the profitability of software companies by potentially freeing up workloads or reducing technical debt. Increased test coverage should increase application reliability, which shifts the general opinion on a digital society as stable software can be achieved in greater quantities. Fewer bugs means greater trust in digital systems [53].

Finally, the increased separation of data during production and testing means that the customers' privacy is increased. Testers do not need to have access to production data if they can instead make assertions about how certain changes behave. The increased reliance on open-source software such as Liquibase [35], and reduced reliance on companies that may go out of business, should provide further economical benefits while also promoting openness and trust in the software development community [65].

5.9 Future Research

The assertion method's scalability is only examined theoretically in this report. Further research could be performed on large database states from deployed production databases or from randomized content.

Moreover, future research can focus on finding a scalable solution for verifying the compatibility of data types as this is problematic with the two proposed validation methods.

Finally, research can be made on developing a hybrid validation method that uses the schema from the schema method to generate pre-conditions without the complexity overhead that prevents the validation of higher-order unique constraints as discussed in section 5.7. One such hybrid approach is the CAS method that is presented in section 5.1.

5.10 Summary

This work can be used to perform automation throughout the entire software build process using a pipeline that is altered to work with migrations in the form of Liquibase changelogs. The reliability provided by the assertion method and the schema method is generally high, although data types and scalability issues with complex uniqueness constraints reduce it to some extent. The interoperability is good through the 13 Liquibase-supported databases and a well-known XML format that is interpreted by open-source software. The data integrity is preserved and all changes in the production database can be traced back to their source code commit and its author.

A CAS method (Combined Assertion and Schema method) is presented in section 5.1. It uses the reliability in validating data compatibility from the assertion method together with the simpleness and rapid feedback of the schema method. By combining the best parts of the assertion method and the schema method, the CAS method fulfills all of the requirements from section 1.2.

Chapter 6

Conclusions

Two validation methods were implemented in order to prevent invalid database migrations from being performed. They required reliability, preservability, automation, interoperability, and scalability throughout the migration process without leaking data from the production environment. Traceability could also be achieved. The results showed that the assertion method (see section 3.3) can be used to verify migrations in which data records can alter the outcome whereas the schema method (see section 3.4) cannot. The assertion requires design decisions to be taken in regards to if non-altering changes should be allowed or not while the schema method does not.

It is infeasible to check data type compatibility without using a schema from the production database. The schema method verifies type compatibility when performing migrations, but is unable to do so when modifying data. Furthermore, there is no feasible way to validate uniqueness constraints imposed on multiple columns with duplicate data records without accessing the production data. The assertion method provides marginally better support for validating migrations than the schema method, but does so during deployment rather than testing. The schema method is easier, and takes less time, to implement than the assertion method and can be used to detect faulty migrations during the testing stages of a continuous integration pipeline.

Finally, both methods can be used to form a combined assertion and schema method, the CAS method (see section 5.1), that would only fail in at most 2/108 of the implemented test cases although it should be possible to further increase the reliability with it. An entirely automated continuous integration and deployment process can thus be implemented for database schema migrations.

Bibliography

[1] solid IT GmbH. *DBMS popularity broken down by database model.* 2019. URL: https://db-engines.com/en/ranking_categories (visited on 02/21/2019).

[2] Katarina Grolinger and Miriam AM Capretz. "A unit test approach for database schema evolution." In: *Information and Software Technology* 53.2 (2011), pp. 159–170.

[3] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[4] Carlo Curino et al. "Automating the database schema evolution process." In: *The VLDB Journal—The International Journal on Very Large Data Bases* 22.1 (2013), pp. 73–98.

[5] Wikipedia contributors. *SQL — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/wiki/SQL (visited on 04/15/2019).

[6] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Nov. 26, 2008. URL: https://www.w3.org/TR/REC-xml/ (visited on 06/08/2019).

[7] Edgar F Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[8] Joachim Biskup. "Achievements of relational database schema design theory revisited." In: *International Workshop on Semantics in Databases*. Springer. 1995, pp. 29–54.

[9] Barry E Jacobs. "On database logic." In: *Journal of the ACM (JACM)* 29.2 (1982), pp. 310–332.

[10] Henryk Rybiński. "On first-order-logic databases." In: *ACM Transactions on Database Systems (TODS)* 12.3 (1987), pp. 325–349.

[11] Scott W Ambler and Pramod J Sadalage. *Refactoring Databases: Evolutionary Database Design (paperback)*. 2006.

[12] Ben Shneiderman and Glenn Thomas. "An architecture for automatic relational database system conversion." In: *ACM Trans. Database Syst.* 7.2 (1982), pp. 235–257.

[13] John F Roddick. "A survey of schema versioning issues for database systems." In: *Information and Software Technology* 37.7 (1995), pp. 383–393.

[14] Javier Tuya et al. "A Controlled Experiment on White-box Database Testing." In: *SIGSOFT Softw. Eng. Notes* 33.1 (Jan. 2008), 8:1–8:6. ISSN: 0163-5948. DOI: 10.1145/1344452.1344462. URL: http://doi.acm.org/10.1145/1344452.1344462.

[15] Amir Hassan Bahmani, Mahmoud Naghibzadeh, and Behnam Bahmani. "Automatic database normalization and primary key generation." In: *2008 Canadian Conference on Electrical and Computer Engineering*. IEEE. 2008, pp. 000011–000016.

[16] Michael de Jong, Arie van Deursen, and Anthony Cleve. "Zero-downtime SQL database schema evolution for continuous deployment." In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2017, pp. 143–152.

[17] Michael de Jong and Arie van Deursen. "Continuous deployment and schema evolution in SQL databases." In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. IEEE. 2015, pp. 16–19.

[18] H.D. Foster, A.C. Krolnik, and D.J. Lacey. *Assertion-Based Design*. Springer US, 2006. ISBN: 9781402080289. URL: https://books.google.se/books?id=BDcy9JsM-%5C_gC.

[19] Alan Turing. "Checking a large routine." In: *The early British computer conferences*. MIT Press. 1989, pp. 70–72.

[20] Charles Antony Richard Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[21] Paul Jorgensen. *Software testing : a craftman's approach*. eng. 2. ed.. Boca Raton, FL: CRC Press, 2002. ISBN: 0-8493-0809-7.

[22] Martin Fowler and Matthew Foemmel. “Continuous integration.” In: *Thought-Works* 122 (2006), p. 14. URL: http://www.thoughtworks.com/Continuous_Integration.pdf.

[23] Srinivas Nidhra and Jagruthi Dondeti. “Black box and white box testing techniques-a literature review.” In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[24] Felix Redmill. “Understanding the use, misuse and abuse of safety integrity levels.” In: *Proceedings of the Eighth Safety-critical Systems Symposium*. Citeseer. 2000, pp. 8–10.

[25] Derk-Jan De Grood. “Smoke Test.” In: *TestGoal: Result-Driven Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 293–296. ISBN: 978-3-540-78829-4. DOI: 10.1007/978-3-540-78829-4_17. URL: https://doi.org/10.1007/978-3-540-78829-4_17.

[26] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. “Climbing the "Stairway to Heaven"–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software.” In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE. 2012, pp. 392–399.

[27] Paolo Guagliardo and Leonid Libkin. “A formal semantics of SQL queries, its validation, and applications.” In: *Proceedings of the VLDB Endowment* 11.1 (2017), pp. 27–39.

[28] Manik Jain. “Automated Liquibase Generator And Validator(ALGV).” In: *International Journal of Scientific & Technology Research* 4 (Sept. 2015), pp. 248–256.

[29] Radoslaw Dziadosz et al. “Liquibase: Version Control for Database Schema.” In: *Jyväskylän ammattikorkeakoulu* (2017).

[30] Sirkka-Liisa Jämsä-Jounela. “Future trends in process automation.” In: *IFAC Proceedings Volumes* 40.1 (2007), pp. 1–10.

[31] Akond Ashfaque Ur Rahman et al. “Synthesizing continuous deployment practices used in software development.” In: *Agile Conference (AGILE), 2015*. IEEE. 2015, pp. 1–10.

[32] Lianping Chen. “Continuous Delivery: Huge Benefits, but Challenges Too.” In: *IEEE Software* 32 (Mar. 2015), pp. 50–54. DOI: 10.1109/MS.2015.27.

[33] Eric A Brewer. “Lessons from giant-scale services.” In: *IEEE Internet computing* 5.4 (2001), pp. 46–55.

[34] Fredy Oertly and Gerald Schiller. “Evolutionary database design.” In: *[1989] Proceedings. Fifth International Conference on Data Engineering*. IEEE. 1989, pp. 618–624.

[35] Datical. *Liquibase*. 2019. URL: `http://www.liquibase.org/index.html` (visited on 04/10/2019).

[36] Datical. *Supported Databases*. 2019. URL: `http://www.liquibase.org/databases.html` (visited on 04/10/2019).

[37] Datical. *Database Change Log File*. 2019. URL: `http://www.liquibase.org/documentation/databasechangelog.html` (visited on 04/10/2019).

[38] *DATABASECHANGELOG table*. 2019. URL: `https://www.liquibase.org/documentation/databasechangelog_table.html` (visited on 06/18/2019).

[39] *DATABASECHANGELOGLOCK table*. 2019. URL: `https://www.liquibase.org/documentation/databasechangeloglock_table.html` (visited on 06/18/2019).

[40] Datical. *<changeSet> tag*. 2019. URL: `http://www.liquibase.org/documentation/changeset.html` (visited on 04/10/2019).

[41] Oracle. *Limits on InnoDB Tables*. 2019. URL: `https://dev.mysql.com/doc/refman/8.0/en/innodb-restrictions.html` (visited on 06/18/2019).

[42] Oracle. *Undo Logs*. 2019. URL: `https://dev.mysql.com/doc/refman/8.0/en/innodb-undo-logs.html` (visited on 06/18/2019).

[43] Oracle. *Configuring Transaction Size*. 2019. URL: `https://docs.oracle.com/cd/E23095_01/Platform.93/ATGProgGuide/html/s1008configuringtransactionsize01.html` (visited on 06/18/2019).

[44] Datical. *Preconditions*. 2019. URL: `http://www.liquibase.org/documentation/preconditions.html` (visited on 04/10/2019).

[45] Sebastian Mueller and Raphael Müller. "Conception and Realization of the Versioning of Databases Between Two Research Institutes." In: *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017): Barcelona, Spain, October 8-13, 2017.* 2018, TUPHA041. DOI: `10.18429/JACoW-ICALEPCS2017-TUPHA041`.

[46] *Git.* June 8, 2019. URL: `https://git-scm.com/` (visited on 06/11/2019).

[47] Datical. *Bundled Liquibase Changes.* 2019. URL: `https://www.liquibase.org/documentation/changes/index.html` (visited on 05/16/2019).

[48] Datical. *Liquibase Best Practices.* 2019. URL: `http://www.liquibase.org/bestpractices.html` (visited on 04/10/2019).

[49] Tom Preston-Werner. *Semantic Versioning 2.0.0.* Jan. 2019. URL: `https://semver.org/spec/v2.0.0.html` (visited on 04/23/2019).

[50] Oracle. *MySQL.* 2019. URL: `https://www.mysql.com/` (visited on 05/07/2019).

[51] Oracle. *Download Connector/J.* 2019. URL: `https://dev.mysql.com/downloads/connector/j/` (visited on 05/07/2019).

[52] Tobias (wumpz). *JSqlParser.* 2019. URL: `https://github.com/JSQLParser/JSqlParser/wiki` (visited on 04/24/2019).

[53] M. Zhivich and R. K. Cunningham. "The Real Cost of Software Errors." In: *IEEE Security Privacy* 7.2 (Mar. 2009), pp. 87–90. ISSN: 1540-7993. DOI: `10.1109/MSP.2009.56`.

[54] Birgit Penzenstadler. "What does Sustainability mean in and for Software Engineering?" In: *Proceedings of the 1st International Conference on ICT for Sustainability (ICT4S).* Jan. 2013.

[55] Patrycja Dybka. *CHAR and VARCHAR Data Types in Different Database Engines.* May 2016. URL: `https://www.vertabelo.com/blog/technical-articles/comparing-char-and-varchar-data-types-in-different-database-engines` (visited on 06/18/2019).

[56] Datical. *Change: 'createView'.* 2019. URL: `http://www.liquibase.org/documentation/changes/create_view.html` (visited on 04/24/2019).

[57] Datical. *Change: 'delete'*. 2019. URL: http://www.liquibase.org/documentation/changes/delete.html (visited on 04/24/2019).

[58] Datical. *Rolling Back ChangeSets*. 2019. URL: https://www.liquibase.org/documentation/rollback.html (visited on 04/24/2019).

[59] Scott Chacon and Ben Straub. *Pro Git*. 2nd. Berkely, CA, USA: Apress, 2014. ISBN: 978-1484200773.

[60] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. "Inferring versioned schemas from NoSQL databases and its applications." In: *International Conference on Conceptual Modeling*. Springer. 2015, pp. 467–480.

[61] Datical. *Maven Liquibase Plugin*. 2019. URL: http://www.liquibase.org/documentation/maven/index.html (visited on 04/18/2019).

[62] *Java JDBC API*. 2019. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/ (visited on 06/12/2019).

[63] Michael Neale. *Parallelism and Distributed Builds with Jenkins*. Sept. 2015. URL: https://www.cloudbees.com/blog/parallelism-and-distributed-builds-jenkins (visited on 06/18/2019).

[64] Theo Combe, Antony Martin, and Roberto Di Pietro. "To Docker or Not to Docker: A Security Perspective." In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[65] Josh Lerner and Jean Tirole. "Some simple economics of open source." In: *The journal of industrial economics* 50.2 (2002), pp. 197–234.

TRITA-EECS-EX-2019:470