UNIVERSITY OF TWENTE.

# ZERO-DOWNTIME SCHEMA CHANGES

PDENG THESIS

**Jorryt-Jan Dijkstra**

JJDIJKSTRA1@GMAIL.COM

# Zero-Downtime Schema Changes

## at a Financial Institution

# UNIVERSITY OF TWENTE.

## Zero-Downtime Schema Changes

PDENG THESIS

to obtain the degree of
Professional Doctorate in Engineering (PDEng)
at the University of Twente,
on the authority of the rector magnificus,
prof. dr. ir. A. Veldkamp,
on account of the decision of the graduation committee
to be defended
on Wednesday 1 September 2021 at 13.00h

by

**Jorryt-Jan Dijkstra**

born in Brummen, The Netherlands

**Graduation Committee:**

**Chairman and Director of PDEng Program**

| | |
|---|---|
| prof. dr. ir. D. J. Schipper | University of Twente |

**Supervisors**

| | |
|---|---|
| prof. dr. ir. A. Rensink | University of Twente |
| dr. ir. M. van Keulen | University of Twente |

**Members**

| | |
|---|---|
| dr. ir. A. J. J. Braaksma | University of Twente |
| J. W. Bosman | FI |

This dissertation has been approved by:

## Supervisors

prof. dr. ir. A. Rensink
dr. ir. M. van Keulen

# Abstract

Financial products have evolved to digital commodities. With Payment Services Directive (PSD2), banks and payment proviers are opening up their digital services to third party providers. Competition is growing as Bigtechs step into banking and Fintechs are quickly gaining ground. With the growth of competition and customer expectations shifting, it is vital to rapidly respond and adapt software. The financial institution where this project has been conducted, has the strategy to embrace this shift and adapt, such that changes are delivered faster and more often.

One of the expectations is 24/7 availability of digital services. To adapt faster to customer's needs and gather feedback more frequently, it is key to release new software versions more often. Releasing new software versions whilst staying available is challenging. Availability can only be guaranteed if two software versions (old and new) shortly live together whenever a new release takes place. Without this overlap, there would be a time window in which customers would face downtime.

The goal of this project is to mitigate downtime for customers, such that availability expectations can be met whilst releasing often. The project took place in the Digital Channels department, which facilitates a portal and mobile application to provide several financial products to large corporate clients. The project started by identifying technical challenges that affect availability at the time of a software release.

The challenge that this project addresses entails database schema changes, which might block queries or break the schema of the existing application. Little effort in the department has been spent on overcoming this challenge and the industry does not have a clear-cut solution for this. Therefore, the focus of the project has been nullifying downtime that is caused (or required) by schema changes. Criteria and requirements for the solution were gathered. These were combined with identified historical schema changes, such that the solution space could be explored. Primarily this led to a conceptual design that addresses how to propagate schema changes without downtime and how to test it for correctness and performance. Secondarily, the design has also been implemented into a solution that is directly applicable in the department.

The implementation consists of a plugin that works on top of the adopted

schema management tool (Liquibase) and a repository of patterns to cater for schema compatibility between two application versions. The plugin provides new schema changes by leveraging non-blocking equivalents of schema changes that otherwise would block queries of existing clients. The repository of patterns is designed using two approaches to deal with schema compatibility. Both of them adopt the Expand and Contract pattern, which entails postponing breaking schema changes until the to be deleted schema objects would not be in use any more by existing software versions. Next to that, it requires synchronization between old and new schema objects, such that data remains consistent between software versions. The first approach handles the expansion and synchronization both in the table close to the original data (referred to as in-place). The second approach creates a full table copy and atomically swaps the to be changed table. The first approach requires less space; no manual intervention, and fits the department tooling best. The second approach requires more space; manual intervention, and might have short perceived downtime during the atomic swap. As the first approach works in-place and the second approach operates on a full table copy, their risks are also different.

Recommendations such that the department can adopt the design and the example implementations are presented. Finally, information on how to extend and test both the design and implementation to support new types of changes has been included.

# Contents

# Glossary

**ACID** ACID is the abbreviation for Atomicity; Consistency; Isolation and Durability, which describe the safety guarantees of database transactions

**API** API is the abbreviation for Application Programming Interface, which is an interface that defines interactions between multiple software applications

**CD** CD is the abbreviation for Continuous Delivery

**ChangeLog** A Liquibase type that aggregates one or multiple groups of schema changes, often the root element of a tree structured file

**ChangeSet** A Liquibase type that contains one or more schema changes, which is intended to group a logical set of changes together

**CI** CI is the abbreviation for Continuous Integration

**CLI** CLI is the abbreviation for Command Line Interface

**Continuous Delivery** Continuous Delivery takes Continuous Integration one step further by always having the application in a deployable state

**Continuous Integration** Continuous Integration entails the practice of rebuilding and testing the application every time a developer commits a change to the version control system

**Database** A database is a collection of related data

**DBMS** A Database Management System is a computerized system that enables users to create and maintain a database

**DCL** DCL is the abbreviation for Data Control Language, which is a language that is used to control access and permissions to database objects

**DDL** DDL is the abbreviation for Data Definition Language, which is the language that allows a database administrator to define the database structure, schema and subschema

**Deployment** A complex post-production process that consists in making software available for use and then keeping it operational

**DevOps** DevOps is a set of practices that combines software development and IT operations

**DML** DML is the abbreviation for Data Manipulation Language, which is the language that facilitates to insert, update, delete and retrieve data from database tables

**FI** FI abbreviates financial institution, which is anonimized to make this thesis public

**JPA** JPA is the abbreviation for Jakarta Persistence API, which is a Java API to facilitate persistence

**Lead Time** Lead time resembles the time of the code commit (the change) and the moment of deployment to production

**Liquibase** An open-source database schema change management solution which enables you to manage revisions of your schema changes easily

**Meta-Data DDL** DDL statements that can be effectuated by only changing the definition of the table itself and not by inspecting/changing the underlying data

**Microservice** A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication

**Mixed-State** The phenomenon where the schema is compatible with two or more application versions

**MTTR** MTTR abbreviates Mean Time to Restore, which is entails the mean time it takes to restore service after an incident

**Online DDL** Online DDL provides building blocks to change the database schema real-time whilst still being able to serve clients and change table contents by using little to no-locking

**PAM** Party and Agreement Management

**PBT** PBT is the abbreviation for Property-Based Testing

**Property-Based Testing** Property-Based Testing is a novel approach to software testing, where the tester only needs to specify the generic structure of valid inputs to the program under test, along with a number of properties which are expected to hold for every valid input

**RDBMS** A DBMS for relational databases

**Relational Database** A relational database is a database built according to the relational model

**Relational Model** The relational model organizes data into relations (called tables in relational databases), where each relation is an unordered collection of tuples (called rows in relational databases)

**Schema** A schema is the logical structure of the data in a database

**SQL** SQL is the Structured Query Language that can be segregated in 4 sublanguages: DDL, DML, DCL and TCL

**TCL** TCL is the abbreviation for Transaction Control Language, which is a language to manage DML changes in logical groups

**TPM** TPM is the abbreviation for Transactions Per Minute

**Transaction** A transaction is a mechanism to group several reads and writes into a logical unit, which either succeeds (called commits) or fails (called aborts or rollbacks)

# 1 Introduction

This document serves as a final deliverable of the PDEng project of Jorryt-Jan Dijkstra. The main goal of a PDEng project is to deliver a relevant design for a company, in which theory and practice come together. The project has been executed at a large financial institution (further referred to as the FI) in 2020 and 2021. The purpose of this department is to provide a unified platform for large corporate customers, where corporate products are exposed through a single portal and mobile application.

The project focused on eliminating downtime for software deployments of database intensive applications. The aim of this document is to outline the problem that the design solves and to sketch the importance of solving the problem. Furthermore, it presents the design, how it is validated, how it can be adopted and what future work is left. The document is written for an audience that has some familiarity with software engineering, although a lot of the concepts are explained throughout the chapters. The Glossary has been adopted to lookup recurring concepts.

Firstly, the Background of the project will be discussed. The information provided in the background provides a solid understanding for the Problem chapter. The chapter is followed up by a description of the current Technical Context. The subsequent chapter Change Analysis presents department data and an analysis. Afterwards, the Solution Space for this problem will be discussed by taking the analysis into account and investigating several known strategies to (partially) solve the problem. The Solution chapter then describes the proposed solution. The Testing chapter follows up and tells about the testing strategy and results. The subsequent Conclusion chapter reflects on the solution, the test results and the initial requirements. Finally, the document is wrapped up with Recommendations on how to proceed with the designed solution in the Digital Channels Department.

# 2 Background



This chapter will describe the background of the project. It illustrates the business context as well as the technical context from a high-over perspective. The presented details in this chapter comprise enough details to formulate the Problem.

## 2.1 Business Context

Financial products have evolved from manual labor to digital commodities. With Payment Services Directive (PSD2 [1]), banks and payment providers are opening up their digital services to third party providers [2]. It is vital to constantly adapt to customer needs as well as respond to the major competition. This necessity gets confirmed in the annual report by the FI[1].

### 2.1.1 Department

The Digital Channels department (where this project is conducted) aims to facilitate product teams to adapt and expose their digital products to corporate clients. It provides a set of standard services (i.e., contract management) and tooling such that product teams can develop products that integrate into the portal. Furthermore, this department is key for the strategy of digitizing the FI. High service availability is an important requirement, due to the department providing the single holistic platform that exposes products to customers. Maintenance and product evolution becomes challenging as clients are distributed around the globe, such that continuous service availability in different time zones is expected.
In the rest of this document, "the department" will refer to the mentioned Digital Channels department.

[1] This citation has been removed due to the anonimization of this report

## 2.2 Way of Working & Paradigms



This section first elaborates on the way of working within the FI, which radically changed in 2015. Furthermore, it discusses paradigms that are used throughout the FI as well as how they relate to the way of working.

### 2.2.1 Agile

In 2015, the FI embarked in a transformation to an Agile way of working. Agile is a term that entails a few core principles to software development. These core principles are defined by the Agile Manifesto [3]:

**A1** Individuals and interactions over processes and tools

**A2** Working software over comprehensive documentation

**A3** Customer collaboration over contract negotiation

**A4** Responding to change over following a plan

The core principles emphasize new pillars that demonstrate the contrast with the traditional way of working, where projects were approached using the waterfall model (e.g., strictly following an upfront created plan) [4, p. 5].

In an interview that reflects on the transformation, the FI managers stated that adopting Agile has improved time to market, boosted employee engagement and increased productivity[1]. One of the managers states that Continuous Delivery and DevOps also played crucial roles in the transformation. These terms will be further clarified in the next subsections.

### 2.2.2 Continuous Delivery

Continuous Delivery (CD) is a concept that takes the Continuous Integration (CI) concept one step further. Humble et al. define CI as follows [5, p. 55]:

[1] This citation has been removed due to the anonimization of this report

> "Continuous Integration entails the practice of rebuilding and testing the application every time a developer commits a change to the version control system."



If the build or test process fails, the development team starts to fix the problem(s) immediately. The goal of CI is to have the software in a working state all the time. Continuous Delivery is the practice to always have the application in a deployable state [6]. This is done by developing so-called deployment pipelines that take the integrated code (i.e., Continuous Integrated code) into production-grade software [5, p. 4]. Such a pipeline looks as follows [5, p. 4]:


Figure 2.1: Software Deployment Pipeline

The deployment pipeline clearly demonstrates the difference between CI and CD, where CI only consists of the steps in the commit stage.

The deployment pipeline serves the following key principles [5, p. 4]:

**CD1** Make the full build-, deploy-, test- and release process visible to aid collaboration

**CD2** Improve feedback to identify problems as early as possible

**CD3** Enable teams to deploy and release any version of their software to any environment in an automated fashion

By reflecting on these principles, it becomes visible that there is a clear relationship between Agile and CD:

- **CD1** relates to **A1**: Making the process visible to aid collaboration
- **CD2** relates to **A2**: Providing feedback to identify and fix problems that hamper working software
- **CD3** relates to **A2**: Guaranteeing working software always being ready to deploy

- **CD3** relates to **A4**: Minimizing manual effort such that adapting to change is easy



Given the mentioned relationships, many consider CD a natural fit with Agile software development. In 2021, it is a common practice to have deployment pipelines for any software that is under development.

### 2.2.3 DevOps

DevOps is a set of practices that combines software development and IT operations. There is a lot of debate on the definition and set of practices. This document will use the definition by Bass et al. They define DevOps as follows [7]:

> "DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."

The provided definition can be considered an addendum to the deployment pipeline.

Lwakatare et al. have put effort into formalizing what DevOps constitutes. They found 4 dimensions [8]:

- Collaboration: Collaboration is enforced through information sharing, broadening of skill-sets and shifting responsibilities between the two teams as well as instilling a sense of shared responsibility
- Automation: In order to keep up with the pace of Agile software development and Continuous Integration practices, operations processes need to be flexible, repeatable and fast by eliminating manual processes
- Measurement: The ability to measure the development process by incorporating different metrics that will help increase efficiency in product development
- Monitoring: Emphasizing collaboration between developers and operations so that the systems are designed to expose relevant information

In the department, these practices are not as widely adopted as the Agile Way of Working and Continuous Delivery yet. Currently, the department is in the transition to adopt DevOps principles. A recent example of the transition to DevOps is that the department is migrating to a cloud platform, in which the software engineers develop the deployment pipeline and also take care of having their software operate in the cloud. In traditional practice, development and operations were separate tasks, such that the software adaptation to be cloud-ready would be developed by the software engineers and the operational migration would be facilitated by operations engineers.



## Research

Forsgren et al. conducted a broad research on the adoption of DevOps, where differentiation in software delivery performance has been related to four key metrics:

- Deployment Frequency: High performers had 46 times more software deployments than low performers [9, p. 10]. Arcangeli et al. define software deployments as follows [10]:

  > "Software deployment is a complex post-production process that consists in making software available for use and then keeping it operational."

  In practice, common activities in the software deployment process are backing up data, copying compiled binaries to a server, execute schema changes and starting the application.
  Software deployment and software release are often used interchangeably. In this document, a software deployment to the production system is considered a software release.

- Lead Time for Changes: lead time resembles the time of the code commit (the change) and the moment of deployment to production [9, p. 10]. The identified high performers had significantly faster lead times than the low performers (440 times) [9, p. 10].

- Mean Time to Restore (MTTR): The MTTR is the mean time it takes to restore service when an incident occurs. The high performers had 170 times faster service recovery than the low performers [9, p. 10].

- Change Failure Rate: The change failure rate embodies the percentage of changes that subsequently required a fix or/and resulted in degraded service. The high performers had a 5 times lower failure rate than the low performers [9, p. 10].



Forsgren et al. found that a high software delivery performance has positive impact on both the organizational and non-commercial performance of companies. For instance: high performers were twice as likely to exceed their goals on profitability, market share and productivity than low performers [9, p. 24].

## 2.3 Aspirations of the FI

In the earlier mentioned interview with the FI management, one of the managers stated that releasing software much more frequently is an aspiration within the FI[1]. This is a logical consequence of having to respond to change rapidly, as it reduces the delay between implementing the change and serving the change to customers. It also strongly relates to the DevOps definition provided by Bass et al., where reducing the time between introducing a change and serving it to customers should be optimized. Releasing more often induces a shorter feedback loop with customers (facilitating **A3** and **A4**).

It is clear that in order to be a high performer, Continuous Delivery and DevOps principles have to be strongly adopted. Any manual intervention or labor-intensive process hampers the short feedback loop and increases lead times. Besides, manual actions are likely to increase the failure rate and the MTTR.

# 3 Problem



Due to the strategy shift and urge to become a high performer, increasing deployment frequency and reducing lead times both became objectives for the department in 2021. For each product team (often referred to as squads), the target is to deploy at least once a month. Another target is to have less than a month of lead time for 80% of the squads. Currently, the target is not being met.
This chapter first discusses what currently stops lead times from becoming shorter. Afterwards, one of the underlying causes is extracted and further explored to establish a problem statement. The problem statement leads to an overview of challenges to be overcome. Finally, the requirements are presented, which also embody the discussed challenges.

## 3.1 Problem Identification

To get a better understanding of underlying aspects that negatively affect lead times, an assessment has been made as part of this project. Input from the Agile coach and several managers led to the identification of three aspects that underpin long lead times in the department:

- Scrum stories were often too large, which caused work to be stacked for a longer period, before it could be released. This department was quite late to go through the Agile transformation (2019), which is a plausible explanation for this problem to occur. Teams learn to better slice and estimate their work over time, and therefore lead times are improving as commits that contain business value become smaller. Metrics for the amount of work done versus the planned work, as well as lead times are actively monitored. Both of them show a continuous improvement over time since 2020.
- Each production release requires a vast amount of release evidence for reasons such as quality assurance, audits and regulatory requirements. This evidence mainly consists of proof of adhering to the quality criteria of the process and tools (such as security and test reports). Gathering this evidence takes a lot of time (estimated to

be 1 working day per release) and effort, due to the large number of regulations and risks that have to be accounted for. Currently, an automated way of gathering evidence in the deployment pipeline is being adopted throughout the department. Some squads already have this automation implemented in their pipeline and report positive effects on the lead times.



- Deployments impose downtime of services. A service has to be taken offline in order to upgrade it (albeit for a short period of time). Downtime requires planning and communication with customers, as it will prevent customers from using the service. It also requires coordination amongst teams that have interdependent software products. The effect on customers and other teams makes it is currently not desirable to release more often.

The first two aspects underpinning longer lead times show clear progression. Imposed downtime also improved over time (as displayed in Figure 3.1), but does not directly facilitate a higher deployment frequency. This is because any downtime (no matter how short) could impede customers. One can imagine that having downtime often (albeit short) is undesirable for the portal and that having downtime once per month (but longer) might be easier for customers to deal with.

Next to increasing the deployment frequency, another important objective for the department is to minimize planned downtime with a target of 20 minutes per month. Planned downtime consists of deployment activities as well as (foreseen) maintenance activities. Reducing downtime for deployments is an important aspect to meet this target, such that there is still time for maintenance as soon as the deployment frequency increases.

Since June 2015, data has been collected on the downtime of the portal as a whole. Deployments and maintenance of core applications that address concerns that are required by other portal applications, require the whole portal to be offline. This is to prevent dependent applications from not being able to function correctly. An example of such core application is the Party and Agreement Management application (PAM). The mentioned collected downtime data of the portal is displayed in the following figure:

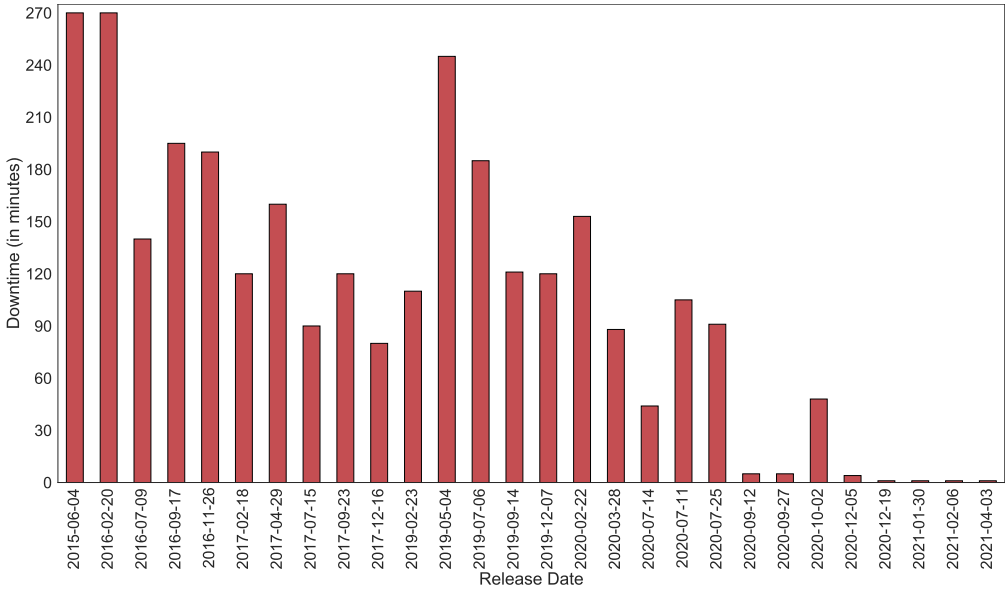



Figure 3.1: Downtime of the portal on a per-release basis (date: May, 10th 2021)

The chart displays the improvements, where most recent releases ended up with 1 minute of downtime each. This is due to a migration to different deployment software. The chart also depicts that the deployment frequency is low (at least for core applications), which remains limited as long as downtime is necessary for deployments. Even though the downtime has been mitigated, a deployment still requires communication with customers as the portal serves customers 24/7 across the globe. If 1 minute downtime would be the norm, more frequent deployments would lead to short and frequent service interruptions. A high deployment frequency can only be attained, when customers would not experience any service interruption for the majority of the deployments.

## 3.2 Problem Exploration

This section further discusses downtime as a result of software deployments. Deploying a new version without imposed downtime requires (at least) two versions of the same application to operate simultaneously. In other words: it is inevitable that at least one version of an application has to be able to serve customers at any given time. If this would not be the case, there would be (a short) service interruption. According to Nygard there are two key elements that need to be tackled to mitigate downtime

[11, p. 250]:

1. Protocol versions
2. Schema changes

Both of these elements will be discussed in the subsequent sections.



### 3.2.1 Protocol Versions

The protocols and APIs of newly deployed application versions may be incompatible with surrounding services. Their changes can be breaking, which dependent services would not be accustomed to at the time of the deployment.
There are plenty of ideas about versioning protocols [12, p. 128], such as: strictly adhering to the Semantic Versioning convention and interface evolution patterns as described by Lubke et al [13, p. 2]. The idea behind Semantic Versioning is that software- and protocol version numbers reflect whether changes are breaking or not. In a large scale analysis of 22K open source projects by Raemakers et al., about one third of the releases contained breaking protocol changes [14, p. 244]. The paper showed that in 2014 many developers did not comply to the convention of Semantic Versioning yet. These rules have started to become more of an engineering standard the last few years and are part of the internal conventions of the FI.

### 3.2.2 Schema Changes

The problem of schema changes is only relevant when applications deal with a database. This is because a schema is an aspect of databases. A database is a collection of related data [15, p. 4]. These collections are often managed by a Database Management System (DBMS), which is a computerized system that enables users to create and maintain a database [15, p. 4]. Schema changes are changes made to the logical structure of the data in a database [16, p. 1]. Many of the (core) applications in the department use a database, of which most of them are relational databases. A relational database is built according to the relational model. The relational model organizes data into relations (called tables in relational

databases), where each relation is an unordered collection of tuples (called rows in relational databases) [12, p. 28].
Currently, there are no efforts in the department to be able to deal with schema changes whilst allowing multiple application versions to be running at the same time.



## 3.3 Problem Statement

Dealing with schema changes lags behind other efforts to reduce (or overcome) downtime. Engineers in the department have little knowledge about databases in general and do not have the time to investigate related solutions. Solving the above problem will facilitate increasing the deployment frequency and decreasing lead times, as a lot of the department-built software relies on relational databases. Requirements of the applications often change, which also require the schemas to evolve. This project aims to close the gap such that schema changes without downtime are on par with other efforts that aid in increasing deployment frequency. This leads to the following problem statement:

> "Mitigating downtime of application deployments due to schema changes in the current context."

What is meant with the current context is the department as well as the technical context in which this project is being executed.
The next section elaborates how schema changes are achieved and what definition is used throughout this project. Afterwards, challenges that need to be overcome will be discussed. The last section presents the requirements that are distilled from the challenges and the working context.

## 3.4 Definition of Schema Changes

Schema changes are effectuated through the execution of statements that are written in the Data Definition Language (DDL). DDL is "the language that allows a database administrator to define the database structure, schema and subschema" [17, p. 42]. A schema change is considered to be a change that is issued through a single DDL statement. A database refactoring is defined by Ambler et al [18, p. 3]:

> "A database refactoring is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. You could refactor either structural aspects of your database schema such as table and view definitions or functional aspects such as stored procedures and triggers. When you refactor your database schema, not only must you rework the schema itself, but also the external systems, such as business applications or data extracts, which are coupled to your scheme."



Schema changes and refactorings are very related and can be used interchangeably in this document, as both entail changes to a schema.

## 3.5 Challenges

There are three challenges to zero-downtime application deployments which require schema changes [19] [20]:

**CH1** **Schema compatibility between both versions**: The old application has no notion of any new version at the deployment time. As soon as a new version gets deployed, it may contain schema changes that break the old version of the application.

**CH2** **Schema changes can be blocking in nature**: Changes to the structure of the schema may block queries during effectuating the changes. This may hamper the already running version from executing queries.

**CH3** **Preserving foreign key relationships**: Existing tools have no or insufficient support for foreign keys. In the case of insufficient support, there is often a time window where violations may occur due to temporary disabling the constraint.

The phenomenon where the database facilitates multiple application version schemas is referred to as the Mixed-State [19].

## 3.6 Requirements

This section outlines the requirements for a solution to achieve schema changes without requiring downtime. The first part is context based and relates to the way of working in the department and the technical stack (**R1** - **R4**). The second part of the requirements has been taken from the research by de Jong et al., which primarily addresses the mentioned challenges. These are the requirements from **R5** onwards [20].



**R1** **Integration**: Fits the current stack and software engineering process.

**R2** **Detachable**: The design can be omitted at any point and the way of working can continue where it left off.

**R3** **Transparent**: The effects of the design should be fully transparent to the software engineer prior to utilizing it.

**R4** **Generalizable**: The design is generic in a sense that it is not specifically tied to the current RDBMS and can be adapted to support a different RDBMS.

**R5** **Non-Blocking Schema Changes**: Changing the schema should not block queries issued by any database client.

**R6** **Schema Changesets**: It should be possible to make several non-trivial changes to the database schema in one go. This prevents software engineers from having to develop and deploy intermediate versions of the web service.

**R7** **Concurrently Active Schemas**: Multiple database schemas should be able to be "active" at the same time, to ensure that different versions of the web service can access the data stored in the database according to their own chosen schema. This avoids putting restrictions on the method of deployment when upgrading the web service.

**R8** **Referential Integrity**: The solution must support both the migration and evolution of foreign key constraints during both normal use and while evolving the database schema. These constraints should be enforced at all times.

**R9** **Schema Isolation**: Any changes made to the database schema should be isolated from the database clients. In other words, no client should see any database schema other than the version it relies on.

**R10** **Non-Invasiveness**: Any integration with the application should require as little change to the source code as possible.



**R11** **Resilience**: The solution must ensure that the data stored in the database always remains in a consistent state. In other words, when the migration fails, it must be possible to roll back the changes and return to a consistent state without affecting the database clients.

# 4 Technical Context

This chapter elaborates on what the Digital Channels environment looks like from a technical perspective. This is important as the first few requirements entail integrating easily and refrain from completely changing the architecture and technical process. It also provides a starting point to explore the Solution Space.
The first section will talk about the recent architectural shift towards a microservices architecture. The second section elaborates on the technologies that are used for typical applications. How schema management is done will be discussed separately, as it is core to the project.



## 4.1 Application Architecture & Technology

In the department, there are different squads that are responsible for one or more applications. Before 2019, each release was done in a big-bang manner, in which all applications that comprise the portal would be updated all at once. This is closely related to monolithic application deployment and it is common for Service-Oriented Architectures [21, p. 35]. Big-bang releases typically require a lot of communication amongst teams and logically induce moving at the pace of the component with the slowest deployment cycle.

### 4.1.1 Shift to Microservices

The implementation of Agile in the department went hand in hand with moving towards more autonomy amongst teams. This is where microservices come into play:

> "A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices." [22, p. 6]

The essence of a microservice architecture is to have independent logical components that can move at their own pace and are able to communicate (where necessary) with other microservices. The adoption of a microservices architecture leads towards a higher service autonomy and decoupling [21, p. 29]. Applications get released separately, although sometimes coordination is still necessary for cross-cutting concerns. Shifting towards microservices from traditional systems requires decomposition of the existing systems. There are several strategies to decompose existing systems to microservices, such as using Domain-Driven Design techniques [22, p. 64]. These decomposed microservices should be able to evolve and be deployed independently [23, p. 14]. In the department, each team is responsible for one or more microservices. Some larger applications are still in the process of being decomposed to microservices.



## Relationship to Databases

Microservices need to be well-understood, because of their relationship with databases. It is a best practice to have each microservice to be responsible for its own database [23, p. 13] [21, p. 31]. All microservices in the department that require a database, have their own database. This is often called a Single Application Database environment [18, p. 15]. Richardson states the following about such an environment [23, p. 12]:

> "At development time, developers can change a service's schema without having to coordinate with developers working on other services. At runtime, the services are isolated from each other — for example, one service will never be blocked because another service holds a database lock."

This is where the discussed protocol versioning plays a role: whenever data needs to be retrieved from another microservice's database, it has to be done through message-based communication using a protocol.

### 4.1.2 Database Technology

Most of the microservices in the department that need data storage use a Relational Database Management System (RDBMS). The RDBMS that is used in the department is Oracle Enterprise Edition 19c. This section discusses the languages and properties of RDBMSes.

#### Languages



All RDBMSes support the Structured Query Language (SQL) [17, p. 244]. SQL can be segregated in different languages [17, p. 245]:

**Data Manipulation Language** The Data Manipulation Language (DML) is a language that facilitates to insert, update, delete and retrieve data from database tables.

**Data Definition Language** The Data Definition Language (DDL) is a language that is used to create and modify database objects such as tables, indexes, views and triggers et cetera.

**Transaction Control Language** The Transaction Control Language (TCL) is a language that is used to manage DML changes. It provides commands to group those changes into logical transactions.

**Data Control Language** The Data Control Language (DCL) is a language that is used to control access and permissions to database objects.

#### ACID Properties

Oracle is an ACID compliant database. ACID describes the safety guarantees of database transactions [12, p. 223]. A transaction is a mechanism to group several reads and writes into a logical unit, which either succeeds (called commits) or fails (called aborts or rollbacks) [12, p. 222]. ACID abbreviates: Atomicity, Consistency, Isolation and Durability. Their meanings will be explained briefly, using the exact definitions by Coronel et al [17, p. 487].

**Atomicity**
Atomicity requires that all operations (SQL requests) of a transaction be completed; if not, the transaction is aborted. If a transaction $T_1$ has four SQL requests, all four requests must be successfully completed; otherwise, the entire transaction is aborted. In other words, a transaction is treated as a single, indivisible, logical unit of work.

**Consistency**
Consistency indicates the permanence of the database's consistent state. A transaction takes a database from one consistent state to another. When a transaction is completed, the database must be in a consistent state. If any of the transaction parts violates an integrity constraint, the entire transaction is aborted.



The idea of consistency is that certain statements about data (known as invariants) must always be true, for instance: the credits and debits of a banking system must always be balanced [12, p. 225]. Consistency is a property of the application and is not up to the database alone (in contrast to the other properties) [12, p. 225].

**Isolation**
Isolation means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. In other words, if transaction $T_1$ is being executed and is using the data item `X`, that data item cannot be accessed by any other transaction ($T_2 \ldots T_n$) until $T_1$ ends. This property is particularly useful in multiuser database environments, because several users can access and update the database at the same time.

There are different levels of isolation, whereas the conceptual ideal is *serializability*. Whenever transactions are committed, the result is the same as if they had run serially (one after another), even though in reality they may have run concurrently [12, p. 225]. In practice, weaker isolation levels are often used, due to the performance penalty of serializability [12, p. 226]. The SQL standard defines different isolation levels that typically have different trade-off characteristics in terms of performance and predictability when transactions run concurrently.

**Durability**
Durability ensures that once transaction changes are done and committed, they cannot be undone or lost, even in the event of a system failure. To provide a durability guarantee in a replicated database, a database has

to wait until replications are complete before reporting a transaction as successfully committed [12, p. 226].

## 4.2 Languages, Frameworks & Libraries

In Digital Channels, microservice projects are developed in a language on top of the Java Virtual Machine (JVM). Most of them are written in Java, whilst newer projects are developed in Kotlin (which is interoperable with Java [24]).
Basic building blocks to engineer microservices are provided by the Spring Boot framework, which uses Liquibase for database schema management. For persistency and relational data management, the Jakarta Persistence API (formerly Java Persistence API) is used throughout the projects (abbreviated as JPA). The Hibernate framework is Spring Boot's default JPA implementation. The microservices that require a relational database are using Oracle Enterprise Edition 19c. Unit and integration tests that do not require Oracle explicitly use H2 or HSQLDB as a lightweight alternative database.
It is worthwhile to understand Liquibase to a further extent, as schema changes are core to this project.



### 4.2.1 Schema Management using Liquibase

Liquibase is an open-source database schema change management solution, which enables managing revisions of schema changes easily [25]. Spring Boot provides a default integration for Liquibase. Both the code and schema changes are managed in the same Spring Boot project and evolve together. How this is administered is best visualized using the directory structure of such a project:

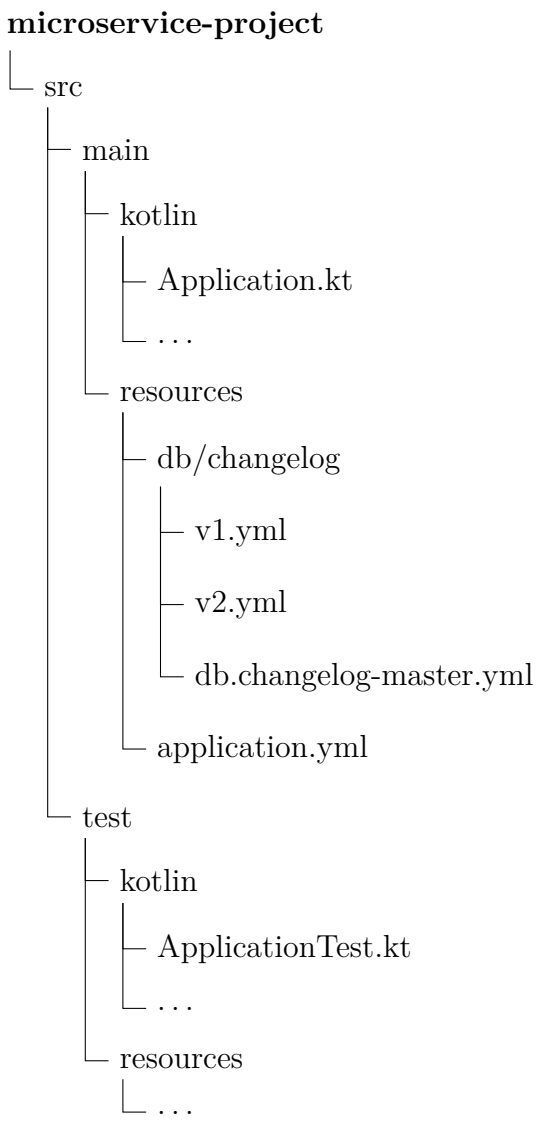



Figure 4.1: Example structure of a Spring Boot project that uses Liquibase

The several database changes are aggregated in the `db.changelog-master.yml` file, which captures the Liquibase `ChangeLog` type. The following Liquibase types are important to grasp:

- A `ChangeLog` is the aggregation of grouped schema changes (which are called `ChangeSets`). It is also important to note that a `ChangeLog` can be recursive (i.e., it can refer to other files that contain a `ChangeLog`).
- A `ChangeSet` contains one or more changes and is intended to group a logical set of changes together. Each `ChangeSet` is running in its own database transaction.
- A `Change` captures something that is directly translatable to a SQL statement. In Liquibase terms, a change does per definition relate to the earlier defined schema change. A Liquibase `Change` can encompass DML and DDL changes. For instance: `insert` is also a Liquibase change.



It is considered a best practice to keep a single schema change per `ChangeSet`. Many databases (including Oracle) auto-commit DDL statements. Auto-commit is a construct where each DDL statement commits the current transaction and then runs the DDL statement in its own transaction [26].
When a `ChangeSet` contains multiple changes including a DDL statement, it may be that the database ends up in an unexpected state. This is due to auto-commit violating the design choice of having one transaction per `ChangeSet` [27].

## Capturing Changes

Now that the relationships between ChangeLogs, ChangeSets and changes are clear, it is useful to present a typical change. The following code corresponds to `v1.yml` in Figure 4.1:

```yaml
databaseChangeLog:
  - changeSet:
      id: initial
      author: Jorryt
      comment: Initial schema
      changes:
        - createTable:
            tableName: customer
            columns:
              - column:
                  name: id
                  type: int
                  autoIncrement: true
                  generationType: BY DEFAULT
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: name
                  type: varchar(512)
                  constraints:
                    nullable: false
```

The presented code entails creating a table named *customer* with two non-nullable columns, in which one of them is a primary key.

## Version Tracking

Liquibase uses two administration tables to keep track of previously propagated schema changes. These two tables reside in the same database as the application schema. Figure 4.2 visualizes the schema after migrating the changes of `v1.yml`. It also includes the administration tables.

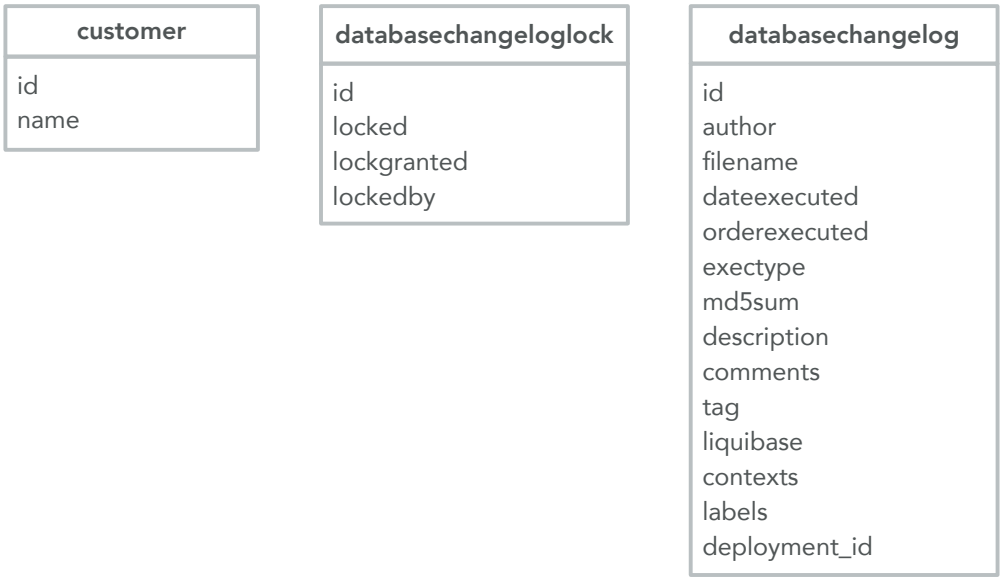



Figure 4.2: A project specific table and Liquibase administration tables

The first Liquibase administration table is named `databasechangeloglock`. This table is solely used to construct a temporary lock to make sure that only one Liquibase instance can propagate schema changes at once. The other table named `databasechangelog` keeps track of all ChangeSets that have been propagated. ChangeSets are uniquely identified using the triple: $\langle Path, Id, Author \rangle$. In the outlined example, this triple would correspond to $\langle v1.yml, initial, Jorryt \rangle$.
Besides, it is important that previously applied changes do not get modified over time. Therefore, Liquibase normalizes each ChangeSet to compute a checksum. The checksum aids in making sure that the semantics remain the same over time for each ChangeSet. The normalization process removes noise that has nothing to do with the semantics, such as comments and spaces. Schema propagation will fail as soon as Liquibase detects that a previously applied ChangeSet (identified by the identity triple) has a different checksum than what was administered in the `databasechangelog` table.

### 4.2.2 Spring Boot Integration

The Spring Boot framework provides default integration with Liquibase. In a Spring Boot application, Liquibase will compute all schema changes that have to be propagated at startup time. It bases this on the project's master ChangeLog and checks that against the historical runs, which are logged in the `databasechangelog` table. The previously propagated changes will then be skipped and only the necessary changes will be executed. A Spring Boot application will be operational as soon as all the necessary schema migrations are successfully executed.



### 4.2.3 Benefits

There are a few key benefits to Liquibase as a schema management tool:

1. Couples source code and schema, such that they evolve together (related to the Single Application Database environment)
2. Archives and versions schema history
3. Provides the ability to easily swap DBMS (DBMS agnostic)[1]
4. Provides the ability to integrate in existing projects
5. Provides the ability to opt out any time
6. Automatically generates rollbacks for typical changes

A rollback in Liquibase is a change to revert another change. This only works out of the box for statements that do not delete data. For instance: dropping a column can not be rolled back.

[1] In case changes use widely adopted database constructs and do not contain raw SQL code

### Object Mapping

Applications that work with a database, require mapping from data to Java/Kotlin objects and vice versa. The Hibernate framework takes care of this mapping. An `entity` represents persistent data stored in a relational database [28]. How this mapping takes place will be omitted, as this is not relevant for the understanding of the project. Understanding how entities are captured, is important for **R9** and **R10**. The entity that corresponds to the table in `v1.yml` looks as follows in Kotlin syntax:



```
1  @Entity
2  data class Customer(
3    @Column
4    val name: String,
5
6    @Id
7    @GeneratedValue(strategy = GenerationType.IDENTITY)
8    val id: Long = 0
9  )
```

Whenever a type is suffixed with a question mark, it may be nullable (and thus also nullable in the database schema). There are no nullable columns in the presented example.

## 4.2.4 Architecture Diagram

This section presents the microservice application architecture of a typical Spring Boot application. Figure 4.3 displays how code and resources are organized within a single microservice application. The depiction is inspired by the picture and descriptions by Gos et al [29]. All the department's microservices comply to this architecture.

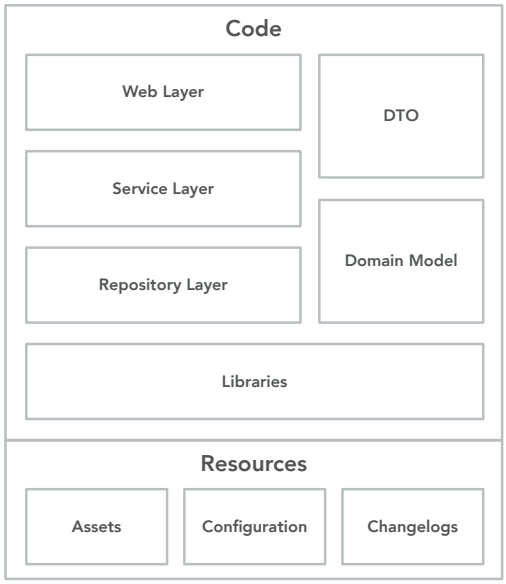



Figure 4.3: Application Architecture

Each of the depicted boxes will shortly be elaborated:

**Web Layer** implements the API and handles requests.

**Service Layer** implements business logic and communicates using DTO's and the Domain Model.

**Repository Layer** implements the communication with the database and persists and retrieves entities. A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection [30, p. 322].

**DTO (Data Transfer Objects)** contain data carrying objects that are received and sent. They are used to communicating with the Web- and Service Layer.

**Domain Model** implements the domain, which contains the entities that will be persisted and retrieved from the database. The domain is known both in the Service- and Repository Layer.

**Libraries** contain all the dependencies that the application is built on top of, including Spring Boot, Hibernate and Liquibase.

**Assets** contain all static data that need to be accessible for the application regardless of where the application resides (such as translations or images).

**Configuration** contains configuration required to run properly (such as settings for: database connections, logging, and security).



**Changelogs** contain the Liquibase ChangeLogs that will be loaded when starting the application.

The organization of resources is arbitrary and can be configured differently. All resources are available for the application at runtime.

# 5 Change Analysis

This chapter is devoted to getting insight in which schema migrations occur and with what frequency. From a theoretical perspective, there are many schema migrations that could occur. In practice, it might be that not all types of schema migrations need to be supported. Moreover, some changes likely occur more often than others. This serves as input in exploring the Solution Space.

Two approaches have been taken to acquire a view on realistic schema migrations. The first one entails an automated statistics collection of all historical Liquibase changes of multiple applications within the department. The second approach is the manual inspection of more than three years of change data of an important application.



## 5.1 Automated Results

One of the deliverables of the project is *lbstats*, which is a tool that gathers Liquibase statistics.

Table 5.1 presents types of changes on a per-project basis, aggregated by *lbstats*. In the table, project names are displayed horizontally and change types are displayed vertically (using the types from Liquibase Community Edition[1]).

[1] Liquibase Pro changes are not used in the department

| | aax | pam | cha | irs | srv | cpa | nfe | |
|---|---|---|---|---|---|---|---|---|
| **addColumn** | 6 | 0 | 0 | 10 | 0 | 0 | 0 | 16 |
| **addNotNullConstraint** | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 4 |
| **addPrimaryKey** | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 |
| **createIndex** | 8 | 0 | 0 | 6 | 0 | 0 | 0 | 14 |
| **createSequence** | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| **createTable** | 2 | 0 | 0 | 5 | 0 | 0 | 0 | 7 |
| **dropColumn** | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 5 |
| **dropIndex** | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 |
| **dropSequence** | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| **dropTable** | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 2 |
| **modifyDataType** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| **renameColumn** | 3 | 0 | 0 | 5 | 0 | 0 | 0 | 8 |
| **renameTable** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| **sql** | 1 | 0 | 0 | 8 | 0 | 0 | 0 | 9 |
| **sqlFile** | 0 | 5392 | 3437 | 1 | 5 | 1511 | 478 | 10824 |
| **tagDatabase** | 0 | 739 | 646 | 0 | 2 | 60 | 109 | 1556 |
| **update** | 0 | 0 | 0 | 2 | 0 | 0 | 3 | 5 |
| | | | | | | | | |
| $\mathbf{N}_{ChangeSets}$ | 10 | 6131 | 4083 | 38 | 7 | 1571 | 572 | 12412 |
| $\mathbf{N}_{RollbackableChangeSets}$ [1] | 7 | 1545 | 1353 | 24 | 4 | 111 | 267 | 9335 |
| $\mathbf{N}_{Changes}$ [2] | 31 | 6131 | 4083 | 50 | 7 | 1571 | 590 | 12463 |
| $\mathbf{N}_{RollbackableChanges}$ | 23 | 739 | 646 | 34 | 2 | 60 | 109 | 1613 |

Analysis Date: 27th of August 2020

[1] represents the number of ChangeSets that can be rolled back

[2] represents the number of changes that can be rolled back

Table 5.1: Frequency Analysis (automated) of types of changes per project

The table shows that many teams do not use typed changes in Liquibase. They use plain SQL instead. This is heritage from previously outsourcing applications, where teams put little effort in understanding Liquibase. Plain SQL negates some benefits that Liquibase provides (including the discussed ones):

- Database technologies can no longer be changed easily. This is because there are many SQL dialects that are specific to database products.
- Rollbacks are no longer generated automatically. This is because the type of change (and therefore the rollback) can not be inferred by Liquibase. To achieve the same effect, rollback statements would have to be specified manually.
- Risks are introduced. Manual rollback statements are error-prone and might not be the exact inverse of the specified change. Moreover, Liquibase does not require rollbacks to be specified. Therefore, this approach comes with the risk that changes can not be (fully) rolled back.
- Change data can no longer be analyzed automatically. This is because there are many SQL dialects and there are various ways of writing changes in SQL that are semantically equivalent.

Based on above findings, new guidelines and improvements have been set up and became a standard within the department. This is a serendipitous effect of this project.

## 5.2 Manual Findings

The largest application (with the most schema changes) is called Party and Agreement Management (PAM). To get a better understanding of the type of changes in PAM a manual analysis on its SQL code has been conducted. This was desirable due to the lack of typed changes (as noted in Table 5.1). Table 5.2 consists of schema changes since and including PAM version 3.0.0 (December 2018) up to March 2021. Each of the changes has been categorized using the same Liquibase types. Changes to PAM that were made for auditing are left out, because they are not about the structure of the application and were considered a temporary solution.

| | Frequency |
|---|---|
| **addColumn** | 79[1] |
| **addNotNullConstraint** | 0 |
| **addForeignKey** | 38 |
| **addPrimaryKey** | 34 |
| **addUniqueConstraint** | 32 |
| **createIndex** | 15 |
| **createSequence** | 37 |
| **createTable** | 37 |
| **dropColumn** | 3 |
| **dropForeignKey** | 3 |
| **dropIndex** | 2 |
| **dropNotNullConstraint** | 10 |
| **dropTable** | 4 |
| **dropUniqueConstraint** | 7 |
| **modifyDataType** | 2[2] |

[1] of which 2 are Non-Nullable
[2] both to resize the current data type

Table 5.2: Frequency Analysis (manual) of types of changes for PAM



Next to the displayed changes, there are some constructs that are not adopted in Liquibase but seem to be used:

- Adding permissions such that the database user which is used by the application has sufficient access at runtime.
- Populating tables with static data and data from other tables using `merge` statements (similar to what sometimes is called `upsert` statements).
- Updating a settings table (also using `merge` statements) such that new settings can be introduced and values of existing settings can be updated.
- Creating triggers that facilitate setting primary key values based on a sequence object.

`Merge` statements are generated from Excel sheets for each release. Moreover, data scripts are generated for each test and acceptance environment separately. Without fully understanding the context and without access to the databases, it is impossible to deduce what the actual data changes

for a release are. The ChangeLogs are not really used to capture incremental changes, and excessive use of `merge` statements lead to insufficient insights in the actual changes.
Even though the goal is to release more often and therefore have fewer changes per release, the grouping of changes per release has been investigated manually. Releases typically seem to have a combination of changes that revolve around new tables or columns. New application versions often consist of schema migrations that: create a table, add nullable columns to existing tables, create indexes and add constraints to the new table.

# 6 Solution Space

This chapter discusses possible directions to cater for zero-downtime schema migrations. The first section will explore several strategies. The subsequent section will discuss their applicability by taking the requirements into account.

## 6.1 Strategies

This section explores directions that Google suggests. Afterwards, several strategies that are closely related to the technical context will be discussed.



### 6.1.1 Google DevOps Strategies

Google has made a summary of possible zero-downtime schema migration strategies [31]. Note that some strategies can be complementary to each other, whilst some are alternatives (i.e., mutual exclusive). Each of them will be introduced and further elaborated:

1. **Use an online schema migration framework**: There are several tools such as *gh-ost* and *pt-online-schema-change* that allow for online schema migrations. They do so by creating so-called *ghost* tables. Those tables are essentially the version of the tables after the schema change. This allows for parallel operations on the original table whilst redefining the structure of the table. This requires duplication of existing data, as well as bidirectional synchronization between both the *ghost* table and the original table to aid for the parallel operations. Some tools facilitate the synchronization by triggers, whilst other tools achieve the same without triggers by replicating events [32]. After the copy process is done, the original table will be atomically swapped with the *ghost* table. The original table can be dropped afterwards.
2. **Decouple database changes and application changes**: This item talks about using the so called *Expand and Contract* pattern.

Fowler refers to this pattern as *Parallel Change* [33, p. 74], whilst Nygard refers to expansion and cleanup phases [11, p. 250]. Ambler et al. refer to a deprecation period [18, p. 15]. The pattern describes the phenomenon where two versions of an application could operate on the same schema version. Refactoring happens by first expanding the schema, such that the old application version remains compatible. As soon as no version longer relies on the old schema definition, the schema can be contracted (i.e., cleaned up).
We consider for example the refactoring scenario of renaming a column $A$ to $B$. Renaming the column would break the old version of the application (which corresponds to schema $V_1$). Instead, one could add a column $B$ and implement a bidirectional synchronization mechanism between $A$ and $B$. Updates and inserts then have to propagate data from $B$ to $A$, when $B$ is filled and vice versa when $A$ is filled. The schema in which both column $A$ and $B$ are present, is denoted as $V_1 \cup V_2$. This schema corresponds to the Expand-phase. Column $A$ and synchronization mechanisms can be dropped as soon as no running application relies on schema $V_1$. This is done in the so called Contract-phase.
The figure below depicts the different phases using the Expand and Contract pattern, where each circle corresponds to a schema version and each arrow to a deployment:

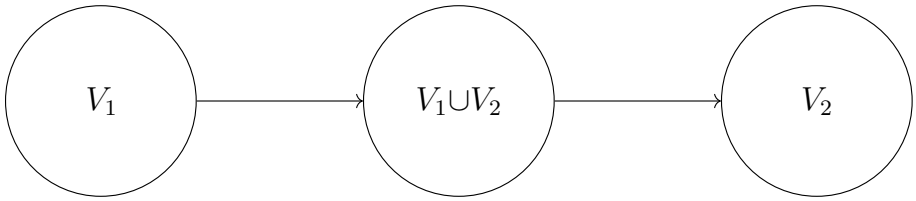



Figure 6.1: The different phases of Expand and Contract

Nygard provides a table of changes that should be easy to do due to their expanding nature. The potential challenges **CH2** and **CH3** are not mentioned nor addressed by Nygard. Table 6.1 gives an idea on what changes are schema compatible [11, p. 250]:

| Change/Refactoring |
|---|
| Add table |
| Add view |
| Add nullable column to a table |
| Add column with default value to a table |
| Add alias/synonym |
| Add new stored procedure |
| Add trigger* |
| Copy data into a new table or column |

* As long as triggers are non-conditional and do not throw errors [11, p. 250]

Table 6.1: Schema changes and their compatibility with previous versions

3. **Design and implement a data partitioning and archiving strategy**: Large tables are a major cause of schema changes taking so long. This strategy is about limiting the table size by implementing a relevant partitioning strategy. A proper archiving strategy comes with the same benefits. Decreasing the amount of data will likely shorten any migration strategy, especially if they need synchronization or locks.

4. **Use an event sourcing architecture**: Event sourcing is about recording all events that encompass (state) change [11, p. 315]. This often goes hand in hand with NoSQL database solutions. The state is then computed by stacking all events that have been stored (which is often called "playing back events"). In an event sourced architecture, events can be queued, which caters for database migrations to be able to take place simultaneously. The queue can be flushed to the database as soon as the migration is finished.

5. **Use a NoSQL solution**: NoSQL databases are an alternative to relational databases. In general, they do not adhere to the ACID principles and do not enforce a schema like a relational database does. The structure of the data is implicit and only gets interpreted by the application when the data is read. This is referred to as the schema-on-read paradigm [12, p. 39]. This is opposed to the schema-on-write paradigm (which relational databases adhere to), where the schema is explicit and enforced at all times [12, p. 40].

### 6.1.2 Contextual Strategies

In this subsection, key technologies for the Digital Channels department will be further investigated to see what directions and strategies could help in zero-downtime schema changes.

6. **Online DDL**: Oracle provides online alternatives to DDL. These have the same semantics on a schema level as their non-online counterparts, but yield limited to no locking. Online DDL provides building blocks to change the database schema real-time whilst still being able to serve clients and change table contents [34, p. 40].

7. **Edition-Based Redefinition**: Oracle provides Edition-Based Redefinition (*EBR*) functionality to facilitate the versioning of schema objects. Triggers, procedures and functions are examples of schema objects that can be versioned. Tables are not versioned, such that data and structure does not need synchronization amongst multiple versions. *Editioning Views* are views that proxy to an underlying table. For example, these views could cater for renames of columns: in one version the view refers to a column as $A$, whilst the subsequent version refers to the same column as $B$. *EBR* does not provide functionality to refactor the underlying table. Research on using *EBR* to aid for zero-downtime solutions has been conducted by Choi et al [35].

8. **Online Redefinition Package**: Oracle provides a package with functions to redefine tables online (named `DBMS_Redefinition`). It has a similar approach to the online schema migration frameworks suggested by Google. It creates an *interim* table, which is a synonym for the previously mentioned *ghost* table. The Oracle package requires a user to define a mapping between the current and upcoming version of the to be redefined table. Original data will be copied to the *interim* table using the defined mapping. This happens in the background, whilst triggers make sure that inserts and updates also get reflected to the *interim* table. Constraints can be copied and added to the *interim* table. Finally, the original table can be swapped atomically with the prepared *interim* table. Data replication to the *interim* table allows for concurrent queries, whilst not blocking clients. The solution uses an exclusive lock for the atomic swap of both the tables [36]. The swap might result in some perceivable downtime at the end of the operation, due to the locking.

The Online Redefinition package does not provide any guidelines or tools on retaining schema compatibility.

## 6.2 Reflection

This section reflects upon the various mentioned strategies by discussing their feasibility and the degree to which they would satisfy the requirements.

1. **Use an online schema migration framework**: There are several tools out there that facilitate migrations using ghost tables. A list of several tools has been found:

   - Gh-ost
   - pt-online-schema-change
   - Large Hadron Migrator (LHM)
   - OnlineSchemaChange
   - oak-online-alter-table
   - smg-live-alter
   - QuantumDB

   All the above tools facilitate online schema migrations for MySQL. Most of them do not support Foreign Key (FK) constraints or violate them temporarily [37, p. 16], which violates requirement **R8**. QuantumDB is the only tool that is developed for PostgreSQL and is currently in an alpha phase. None of these items will integrate well in the department, due to Oracle being the backbone of most back-end applications (**R1**). These tools do not integrate with Liquibase either, except for pt-online-schema-change, which is provided by a third party.

2. **Decouple database changes and application changes**: The Expand and Contract pattern is a high level approach to the problem of satisfying Mixed-State. However, an off-the-shelf solution is not to be found and seems to be context specific. It is a logical implication that (temporarily) being out of service (i.e., downtime) will be a consequence of breaking changes to the schema. Ambler et

al. discuss patterns to refactor database schemas, including how to overcome breaking the semantics of a schema in the Mixed-State. In their book "Refactoring Databases: Evolutionary Database Design" they provide example refactorings written for Oracle databases [18]. The Mixed-State they speak about is the case of multiple applications using the same schema [18, p. 18]. They also discuss Single Application Databases (similar to what is common in the department) [18, p. 17]. In all cases, however, they do consider downtime to be part of the refactoring. When using their solutions without taking the application offline, they will violate the following requirements:

- **R7** Support for Mixed-State. This is because some refactorings are not atomic. For instance, renaming a column by using the Expand and Contract pattern requires synchronization from the old column to the new column. The book proposes to solve this by an update statement to batch copy all the data from the old column to the new. Afterwards, it proposes to create a trigger to make sure that this synchronization is taken care of for newly inserted and updated data. However, in between the update statement and the creation of a trigger, data could already have been changed or newly inserted. The auto-commit property for DDL in Oracle prevents this refactoring to be executed as a logical unit of work [26]. In the mentioned example, queries will continue to operate, but there is a risk of data loss. In different cases, it can be that during this migration, queries from the currently running application version will be temporarily broken.
- **R5** Support for non-blocking schema changes. The example above already proves this by the update statement that encompasses the batch copy. Due to the modification of all rows in the table, it requires a lock on all rows. The update could cause a deadlock to other queries that are trying to modify data in the table. For reads this would not pose a problem, as Oracle implements Multi-Version Concurrency Control where readers and writers do not block one another [38].

The book can serve as an inspiration how to tackle some refactorings on a semantics level. An assessment on a per-pattern basis has to be made on how to use these refactorings without downtime. Testing the backward and forward compatibility of a schema might be cumbersome.

3. **Design and implement a data partitioning and archiving strategy**: This item solely talks about minimizing data such that refactorings and migrations take less time. This could be beneficial in the case of using ghost tables or Expand and Contract, such that a smaller amount of data has to be copied. It also assists in making the granularity of the impact of a change smaller. By itself, this strategy will not nullify downtime when applying changes and refactorings. It can however make the migrations and changes easier, as well as lessen downtime when doing old-fashioned schema changes.

4. **Use an event sourcing architecture**: Event sourcing is a data storage model that does not store the current (or last) state, but all changes leading up to the current state [39, p. 1]. Event sourcing requires a radically different approach, which comes with its own problems to schema and data management [40, section 7]. This strategy will not be further explored due to the migration of such architecture requiring huge effort, which does not meet requirements **R1** and **R10**. The effort would entail redesigning multiple applications, possibly shifting to another database and educating staff.

5. **Use NoSQL**: As previously discussed, most NoSQL databases are not ACID compliant. ACID compliance is a debated term that is considered vague and ambiguous [12, p. 223]. However, the term is not completely meaningless and moving away from ACID databases requires a huge redesign of applications. The applications in the department have been developed with ACID properties in mind and leverage those to protect against data corruption in concurrent scenarios. Similar to moving towards an Event sourcing architecture, it will be a too large concession not to satisfy some requirements at all (mainly **R1** and **R10**).

6. **Online DDL**: Using Online DDL to facilitate changes will allow DML queries to continue to operate (such that **R5** can be satisfied). However, not all types of changes correspond to a single DDL statement. Besides, not all DDL changes can be executed without blocking clients. Furthermore, Liquibase does not support Online DDL at the moment of writing, which might be a problem for requirement **R1** (integration). Different databases support different non-blocking changes, which may affect the generalizability (**R4**) of the solution. Online DDL statements provide a simple and database native way to facilitate changes during service. This is why the mentioned concerns/challenges should be further investigated.

7. **Edition-Based Redefinition**: EBR provides great guarantees and little performance overhead, as found by Choi et al [35]. However, it does not always aid in changing an application's schema. It provides a way to version several schema objects, except for tables, such that each application version can have its own perspective on those objects. Editioning views are constructed per application version as well. Cross-edition triggers are used to perform actions when an insert, update or delete happens in one of the versions. As discussed in Change Analysis, the department does not use schema objects at all. Therefore, there is no need to be able to version those. It can be useful in some specific cases, such as the previously mentioned example of column renames. However, the corresponding table would then be left unchanged. Even though this might functionally work, it does not facilitate a true rename and over time it is likely that there is a lot of incoherence between views and tables. The mentioned constructs might affect the understandability and transparency of the solution (**R3**).

   Another aspect to consider is the support in Liquibase. Liquibase has no support for EBR at all (e.g., creating new editions and creating corresponding editioned views and triggers). It would be possible to use EBR by having plain SQL in each ChangeSet. Next to that, application code would have to be augmented such that each database session switches to the edition that corresponds to the application version. These caveats affect the detachability of the solution (**R2**) and the generalizability (**R4**).

   A more major problem is that Oracle states the following: "Every ordinary object in the application references the table only through the editioning view" [41]. The guidelines also prescribe revoking all table permissions from the application's user. Both these aspects require quite a migration path and affect both the integration (**R1**) and detachability (**R2**).

   EBR does not require copying data from one column to another for cases like renaming a column, unlike the Expand and Contract pattern. Therefore, EBR provides an interesting perspective to taking care of bidirectionality. However, it seems far-fetched in practice, because of the need to change existing applications; their corresponding databases and their table permissions first.

8. **Online Redefinition Package**: This approach is similar to the online schema migration frameworks, but is provided out of the box in Oracle Enterprise Edition. The approach comes with a few caveats:

a) It is a manual process: it requires the user to create a new ghost table and therefore requires a copy of (almost) the full table layout. It also needs to know the mapping from the current table to the interim table. Both go against the way Liquibase is used normally, where only the change to a table would be stated, without having to know the schema at the time the change is to be effectuated. Besides, the process may result in errors that need to be inspected and resolved manually. That in turn, would make it hard to automate and would require database administrators to take care of this process (which affects requirement **R1**).

b) Locking: at the end of the process, it atomically swaps the table with its interim table. This requires an exclusive lock and therefore has influence on the availability (**R5**). The amount of downtime is determined by the amount of data that still needs to be synchronized when the atomic swap takes place.



The first item might make it less attractive to use, but it can be useful in extreme situations where a lot has to be changed at once. Ideally, this would not be the case, because it is easier to keep changes small and release more often [18, p. 60]. As for the second caveat, it is unclear how long this exclusive lock takes. This could be problematic if it takes long (e.g., in case of a large table). The good part about the exclusive lock is that the previous version of the application trying to access the concerning table will wait for the lock to be released. It will therefore not temporarily break functionality, but rather be blocking instead. Note that, similar to the online schema migration frameworks, Mixed-State still has to be accounted for when using the Online Redefinition Package. The package can be combined with the Expand and Contract pattern. The manual inspection and intervention can become a bottleneck if releases have to be done often.

# 7 Solution

This chapter will discuss the developed solution to achieve zero-downtime. The solution is two-fold. The first part is to cater for non-blocking schema changes that are not supported by Liquibase (related to **R5**). Moreover, it discusses non-blocking schema changes that only update meta-data and are supported by Liquibase. The second part of the solution addresses schema compatibility (**R7**), whilst leveraging the non-blocking schema changes from the first part of the solution.

## 7.1 Non-Blocking Schema Changes

The requirement to provide non-blocking schema changes, indicates the need for Liquibase support for Online DDL. One of the deliverables of this project is a Liquibase extension. It has been developed to extend the Liquibase capabilities, such that Online DDL can be used. The following subsection describes the extension and its design decisions. The subsection afterwards discusses DDL statements that are considered to be non-blocking.



### 7.1.1 Extension

This subsection further elaborates on the newly introduced Liquibase extension. The extension is named **liquibase-osc**, where OSC stands for Online Schema Change. It contains the ability to reinterpret some changes, such that Online DDL statements are generated instead. This happens through rewriting Liquibase's original DDL statements.

## Design Decisions

To use Online DDL, one could use plain SQL in the Liquibase changes rather than typed changes. That however would negate the benefits of Liquibase and requires the change script to contain conditional SQL for each of the supported database vendors. The latter would also impose a learning curve and decrease the abstraction level.
The goal of the extension is to assist developers in doing online schema changes, without introducing a new way of working or introducing a tool with a high learning curve. The following design decisions were made in order to meet the requirements:

**E1** Changes are only rewritten for Oracle versions that support all the online changes. Rewrites should therefore only occur when the Oracle database is Oracle Enterprise Edition 19c or higher. Any older Oracle (or non Enterprise) version would end up with the default DDL statement. This relates to **R1**.

**E2** Changes that have no online equivalent remain unaffected. They should be propagated as if the extension is not there. This relates to **R1**, **R2** and **R4**.

**E3** Rollbacks of rewritten changes are rewritten to their online equivalent. When an online change has to be rolled back, it should happen online, such that existing clients do not experience downtime. This relates to **R5** and **R11**.

**E4** Rollbacks remain unaffected for changes that are not rewritten. This is similar to **E2**.

**E5** The schema semantics of a Liquibase change and their online equivalent are always identical. Therefore, the checksums of changes and their online equivalents should be equal. The idea is that a rewritten change that has been effectuated, will be considered as already propagated when disabling the rewrites (and vice versa). This relates to **R2**.

**E6** Rewriting is disabled by default. The `auto-online-ddl` property should allow a user to control when to use online equivalents. This allows projects to incorporate the extension without instantly altering behavior. Next to that, online equivalents might take longer than their offline counterparts and might not always be desirable

(e.g., when doing a deployment with downtime on purpose). This relates to **R1**, **R3** and **R10**.

## Implementation

In order to support online equivalents, the system of Liquibase needs to be able to recognize these changes. The goal is to rewrite offline statements to their online equivalent where possible (and only when the property is set). Therefore, original Liquibase changes should be intercepted.
The extension system of Liquibase provides a mechanism to do so. Each Liquibase change corresponds to a Java class. Change classes are annotated with the name of the change and its parameters. Whenever deserialization of a ChangeLog takes place, Liquibase will search for viable classes to which the change could be mapped (using the mentioned annotations). If it finds more than one that matches, it will invoke the `getPriority()` method that is defined for each class. The class with the highest priority will be the one the change maps to [42].
By inheriting from the original Liquibase change classes and specifying a higher priority, it is possible to end up with the exact same implementation details and adapt functionality only where needed. Figure 7.1 shows how online equivalent changes inherit from default Liquibase types:



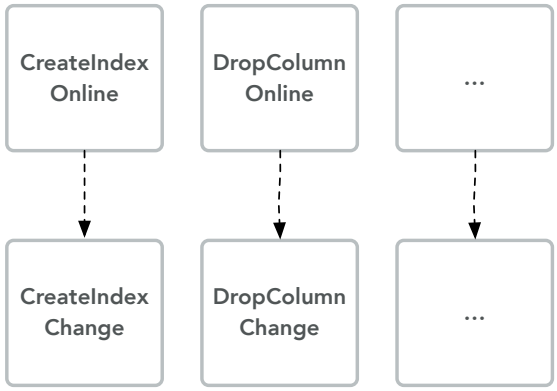



Figure 7.1: Online equivalents inheriting from their offline counterparts

The online equivalents have to override the `generateStatements` method, which is used to generate instances of `Statement` classes. `Statement` classes naturally correspond to SQL statements that have to be executed. An online statement should only be generated if the `auto-online-ddl` property is true and the used database is Oracle Enterprise Edition (version $\geq 19c$). Otherwise, it should result in the original Liquibase statement by returning the parent's result of `generateStatements`.

### Extensibility

The described inheritance allows new Online DDL statements to be added easily. Previously adopted Online DDL statements remain unaffected, due to each statement having its own class.
`Statement` classes result in DDL through a `Generator` class. Each `Generator` class generates DDL for one `Statement` class. The `Generator` class uses pattern matching to facilitate DDL generation for different database vendors. To support a new database, the pattern matching has to be extended to match with the new database, such that it can result in DDL that is specific for that database.

### Example Change

The following snippet illustrates how an online change can be documented in Liquibase using the extension. The format of the ChangeLog is in XML:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <databaseChangeLog
3    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.liquibase.org/xml/ns/
       dbchangelog http://www.liquibase.org/xml/ns/
       dbchangelog/dbchangelog-3.10.xsd">
6
7    <property name="auto-online-ddl" value="true"/>
8
9    <changeSet id="create-index-rewrite" author="jorryt">
10     <createIndex indexName="my_index" tableName="
         my_table">
11       <column name="X"/>
12       <column name="Y"/>
13     </createIndex>
14   </changeSet>
15 </databaseChangeLog>
```

The above example entails creating an index on two columns. `createIndex` is a standard type in Liquibase. The only addition to the script is the `auto-online-ddl` property, which indicates that changes in this ChangeLog may be reinterpreted to their online equivalent. The change will translate to the following SQL:

```
CREATE INDEX `my_index` ON my_table(X, Y) ONLINE;
```

Without the property, the result would be the DDL that Liquibase would generate without the extension:

```
CREATE INDEX `my_index` ON my_table(X, Y);
```

On a non-Oracle database, no rewrite would take place either. It would likely be similar to the above DDL statement, but that depends on the Liquibase implementation for the specific database vendor.

**Supported Changes**
The following Liquibase changes are implemented as online equivalents:

- `createIndex`
- `dropColumn`
- `dropForeignKey`
- `dropIndex`
- `dropPrimaryKey`
- `dropUnique`



### 7.1.2 Meta-Data DDL statements

Some DDL statements can be effectuated on existing tables by solely changing the definition of the table itself (often called meta-data). Such statements do not inspect nor change the underlying data of the table. Statements with these characteristics will be referred to as Meta-Data DDL statements. Meta-Data DDL statements take little time, but sometimes require concessions in terms of data integrity. Useful examples of Meta-Data DDL statements are:

- Mark a column unused [43]
- Adding a column with a default value [44]
- Adding a column with a null value [44]
- Renaming schema objects
- Expanding the size of vector types (such as `varchar2`)

- Adding constraints that only do integrity checks for new statements, but not for existing data [45] (often referred to as `novalidate`)
- Changing constraints to only do integrity checks for new statements, but not for existing data [45]

Some of the above examples are not claimed by Oracle in their documentation and will therefore be validated in the Testing chapter.
Liquibase supports the above constructs, except for marking a column unused. Therefore, the extension plugin implements it by reinterpreting `dropColumn` statements.

## 7.2 Expand and Contract

This section discusses several patterns that can be used to achieve schema compatibility. The patterns use the Expand and Contract pattern to facilitate schema compatibility. The patterns can be seen as alternatives for known (often simpler) changes that otherwise would break schema compatibility or/and would be blocking.
For each Expand and Contract-phase, all schema changes have to be effectuated in order to serve clients through the new version. Spring Boot's integration with Liquibase caters for this, as it requires all schema changes to be executed successfully before it allows any interaction through the API.
There are two different approaches to using the Expand and Contract pattern. Both will be explained separately. Each Expand and Contract pattern will be discussed in its own section.

### 7.2.1 Approaches

There are two approaches to Expand and Contract, which are completely interchangeable.

### In-Place

The first approach to Expand and Contract is doing changes in-place. Whenever a change to a column has to be propagated, the subject table will be changed during the migration. As the migration happens whilst the table is in use, none of the steps should be blocking.

### Interim Table

The alternative approach to the Expand and Contract pattern, is doing changes using an interim/a ghost table. Whenever a change to a column has to be propagated, this will be propagated to the copy of the to be changed table instead. Each arrow depicted in the Figure 6.1 requires the creation of an interim table, whilst each deployment requires a swap between the original table and the interim table. This approach is achieved through the Oracle's Online Redefinition package.

## 7.2.2 Rename Column



This subsection illustrates the scenario of renaming a column. The starting point is a `customer` table that contains a non-nullable column: `name`, and a nullable column: `address`. The corresponding entity looks as follows:

```
1  @Entity
2  data class Customer(
3    @Column val name: String,
4    @Column val address: String?,
5    @Id
6    @GeneratedValue(strategy = GenerationType.IDENTITY)
7    val id: Long = 0
8  )
```

In this example, the goal is to rename the `name` column to `full_name`.

## Expand

To retain schema compatibility, the Expand-phase has to take care of the following steps:

1. Add a new column with the new name
2. Copy data from the old to the new column
3. Guarantee data symmetry between both of the columns[1]

The $V_1 \cup V_2$ entity in the Expand-phase looks as follows:

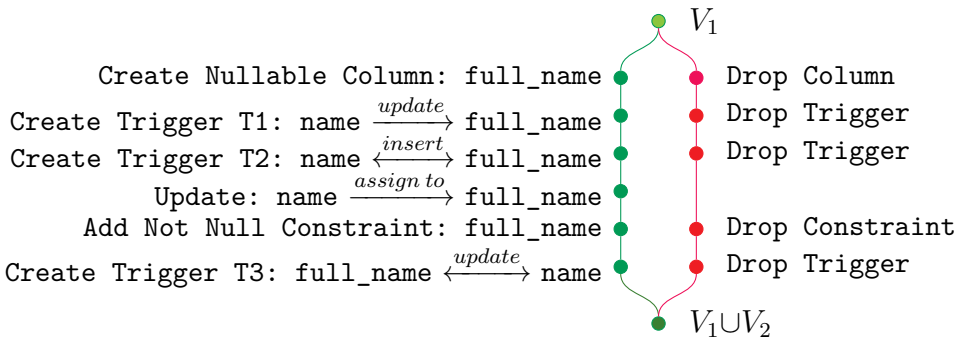
```
1  @Entity
2  data class Customer(
3    @Column val fullName: String,
4    @Column val address: String?,
5    @Id
6    @GeneratedValue(strategy = GenerationType.IDENTITY)
7    val id: Long = 0
8  )
```

Note that schema isolation is taken into account, as the entity has no notion of the `name` column.

**In-Place**

The following graph depicts the schema changes that are required to do the expand in-place. The indented green nodes correspond to schema changes. The red nodes next to them represent their corresponding roll-back statements (to revert the statement denoted at the green node).

[1] Data symmetry in this case means that the values in the old and new data columns are always equal



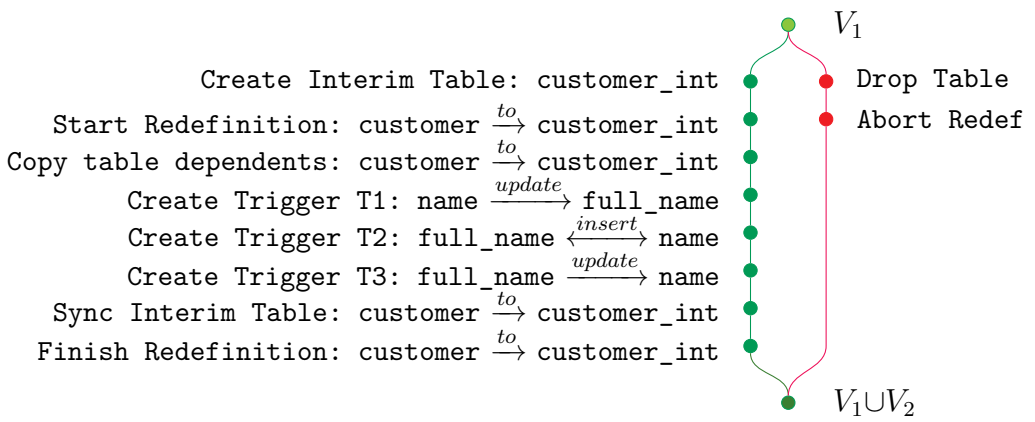

All required schema operations are atomic and non-blocking. The `full_name` column has to be initially nullable (or have a default value), because the table might contain data already (and that would result in a violation of the not-null constraint). After that, triggers are introduced such that inserts and updates that fill the `name` ($V_1$) column also propagate to the new `full_name` column. Then data is copied from `name` to `full_name`, to synchronize the rows that did not pass through triggers `T1` and `T2` in the meantime. After this step, there is the guarantee that the `full_name` is filled everywhere, and it is safe to add the not-null constraint. Finally, the trigger to update `name` whenever `full_name` is updated can be added, such that updates from the new version will also be propagated.
There are two caveats to the described approach:

- The `update` statement that copies data from one column to the other, locks all affected rows. In the worst case, that would entail all rows in the table. Queries of the table will not be affected, but writing will have to wait until the locks are released [46][12, p. 239]. Whether this is a viable solution depends on the balance of reads and writes during the maintenance time window, as well as the size of the table.
  In order to cater for large databases and use cases with a lot of write operations, an alternative to the `update` statement has been designed. The semantics and ideas remain the same and are presented separately in the Non-Blocking Data Migrations section.

- The addition of the not-null constraint will lock all rows similarly to the `update` statement. Therefore, it is necessary to propagate the change without validating the constraint for the existing rows (leveraging Meta-Data DDL).

### Interim Table

With the interim table approach, the following steps have to be taken:




In the Expand-phase, the interim table contains both a `name` column and `full_name` column. Start redefinition, copy table dependents, synchronize interim table and finish redefinition table are all functions provided by Oracle's Online Redefinition package (`DBMS_Redefinition`). The start redefinition step requires a mapping from old column names to new column names, such that synchronization between the original and the interim table is possible. In this case, the mapping would require that `name` from `customer` maps to both `name` and `full_name` in `customer_int`. The copy table dependents function copies indexes, triggers, constraints and grants to the interim table.

It is important that the triggers are created prior to swapping the tables. This is to make sure that when the expanded table is available as `customer`, the data in `name` and `full_name` columns remain symmetrical whenever inserts or updates occur. The synchronize interim table step entails an explicit command to synchronize the tables, such that finish redefinition does not need to do much synchronization during the swapping of the tables anymore. This is useful because swapping the tables requires an exclusive lock, which can result in perceived downtime if a lot of synchronization still has to happen.

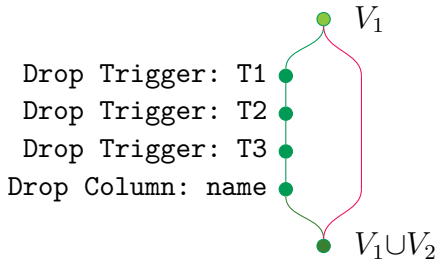
The redefinition process can be aborted anytime (by the `ABORT_REDEF_TABLE` function) and the interim table can be dropped, without affecting the `customer` table and its clients.

## Contract

The Contract-phase can take place as soon as $V_1$ is not serving clients anymore and can be taken offline. The Kotlin code does not need to change due to schema isolation. Both the Expand-phase approaches, require triggers and the `name` column to be dropped during the Contract-phase.

### In-Place

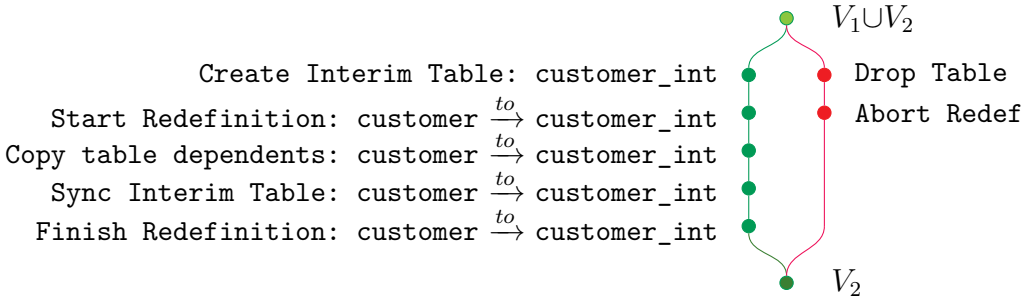
In-Place changes to the schema are depicted in the following graph:




After these schema changes are applied, a rename of `name` to `full_name` has been effectuated. There are no automatic rollbacks in place: any step from $V_1 \cup V_2$ to $V_2$ retains schema compatibility, because of schema isolation. Both $V_1 \cup V_2$ and $V_2$ do not know the `name` column and correspond to the same entity code. All steps after $V_1 \cup V_2$ break schema compatibility with $V_1$.

### Interim Table

The interim table alternative to contract the schema looks as follows:




The interim table has to be created without the `name` column, such that `full_name` in `customer` maps to `full_name` in `customer_int`. The caveat

here is that copy table dependents should be instructed not to copy triggers and constraints, because those would not function due to omitting the `name` column in `customer_int`. Any constraints and triggers that are left should be copied manually. The manual overhead for cleaning up, makes this less attractive than the in-place alternative.

### 7.2.3 Rename Foreign Key Column

This section describes how to rename a foreign key column without downtime. The solution is similar to renaming a regular column, but requires some extra steps for the constraints. The example of a bank account is used, which has a foreign key relationship to the `Customer` entity. The `Account` entity looks as follows prior to the migrations:

```
1  @Entity
2  data class Account(
3    @Column val accountNumber: String,
4    @Column val balance: Long,
5    @Column val currency: String,
6    @ManyToOne(fetch = FetchType.LAZY)
7    @JoinColumn(name = "owner")
8    val owner: Customer,
9    @Id
10   @GeneratedValue(strategy = GenerationType.IDENTITY)
11   val id: Long = 0
12 )
```

The goal is to rename the `owner` column to `owner_id`.

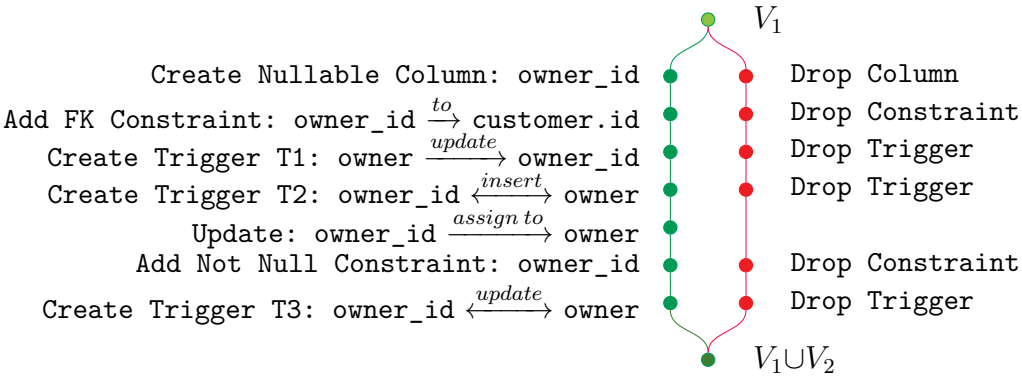
#### Expand

In the Expand-phase, the entity should have no notion of the `owner` column. Instead, it looks as follows:

```
@Entity
data class Account(
  @Column val accountNumber: String,
  @Column val balance: Long = 0L,
  @Column val currency: String,
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "owner_id")
  val ownerId: Customer,
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  val id: Long = 0
)
```

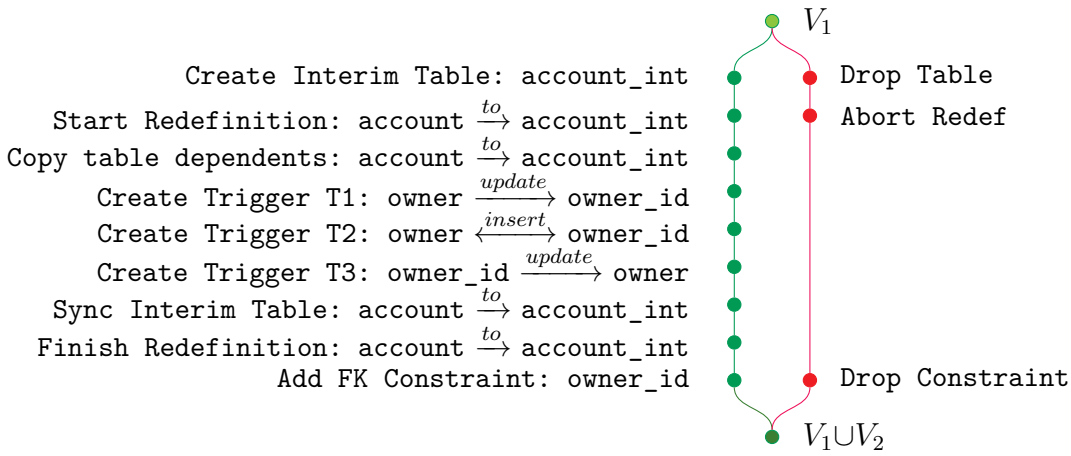
### In-Place

The following graph depicts the schema changes that are required to do expansion in-place:




### Interim Table

With the interim table approach the following steps have to be taken:




Note that the approach is similar to renaming a column. The FK constraint is added after swapping the tables, due to referential constraints needing to be disabled during the redefinition [36]. To prevent blocking $V_1$ clients, the addition of the constraint should be a Meta-Data DDL update. This means that existing rows are not validated prior to setting the constraint.

7

### Contract

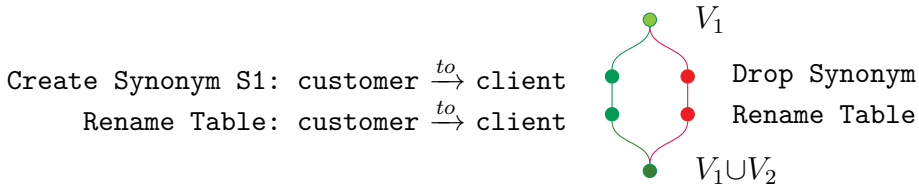
The Contract-phase is almost the same as the one for renaming a regular column. The only difference is that prior to dropping the old column, the old foreign key has to be dropped. This can be achieved by Online DDL.

## 7.2.4 Rename Table

This subsection will describe how to rename a table using the Expand and Contract pattern. The example is based on the same `Customer` entity as the previous examples. The goal is to rename the `customer` table to `client`. There is only one approach to renaming tables without downtime. It does not require any duplication and therefore could be considered in-place.

## Expand

The following graph shows an example of the Expand-phase:




The rollback for rename table does the opposite: it renames the `client` table to `customer`. The corresponding entity for $V_1 \cup V_2$ looks as follows:

```
1  @Entity
2  data class Client(
3    @Column val fullName: String,
4    @Column val address: String?,
5    @Id
6    @GeneratedValue(strategy = GenerationType.IDENTITY)
7    val id: Long = 0
8  )
```

The above approach is similar to what Ambler et al. propose to rename tables whilst retaining schema compatibility [18, p. 115]. However, their approach still requires downtime, due to auto-commit of DDL [26], where in between the steps there is a short moment of schema incompatibility. Renaming the table and creating an alias (called synonym in the case of Oracle) can not be done atomically, due to both being DDL statements which will be auto committed.
The presented solution introduces a `customer` synonym in the `public` scope, such that it refers to `client` (which will be the new table name) in the current schema. This is done prior to renaming the `customer` table to `client`. A synonym is private by default, which means that it resides in the scope of the schema. A private synonym would not be possible in this case, due to a name clash with the `customer` table. The creation of a public synonym however, would not result in a name clash. This allows the previous version of the application to be schema compatible, by exploiting Oracle's implementation of object name resolution. Oracle's name resolution scans the public scope as soon as it can not resolve an object in the schema scope [47]. Renaming the table would then retain

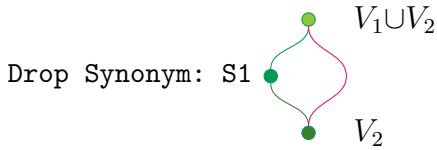
schema compatibility, because queries in the previous version can be resolved through the synonym in the public scope. The rename statement would happen quickly due to it being a Meta-Data DDL statement.
The public scope makes it visible to other users that might use the same database. A database user that wants to operate using a synonym, requires privileges to the underlying schema object. Whether this is a viable solution depends on the use case:

- Does the user have privileges to create public synonyms?
- Is it acceptable that other users may see this synonym?
- Are there clashes with existing objects in the public scope?

### Contract

The Contract-phase can take place as soon as $V_1$ is not running anymore. The changes to the schema look as follows:






As soon as there is no instance of application $V_1$ anymore, there is the guarantee that synonym `S1` is no longer used. Therefore, it is safe to drop it as both $V_1 \cup V_2$ and $V_2$ already operate on the new table name.

## 7.3 Non-Blocking Data Migrations

An alternative to the `update` statements has been designed to overcome the extensive row locking. This allows the previous application version to continue using the table, whenever data copying takes place. The idea is to have an approach that has the same semantics as the following statement (but without the extensive locking):

```
UPDATE `my_table` SET newCol = oldCol;
```

A custom Liquibase change has been introduced: `BatchMigrationChange`. A ChangeSet with such change looks as follows:

```
<changeSet id="migrate" author="jorryt">
  <customChange
    class="liquibase.ext.changes.BatchMigrationChange">
    <param name="tableName" value="my_table" />
    <param name="fromColumns" value="oldCol" />
    <param name="toColumns" value="newCol" />
    <param name="chunkSize" value="1000" />
  </customChange>
</changeSet>
```

The parameters are used to specify the table, as well as the source columns and target columns for the copy operation. The source and target columns are not allowed to overlap: columns $\langle A, B \rangle$ can not be copied to $\langle B, C \rangle$, but can be copied to $\langle C, D \rangle$. The change is implemented using a divide-and-conquer approach to overcome extensive locking. It splits the workload in chunks of equal sizes. The size is specified by a parameter as well (`chunkSize`). Determining the right size is use case specific: with a very small size it may take very long to migrate, but with a too large size there might be too much locking.



An important caveat is that the data in the table might grow during the migration. The triggers in the Expand-phase take care of the mapping from `oldCol` to `newCol`, such that new data does not have to be taken into account by the `BatchMigrationChange`. However, it is difficult to generically determine what data is new and what is not. Without knowing the actual table structure, it is impossible to know upfront what the relevant subset of records is that needs to be migrated. What uniquely identifies a row is case specific (primary key or a unique constraint for example), and sometimes a row is not unique either (e.g., in logging tables). Besides, it is risky to keep the state of the migration in-memory: for large tables this might require a lot of memory, which pipelines cannot always allocate. Moreover, the state is lost whenever the Liquibase execution crashes.

In Oracle, each row contains a `rowId` column, which is the address column of that row. This column provides a unique identifier for the row, but unfortunately Oracle does not provide any guarantee in terms of the order: a deleted row's `rowId` might be reused, but that does not have to be the case. A combination of query results and the above guarantees, can lead to migrations without explicit bookkeeping of the state. The

`BatchMigrationChange` would execute the following code for the outlined example:

```
1  UPDATE my_table
2  SET newCol = oldCol
3  WHERE rowId IN
4  (SELECT rowId
5    FROM my_table
6    WHERE (newCol IS NULL and oldCol IS NOT NULL)
7    FETCH FIRST 1000 ROWS ONLY
8  );
```

This statement executes multiple times until a fix-point is reached. The fix-point is reached when no rows are affected by the statement. In combination with the triggers, we have the guarantee that after that fix-point, values from both columns are identical. Each statement runs in its own transaction, such that only a subset of the rows need to be locked during the update.

This solution would also work in case `oldCol` would be nullable: all `newCol` columns are nullable by default (which is required for the Expand and Contract patterns). The rows where `oldCol` is null would not need migration, whilst the rows that would have values for `oldCol` would result in copying.

# 8 Testing

This chapter discusses how the solutions have been tested. The testing of this project is automated to test the technical effects of the solution, which are related to requirements: **R5**, **R7**, **R8** and **R11**. Primarily, these tests provide evidence that the extension and the Expand and Contract strategies are correct. Secondarily, they serve as an example for future Online DDL and Expand and Contract patterns.
As the solution consists of two deliverables, both have been tested separately. Both have their own section, which elaborates on what has been tested and how it was tested. Extensive use of Property-Based Testing has been made in order to cover a wider variety of scenarios. Papadakis et al. describe Property-Based Testing as follows [48]:

> "Property-Based Testing (PBT) is a novel approach to software testing, where the tester only needs to specify the generic structure of valid inputs to the program under test, along with a number of properties (regarding the program's behavior and the input- output relation) which are expected to hold for every valid input."



How PBT is used will be described on a per-case basis.

## 8.1 Extension

Each supported DDL statement that has an online alternative can be rewritten to its online equivalent. There is no value in validating the semantic equivalence for DDL statements and their online counterparts. It is safe to assume that this well tested by Oracle, as they officially provide these statements as alternatives.
There are other aspects to validate to determine correct functioning of the extension. These aspects come from the design and are key to keeping behavior predictable. Each of the design choices was tested extensively using PBT:

**E1** Changes are only rewritten for Oracle versions that support all the online changes: Generate changes that have an online equivalent and stub an Oracle database to return a specific version. Online statements should only be generated when the Oracle version is $\geq$ 19c.

**E2** Changes that have no online equivalent remain unaffected: Generate changes that have an online equivalent for Oracle and generate different non-Oracle database stubs (including ones that have a version $\geq$ 19c). None of them should generate online statements.

**E3** Rollbacks of rewritten changes are rewritten to their online equivalent: Generate online statements. All the rollbacks should be online statements.

**E4** Rollbacks remain unaffected for changes that are not rewritten: Generate changes that have an online equivalent for Oracle and generate different non-Oracle database stubs (including ones that have a version $\geq$ 19c). None of their rollbacks should result in an online statement.

**E5** The schema semantics of a Liquibase change and their online equivalent are always identical: Generate online changes. Each checksum should be equal to their parents' checksum.



**E6** Rewriting is disabled by default: Generate changes that have an online equivalent and stub an Oracle database ($\geq$ 19c). Generate the `auto-rewrite-ddl` property as well. The generated result for the change should be an online statement if and only if the property is set to true.

Additionally, it is worthwhile to test whether Liquibase deserializes ChangeLogs (in any of the supported formats) correctly, such that the newly introduced change classes get instantiated where applicable. The deserialization mechanism is extensively tested in the Liquibase project. However, it makes sense to test whether it can instantiate the newly introduced classes whenever it should. A few example integration tests for this suffice and are implemented.
A simple way to verify whether online SQL is generated for a ChangeLog, is to run the Liquibase CLI with the `updateSQL` parameter, which outputs the SQL that would otherwise be propagated to the database [49]. Besides, tests of the Expand and Contract pattern also test generation of

Online DDL implicitly, as these Online DDL statements serve as building blocks there.

## 8.2 Expand and Contract Patterns

It is important to extensively test whether the Expand and Contract patterns successfully achieve the following technical aspects of the solution:

- **R7** *Concurrently Active Schemas*: Both database schemas should be active at the same time. Data found in one version should be available in the other, and the data should adhere to both schemas.
- **R8** *Referential Integrity*: Foreign keys should be enforced at all times.
- **R11** *Resilience*: Rollbacks should not affect current database clients. These were only tested for patterns that use the In-Place approach. The Interim Table patterns can be aborted any time by the `ABORT_REDEF_TABLE` command in Oracle's Online Redefinition package. It is safe to roll back half way, as the steps in the Interim Table approach do not make any changes to the original table until the final table swap.



The upcoming section will talk about the approach and architecture of the Expand and Contract tests. The subsequent sections address the mentioned technical aspects on a per-section basis.

### 8.2.1 Architecture & Approach

Testing two versions of an application with two schemas is not trivial. Both applications have their own perspective, which is a subset of the actual schema. An application has been developed, which uses a similar technology stack and design to what is common in the department. The application models the situation of two applications using the same database schema. A basic architecture depiction of the application looks as follows:

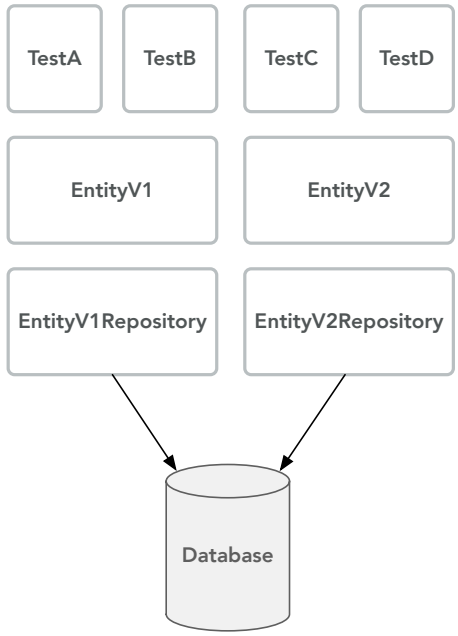



Figure 8.1: Architecture that facilitates testing concurrent application versions

The squares in Figure 8.1 depict classes, whilst the arrows represent the database connection. The larger boxes represent classes that are versioned according to the application version that it resembles. The entities entail classes that correspond to a table in the database. The `V1` classes comprise the perspective of the initial version of the application (prior to the migration phase). The `V2` classes correspond to the evolved (i.e., the Expand-phase) versions of that class. The `EntityRepository` classes are the classes that implement the repository architecture pattern, such that database interaction for a particular `Entity` class is provided.

## Required Workaround

The described architecture needed one specific workaround to model two application versions at the same time: a synonym for the `EntityV2` class had to be introduced, such that `EntityV2` could represent the same table as `EntityV1`. This is to circumvent a constraint of the Hibernate framework, which enforces each table to correspond to a maximum of one `Entity` class. Note that the rename table scenario could not be tested due to this implementation detail. This is because the rename table solution introduces a synonym itself. The three mentioned aspects are tested differently for this scenario, namely by Performance Tests.

### Strategy

PBT was used to validate properties and to generate data to fill the tables with. The test strategy closely mimics deployment and usage scenarios. On a high level, there are a few different starting points for the schema migrations, which need to be tested:

- An empty $V_1$ schema without activity
- A non-empty $V_1$ schema without activity
- An empty $V_1$ schema with activity
- A non-empty $V_1$ schema with activity

## 8.2.2 Concurrently Active Schemas

Testing concurrently active schemas is only relevant for the Expand-phases of the Expand and Contract patterns. This is because both application $V_1 \cup V_2$ and $V_2$ know exactly the same schema (which is a consequence of schema isolation). The Expand-phase tests consist of the following steps:



1. Create a schema $V_1$
2. Generate database activity (create, read, update, delete) for $V_1$
3. Migrate the database schema to $V_1 \cup V_2$ in a separate thread
4. After the $V_1 \cup V_2$ schema is active: generate database activity with $V_1 \cup V_2$
5. Validate whether data in $V_1$ is isomorphic to $V_1 \cup V_2$
6. Validate whether data originating from $V_1$ is still present
7. Validate whether data originating from $V_1 \cup V_2$ is still present

Depending on the starting point: different permutations can be made, such as reordering steps 2 and 3. Furthermore, one-sided activity from $V_1 \cup V_2$ should be reflected in $V_1$ and vice versa. Another scenario that is tested, is data insertion by one version, which gets edited by the other

version. The edited data should be reflected in both versions.
In the Contract-phase, the expected schemas of application $V_1 \cup V_2$ and $V_2$ are equal (due to the schema isolation). All that needs to be tested is that there are no regressions caused by cleaning up, such that database interactions remain functional.

### 8.2.3 Referential Integrity

Referential integrity is tested for both the Expand and Contract-phase of the rename foreign key column solution. Additional tests verify that only existing records could be referred to.
Additional tests for the other Expand and Contract patterns are not necessary, due to them not affecting foreign keys in their strategies.

### 8.2.4 Resilience

Rollback tests consist of the following steps:

1. Create a schema $V_1 \cup V_2$
2. Generate database activity (create, read, update, delete) for $V_1$
3. Execute rollbacks from $V_1 \cup V_2$ to $V_1$ one by one in a separate thread

The test check that no activity aborts, and that data modifications are reflected properly. The scenario of renaming a foreign key column also tests referential integrity during rollbacks.

## 8.3 Performance

Performance tests have been conducted for a couple of Expand patterns as well as some non-blocking DDL statements. The Contract patterns are not performance tested, because they contain breaking schema changes, which will break transactions of any benchmark tool. However, the building blocks of the Contract patterns (namely non-blocking DDL statements), are performance tested separately in this section.

The performance tests measure transactions per minute (TPM) and transaction latency in microseconds. The TPM reflects the size of the workload that the database handles. The latency reflects the time transactions take.

HammerDB is a known database performance benchmarking tool, which is based on a specification created by the Transaction Processing Performance Council (TPC). TPC-C is a common specification for Online Transaction Processing (OLTP) benchmarks. It represents a use case of a warehouse that processes orders and maintains a stock. Wevers et al. used HammerDB to benchmark performance of (online) schema changes [50]. The broad market adoption of HammerDB, as well as being able to compare results to the study of Wevers et al. led to choose HammerDB as the benchmarking tool for this project. HammerDB is designed to test on a fixed schema and for a fixed time. In this project, benchmarking tooling has been designed that builds on top of HammerDB. The key adaptations to cater performance testing migrations are:

- To run HammerDB in an automated fashion, using a script (rather than through a GUI)
- To log each transaction
- To log and copy all run details
- To run a schema change prior to starting the benchmark
- To run a schema change after a specified time during the benchmark
- To specify the time window prior to executing the schema change (called: pre-migration)
- To specify the time window after executing the schema change (called: post-migration)
- To compute TPM per type of transaction using a sliding window

Performance testing helps to verify the following aspects:

- That migrations scripts are non-blocking (**R5**)
- That the schema remains compatible (i.e., it does not break HammerDB's transactions) (an aspect of **R7**)

- That migration scripts actually get scheduled and propagated (during high load)

It also provides insights in the following aspects:

- The duration of schema changes during high load
- The effect of a schema migration on TPM and latency during high load
- The effect after the schema migration on TPM and latency during high load

All benchmarks will be presented using throughput and latency charts. The red dotted lines are considered to be ramp-up time, which is a property of HammerDB that is used to reach a steady transaction rate prior to measuring [51]. In this project, it is simply considered to be warm-up time. The green dotted lines indicate the start of schema migrations. The blue dotted lines represent the moment when schema migrations are finished.

### 8.3.1 Test Setup



This subsection shortly describes the test setup for replication purposes. It also provides insights in the amount of initial data prior to starting the measurements.

#### Server

The specifications of the server (Dell OptiPlex 7050) that hosted Oracle were:

| | |
|---|---|
| Operating System: | Windows 10 Enterprise, v10.0.19041 Build 19041 |
| Processor: | Intel i7-7700 3.6GHz (4 cores / 8 threads) |
| Memory: | 16GB |
| Disk: | Toshiba THNSN5256GPUK 256GB NVMe SSD |
| Connection: | 1Gbit (internet) |

The server ran the following database: Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production, Version 19.3.0.0.0.

## Client

The client that was used to performance test, was a desktop computer with the following specs:

| | |
|---|---|
| Operating System: | Arch Linux with Kernel 5.9.14-arch1-1 x64 |
| Processor: | AMD Ryzen 5900X (12 cores / 24 threads) |
| Memory: | 32GB |
| Disk: | Samsung 970 Evo 1TB NVMe SSD |
| Connection: | 1Gbit (internet) |

The performance tests ran in a Docker container that had no resource limitations. The orchestration and wrapping was built and executed using Python 3.9.1. The Docker version was 19.0.14 (5eb3275d40).

## HammerDB Parameters

HammerDB was instructed with the following parameters:

| | |
|---|---|
| Warehouses: | 40 |
| Users Creating Load: | 64 |
| Ramp-up Time: | 10 minutes |
| Pre-migration Time: | 10 minutes |
| Post-migration Time: | 10 minutes |



This resulted in the following data prior to starting the ramp-up:

| | |
|---|---|
| Customers: | 1,200,000 records |
| Districts: | 400 records |
| History: | 1,200,000 records |
| Items: | 100,000 records |
| Orders: | 1,200,000 records |
| Stock: | 4,000,000 records |

### 8.3.2 Baseline

A baseline without any schema changes has been measured, such that it can be used to compare to. Note that there is no migration period, which means that the green and blue dotted lines are omitted from the baseline charts. All the performance measurements use similar benchmark settings. The baseline run of ∼30 minutes resulted in the history and order tables to more than double in size. They grew to 2,493,086 records and 2,481,733 records respectively.

#### Throughput

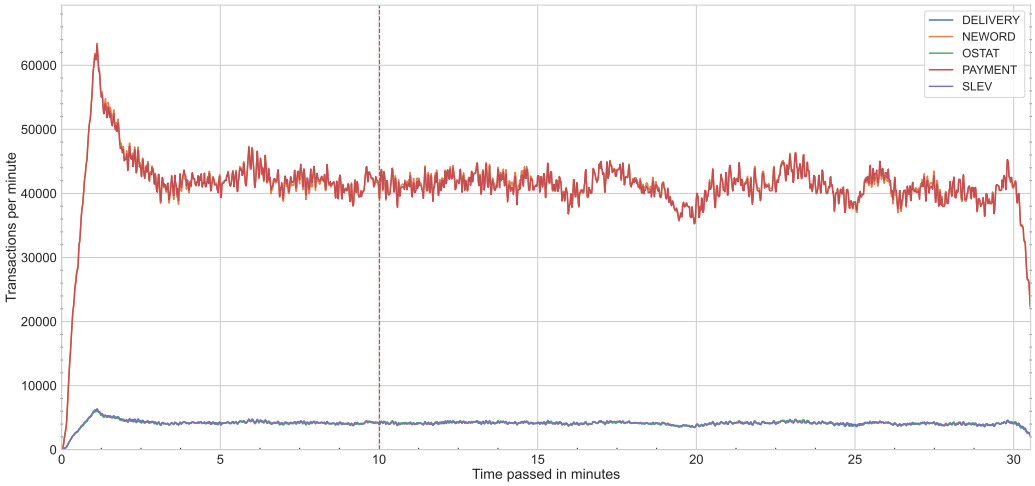



Figure 8.2: Baseline - Throughput

Figure 8.2 depicts throughput per transaction type over time. The transaction types and their distribution come from the TPC-C specification. The plot shows a slight decrease of TPM overtime, the difference however is very small and does not need any further diving into. What is interesting is that even in the baseline, slight drops occur (as can be seen around 18 minutes). This is good to take into account when looking at the effects of schema changes.

### Latency

Most of the migrations that were used for performance testing affect tables related to placing orders. This is because 45% of all the TPC-C transactions are new orders [52]. For brevity, only the latency chart for new orders is displayed here:

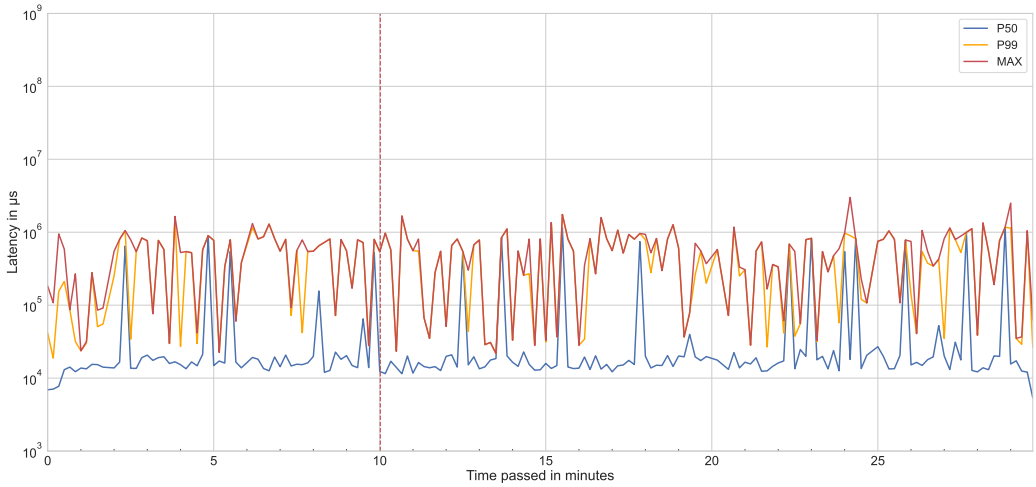



Figure 8.3: Baseline - Latency

The chart presents the latency of `NEWORD` transactions. The three lines represent measurements of the median (P50), the 99th percentile and the maximum over time. The vertical axis is logarithmic and displays the latency in microseconds. It is plotted using logarithms to visualize the large distribution of values more easily and to put more emphasis on the order of magnitude of latency values.
The median seems to follow a pattern of peaks every few minutes. The reason is unclear, but it is good to keep into account when analyzing latency values of the upcoming performance tests.

## 8.3.3 Meta-DDL Statements

This subsection provides performance tests for Meta-Data DDL statements, which have not been directly found in the Oracle documentation to be non-blocking. Some claims are by third parties, rather than by Oracle.

## Add Column (default)

This subsection depicts the performance of a migration in which a column with a default value gets added.

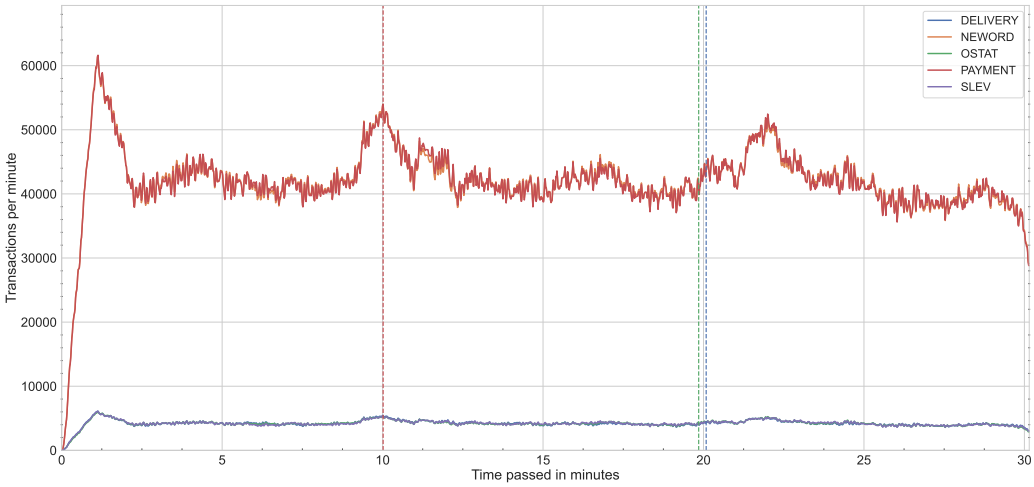



Figure 8.4: Add Column with a Default Value - Throughput

Adding a column with a default value seems not to have a visible effect on the throughput. Besides, the change is almost instant.

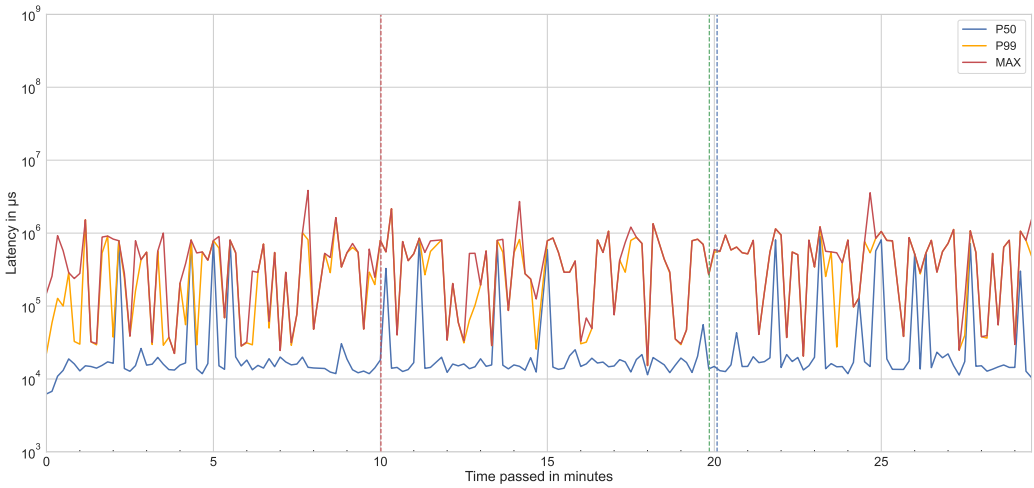



Figure 8.5: Add Column with a Default Value - Latency

The change has no visible impact on the latency of the `NEWORD` transactions.

## Add Column (nullable)

The add column with a nullable value scenario, is performance tested implicitly using the Expand and Contract pattern. It is however also tested in isolation:

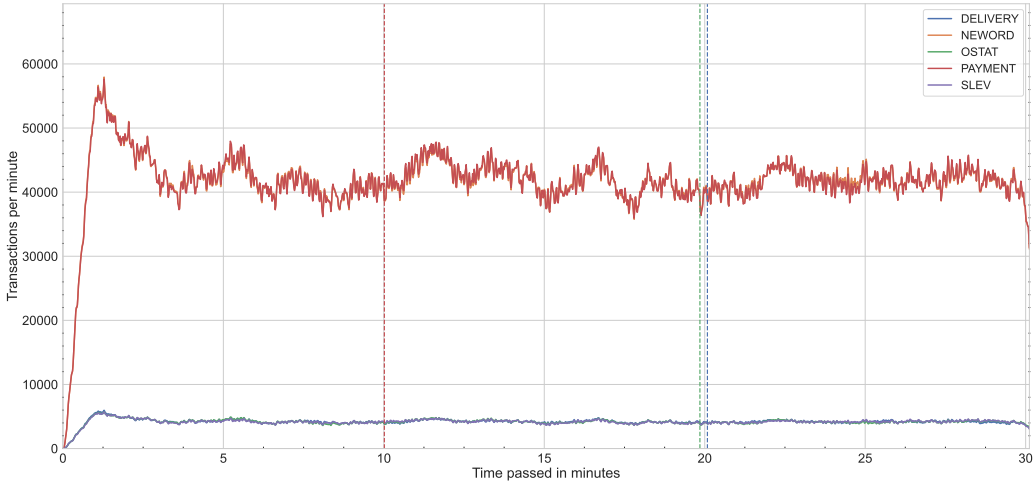



Figure 8.6: Add Nullable Column - Throughput

Similar to adding a column with a default value, there is no significant impact on throughput.

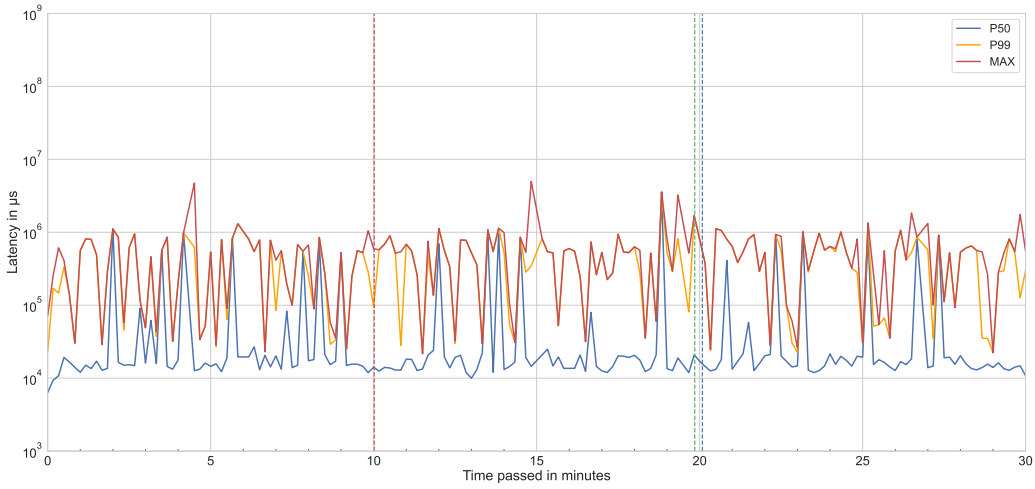



Figure 8.7: Add Nullable Column - Latency

The addition of a nullable column comes with no observable effect on the latency.

## Expanding Size of Varchar

The Change Analysis section mentioned two cases of datatype resizing. Both cases concerned a `varchar2` type. The assumption of it being solely a Meta-Data DDL statement required to be validated through a test under heavy load:

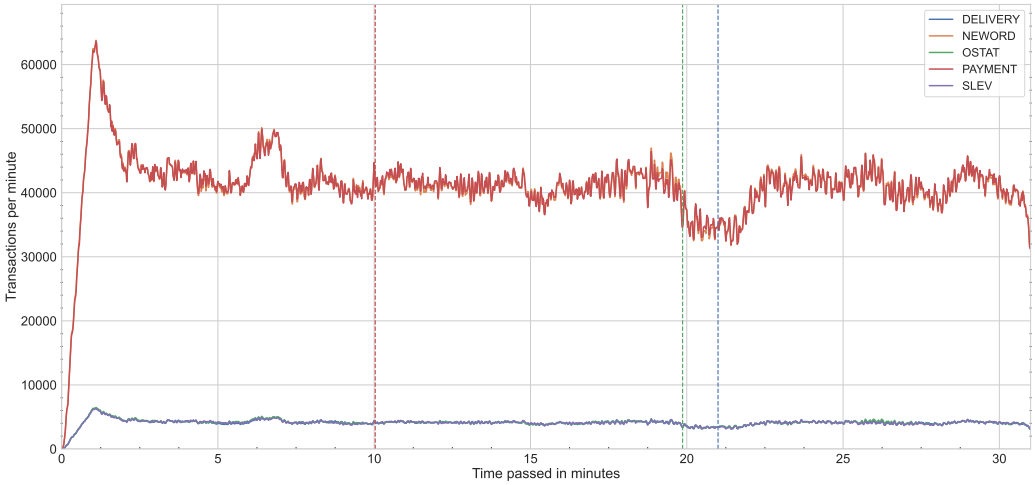



Figure 8.8: Expand Varchar - Throughput

The figure shows that expanding a `varchar2` column has impact on the TPM. It only takes a couple of minutes and does not drop to an extreme.

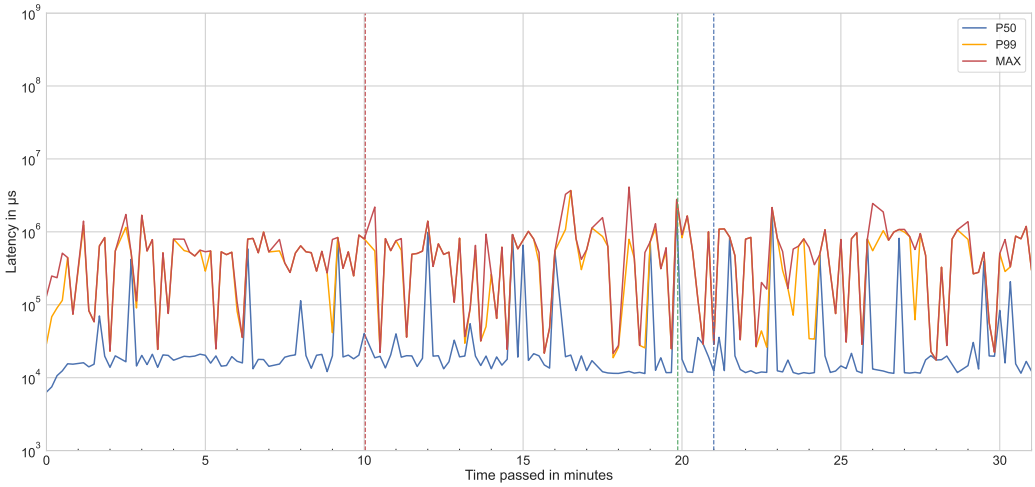



Figure 8.9: Expand Varchar - Latency

The figure does not show significant impact on the latency. A peak in the median latency during the migration might be an effect of it, but it seems to vary little from the other peaks.

## Expand Size of Char

Another performance test has been conducted to verify if expanding the size of a `char` column is a Meta-Data DDL update. The `char` type represents a fixed-length string, which pads smaller strings with blanks to meet the fixed length.

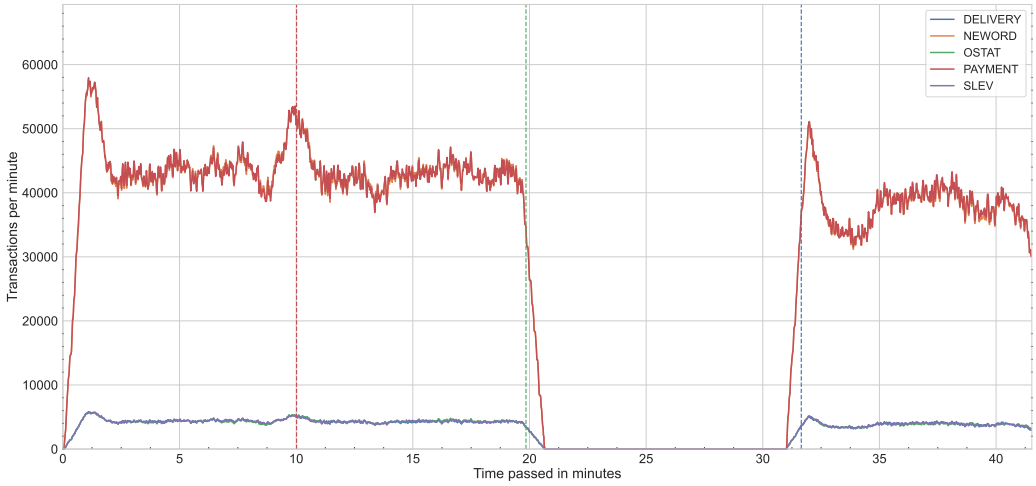



Figure 8.10: Expand Char - Throughput

The figure clearly shows that transactions are not executed during the migration. It is not safe to do this change whilst serving clients.

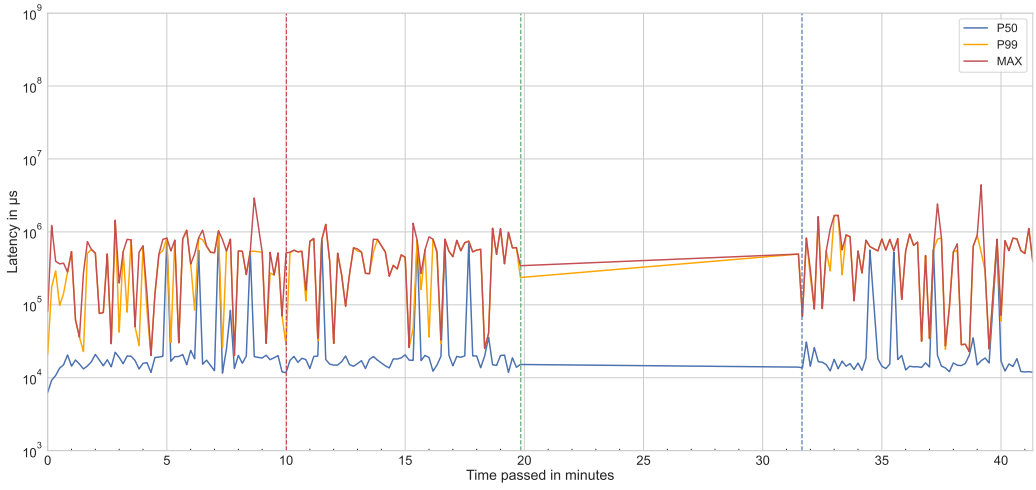



Figure 8.11: Expand Char - Latency

This chart shows that no latency measurements could be done during the migration. It also indicates that it is not safe to do this change whilst serving clients.

## Dropping a Column

Dropping a column does not happen to have an Online DDL equivalent. Because `mark unused` is not the direct counterpart for drop column, performance tests have been conducted to be able to compare the two.

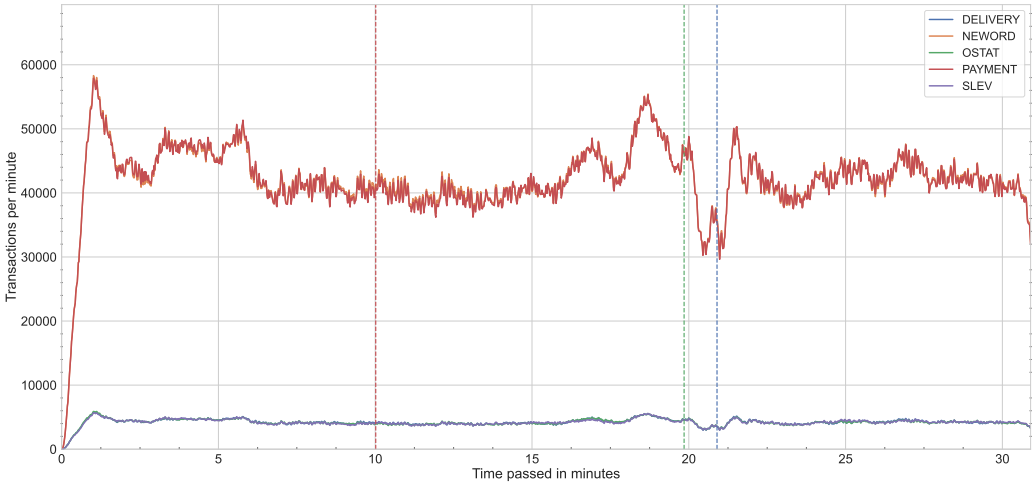



Figure 8.12: Drop Column - Throughput

The figure clearly shows a drop in terms of TPM during and after the schema change. It takes about a minute and seems to be blocking.

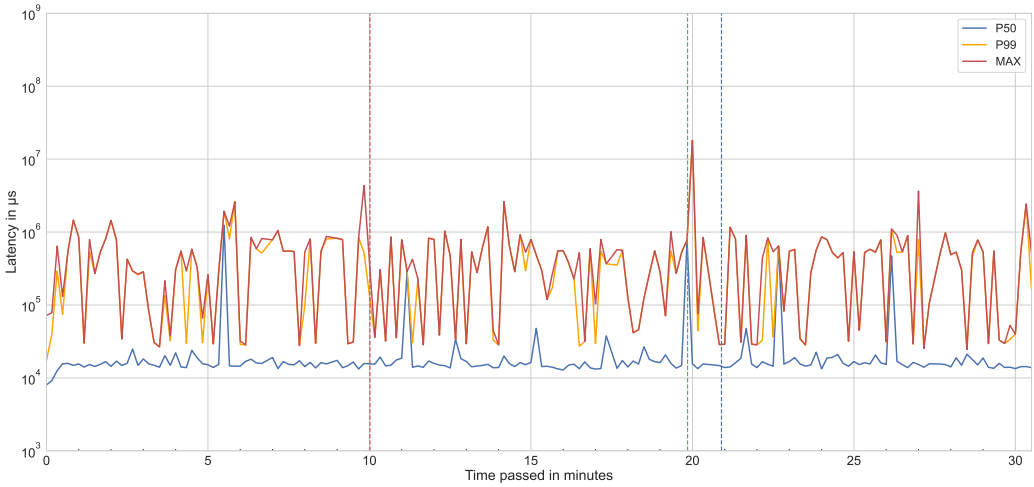



Figure 8.13: Drop Column - Latency

The figure shows a short moment where the 99th percentile and the maximum have an increase to over $10^7 \mu s$, just after starting the migration. The latency seems to be restored in a few seconds.

## Marking Column Unused

Marking a column unused is supposed to be a Meta-Data DDL statement that removes a column from the table description, without reclaiming its consumed space. This subsection is used to validate whether it is a viable non-blocking alternative to dropping a column.

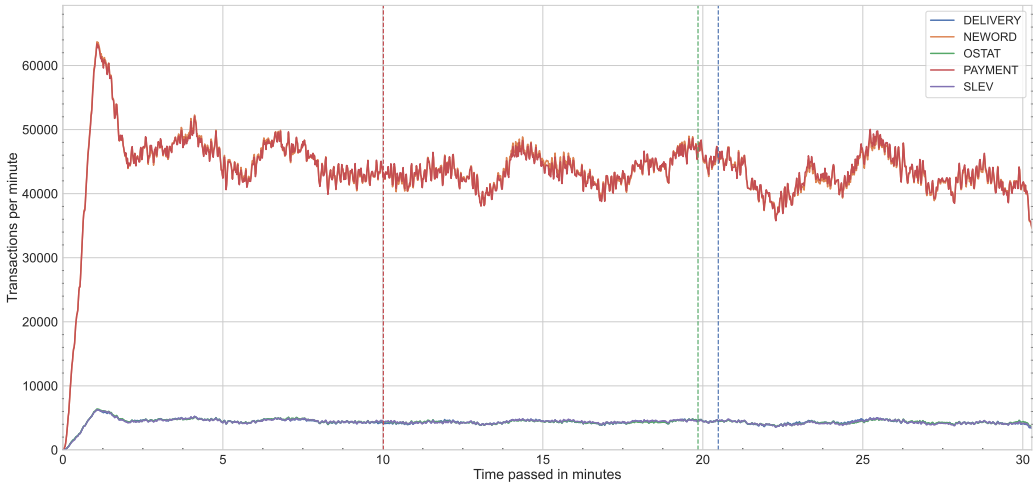



Figure 8.14: Mark Column Unused - Throughput

The figure shows no visible effect on the TPM caused by the schema change.



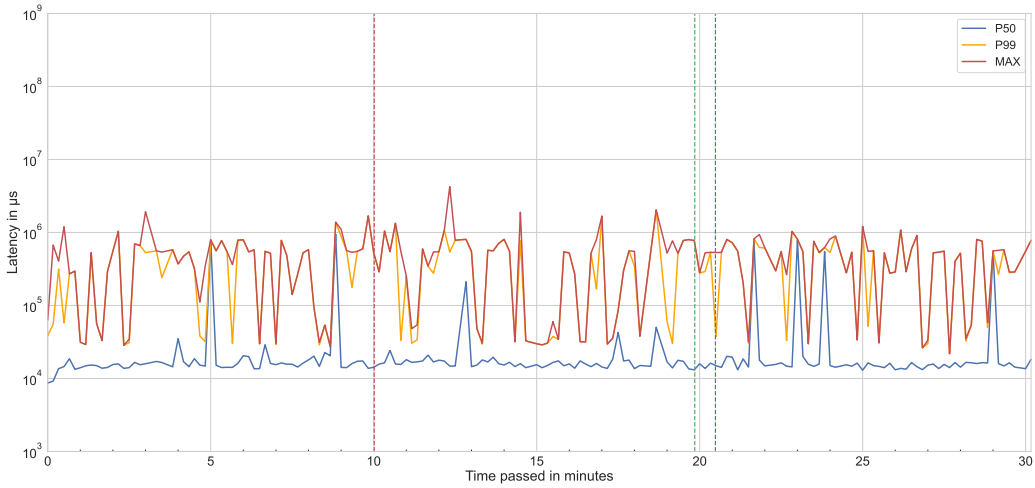



Figure 8.15: Mark Column Unused - Latency

The figure shows no visible effect on the latency caused by the schema change.

## Add Not Null Constraint

To show the effect of adding constraints, the addition of a not null constraint is taken as an example. The expectation is that an integrity check takes place, prior to propagating the constraint. Therefore, blocking of database clients during the migration is within expectations.

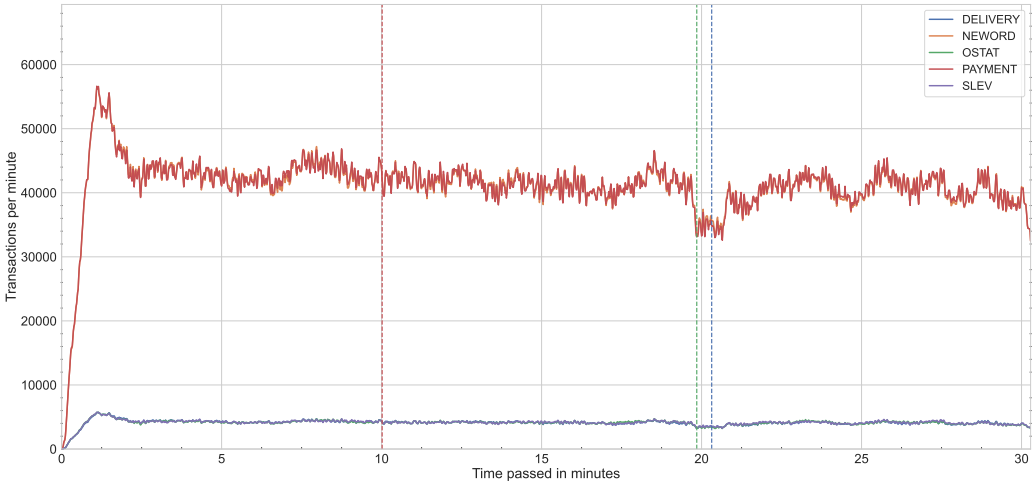



Figure 8.16: Adding a Not Null Constraint - Throughput

The chart clearly shows a drop in throughput when adding this constraint. The change is not instant due to the integrity check.

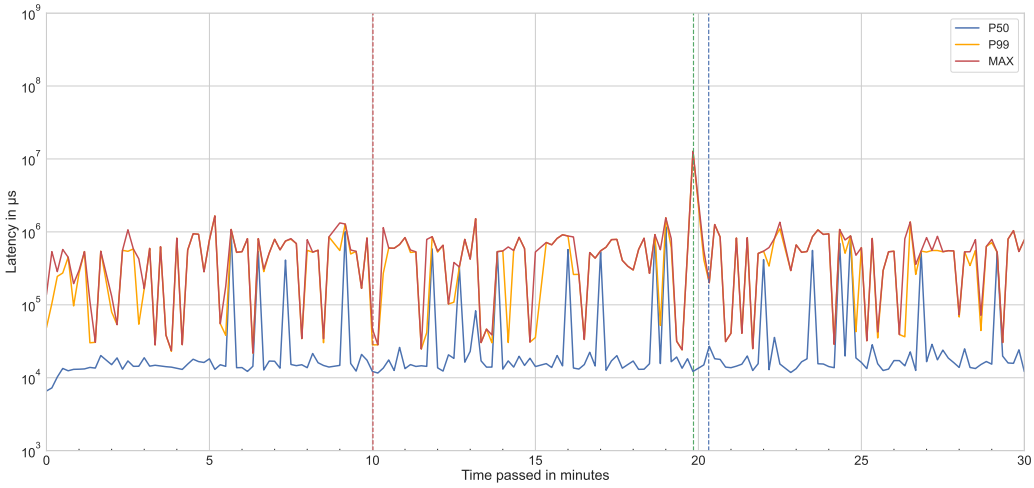



Figure 8.17: Adding a Not Null Constraint - Latency

The depicted latency seems to follow a similar pattern to the latency of dropping a column. An undesirable peak at the start of the schema change is visible.

## Add Not Null Constraint (No Validate)

The alternative to adding constraints that do initial integrity checks, is to add constraints that only validate new data (as described in the Meta-Data DDL statements section). This should not have effect on the throughput and latency of transactions during the migration.

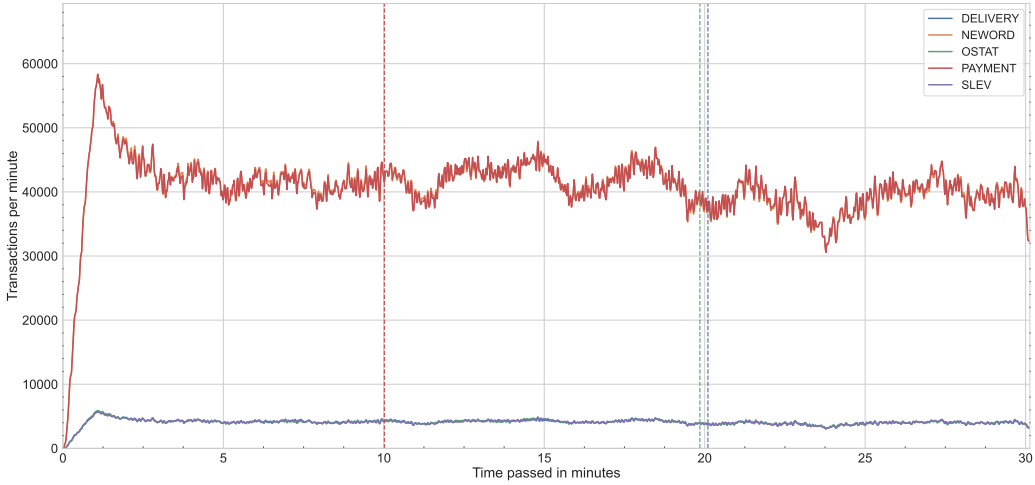



Figure 8.18: Adding a Not Null Constraint (No Validate) - Throughput

The chart does not display a direct effect of the migration on the throughput. Afterwards, the throughput slightly decreases however and has a few drops. The explanation that is most likely, is that Oracle can not optimize when it has no guarantees of all the data in the column to be not null. The change itself seems to go faster than the alternative that does integrity checks, which is logical as it does not need to do any checks. After the migration, the amount of `NEWORD` transactions that can be handled fluctuates around 40K/min. The throughput seems less stable than the blocking alternative, and it also tends to be a few thousand TPM lower.

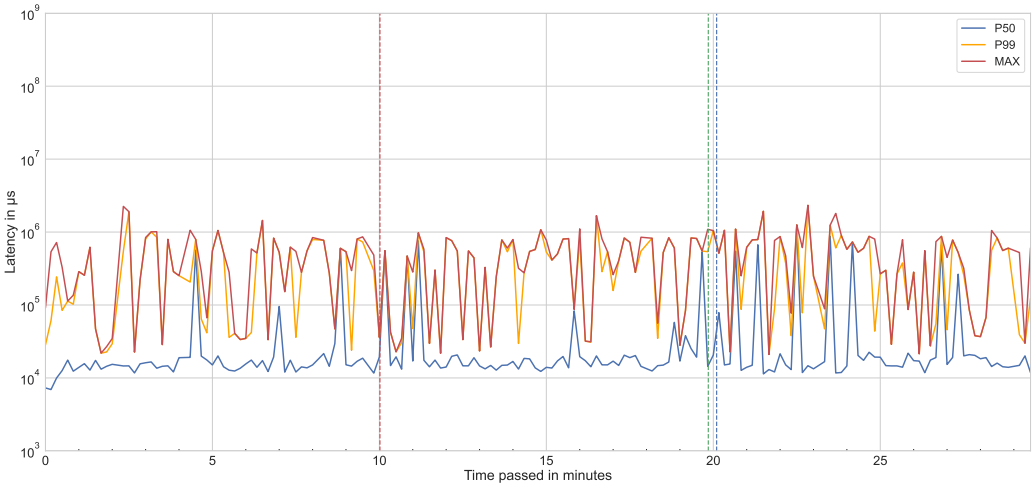



Figure 8.19: Adding a Not Null Constraint (No Validate) - Latency

The change itself does not seem to have an effect on the latency of transactions. Afterwards, there seem to be more spikes for the median latency. Those spikes encompass a latency of $\sim 10^{5,7} \mu$s which is about half a second.

### 8.3.4 Expand and Contract

This subsection provides performance tests for the solution's Expand and Contract patterns.

#### Rename Column

The pattern that was presented as a zero-downtime solution for renaming a column has been performance tested. Both the In-Place version and the Interim Table version are measured.

### Expand
### In-Place

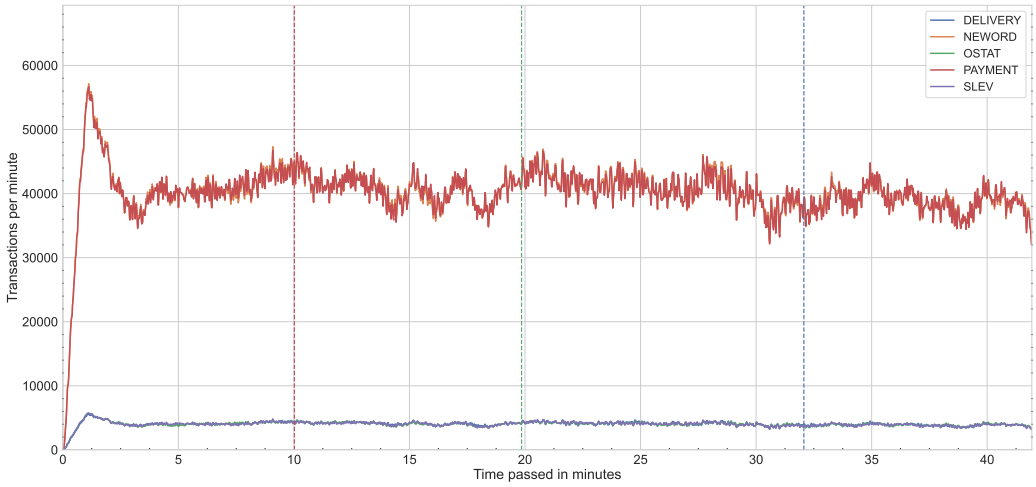



Figure 8.20: Renaming a Column In-Place (Expand) - Throughput

The chart shows that the throughput is not significantly affected by the migration, due to it being spread out. After the migration, a slight but not significant throughput decline might be visible. This could be caused by the synchronization triggers.

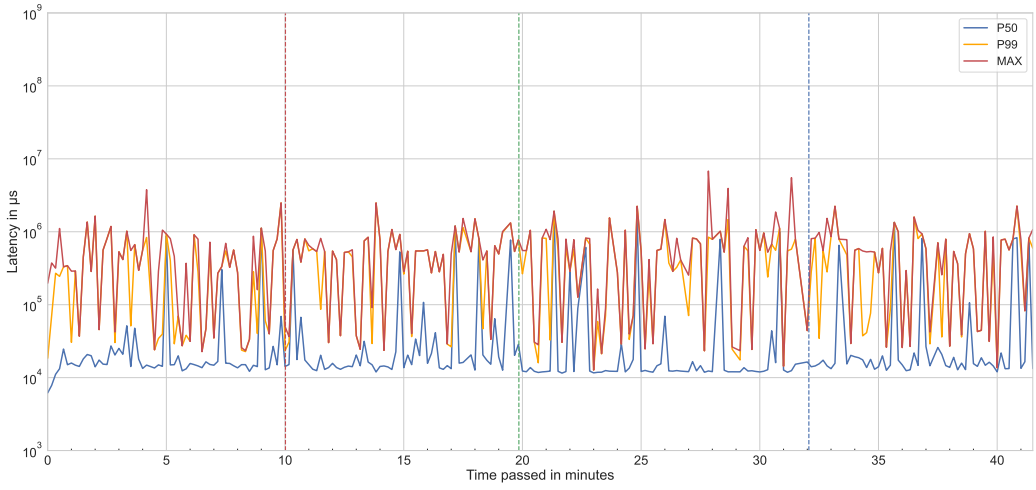



Figure 8.21: Renaming a Column In-Place (Expand) - Latency

The latency chart does not display strong deviations during the migration.

### Interim Table

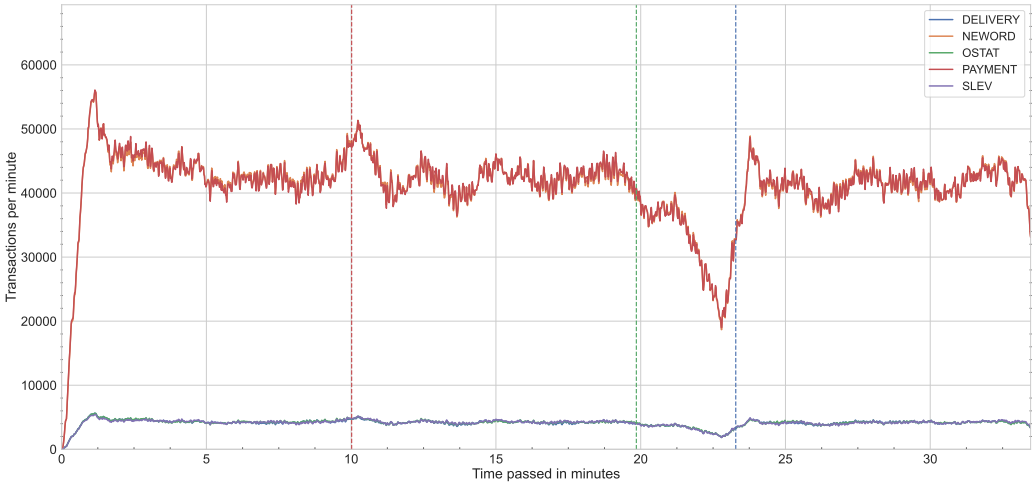



Figure 8.22: Renaming a Column using an Interim Table (Expand) - Throughput

Initially, during the migration there is no clear effect on the TPM as it follows the same trend as the baseline. At the end of the migration, it is clear that the exclusive lock results in a throughput drop. Due to the high load of the benchmark, data changes have likely occurred between the Online Redefinition synchronization step and the swapping of the tables. Therefore, the drop is significant and relatively long.



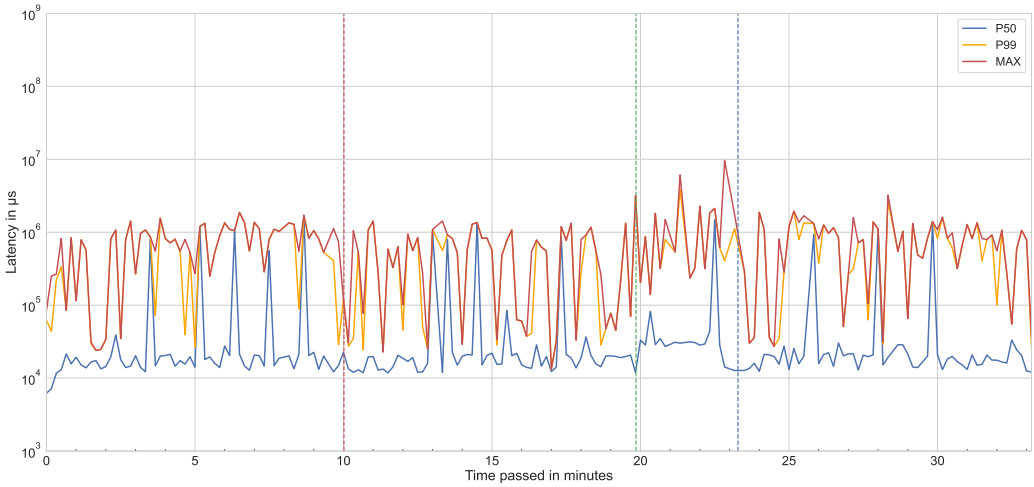



Figure 8.23: Renaming a Column using an Interim Table (Expand) - Latency

The start of the migration does not show significant differences in terms of latency. The TPM drop that was apparent in the throughput chart seems to correspond to higher latency values. This means that less transactions

are executed at the end of the online redefinition, and those that are executed have a longer latency.

## Rename Foreign Key Column

Renaming a foreign key column could not be performance tested, because HammerDB's schema does not use any foreign key constraints. There is, however, little difference between renaming a foreign key column and renaming a column. With the In-Place approach, the difference is the use of one Meta-Data DDL statement (similar to adding a not null constraint) and one Online DDL statement (to drop the original foreign key). With the Interim Table approach, there is only one difference: adding a foreign key constraint using Meta-Data DDL. As the differences only comprise Meta-Data DDL and Online DDL, it is likely that the pattern for this scenario has little performance impact.

## Rename Table

The pattern that was presented as a solution for renaming a table, has been performance tested. The `Order` table gets renamed during the migration.

**Expand**

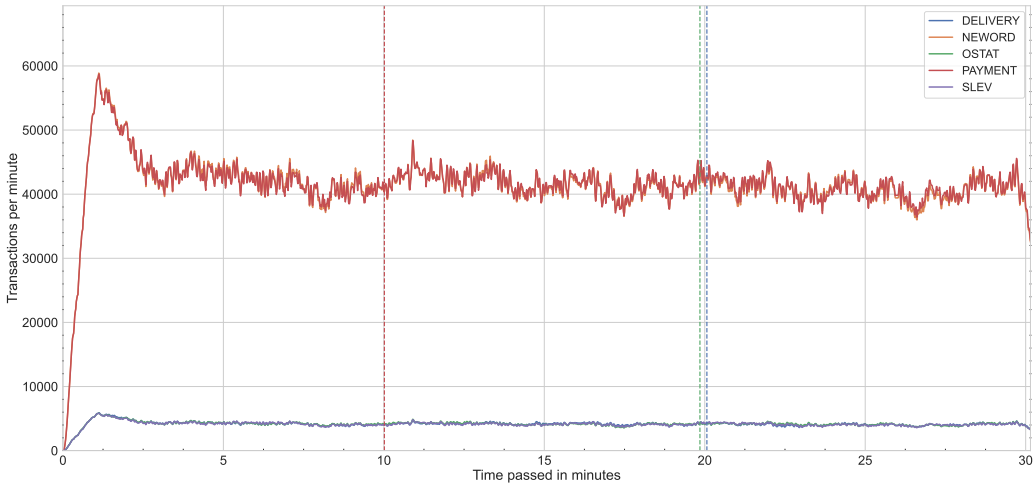



Figure 8.24: Rename Table (Expand) - Throughput

The chart does not display an observable effect of the migration on the throughput.

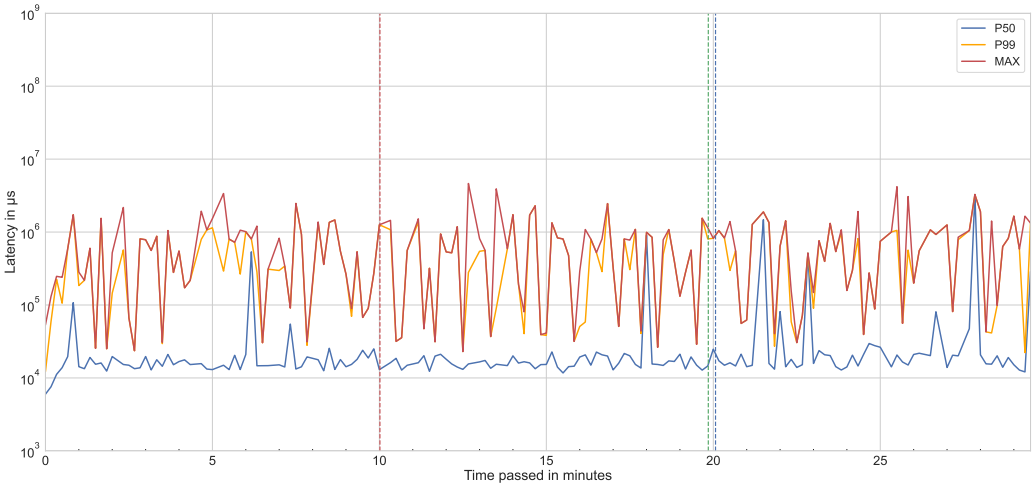



Figure 8.25: Rename Table (Expand) - Latency

The latency chart indicates that the migration has no significant effects on the latency.

### Contract

The Contract-phase was not performance tested, because no changes to actively used objects would be required in the Contract-phase: the `Order` synonym would not be used as soon as the old application version is offline.

### Rollback

The rollback for this pattern has also been benchmarked. This is to demonstrate that the running application will not be affected in terms of schema compatibility. It was not possible to test this in the Expand and Contract Patterns tests due to the design of the tests.

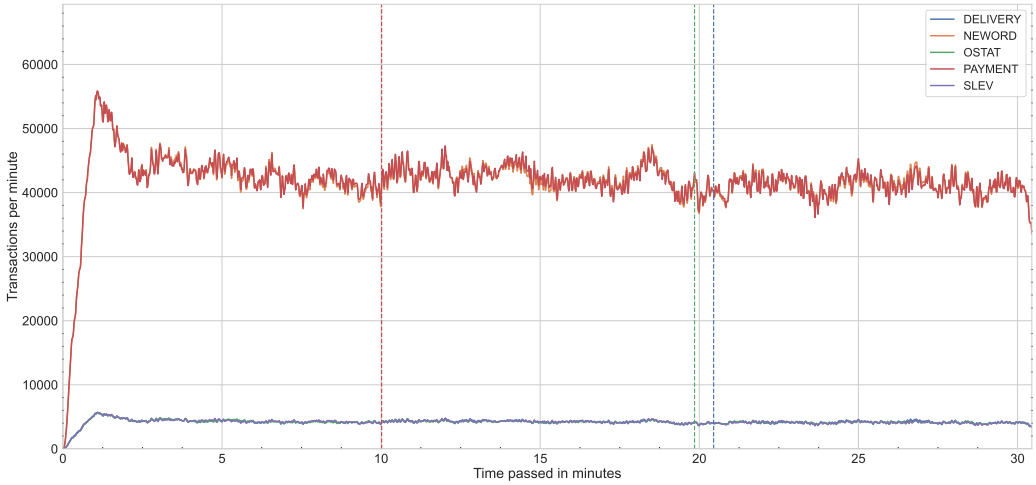



Figure 8.26: Rename Table (Rollback) - Throughput

The throughput of transactions does not seem to be affected by the rollback.

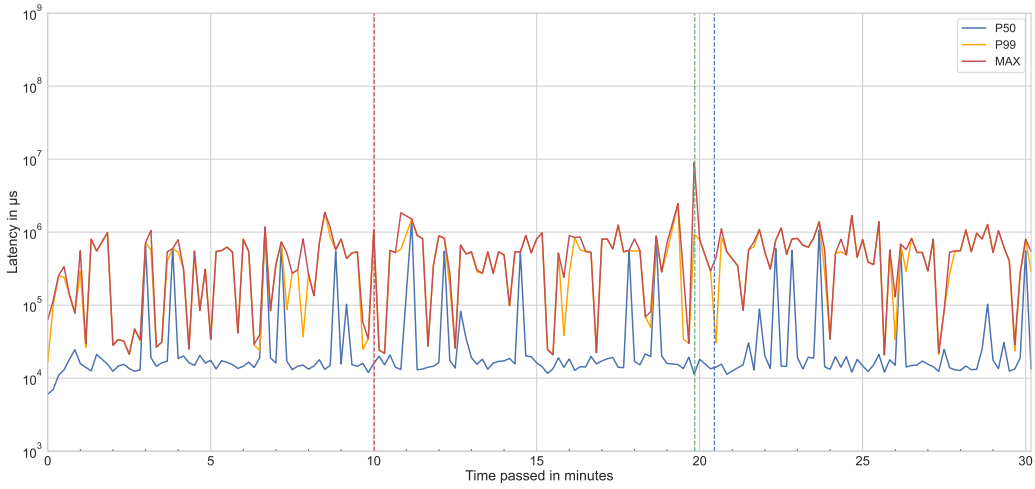



Figure 8.27: Rename Table (Rollback) - Latency

The displayed latencies also do not reflect an impact of the rollback.

# 9 Conclusion

This chapter concludes the project which has been conducted at the Digital Channels department at a FI. Releasing software frequently is difficult and comes with technical challenges. All challenges already have been addressed, except for downtime induced by database schema changes, which still lagged behind. Therefore, the goal of this project was to mitigate deployment downtime required for schema changes. Historical schema changes have been gathered to aid in exploring potential solutions. Several industrial and contextual solutions have been explored. The result of the project is a two-fold solution that leverages Online DDL and the Expand and Contract pattern. Lastly, extensive time has been spent on thorough testing of the solution.

The remainder of this chapter will be used to list the deliverables of the project. Furthermore, it reflects on the solution in correspondence to the requirements. Afterwards, the chapter reflects on common historical use cases that could now be deployed without downtime with the current solution.

## 9.1 Deliverables



During this project, the following assets were developed and are now available on the FI's version control system:

**Lbstats** a tool to automatically analyze Liquibase ChangeLogs of multiple projects and generate a frequency table. This asset has been described in Automated Results[1].

[1] This asset is publicly available at `https://github.com/abort/liquibase-stats.git`

**Liquibase-osc** a Liquibase extension to add support for Online DDL. This asset is the first part of the solution. It integrates with the current technology stack of Digital Channels. The asset has been described in the Extension section[1].

**Benchmark-docker** a modification of HammerDB to benchmark Oracle databases in combination with live schema migrations. This asset has been described in the *Performance Tests* section.

**Benchmark-baseline-script** a Python script to measure the database performance baseline using Benchmark-docker. It was used as a starting point for the *Performance Tests* section.

**Benchmark-migration-script** a Python script that runs migrations whilst measuring the database performance using Benchmark-docker. It was used for all tests in the *Performance Tests* section.

**Benchmark-processing-script** a Python script to process the results of a benchmark and generate charts. It was used for the charts in the *Performance Tests* section.

**Change-templates** a set of Expand and Contract templates that can be used to deploy without downtime. These are implementations of the presented Expand and Contract patterns and are the second part of the solution.

**Expand-contract-tests** a Spring Boot project that simulates two applications at the same time to test schema compatibility. This asset has been described in the *Expand and Contract Patterns Tests* section.

[1] This asset is publicly available at `https://github.com/abort/liquibase-osc.git`

## 9.2 Reflection on Requirements

This section will elaborate on how the solution adheres to the requirements:

**R1 Integration**: The solution integrates well with the current tools and way of working in Digital Channels. The extension integrates in the used schema migration tool (Liquibase). It can be integrated in existing applications at any time.
Most of the Online DDL and Meta-Data DDL statements were demonstrated not to affect the performance significantly. The testing demonstrated that both the extension and the Expand and Contract patterns can correctly function in a technology stack similar to what is used in Digital Channels.

The Interim Table approach provides a few integration challenges that make it less suitable than the In-Place approach for the current way of working:

- It needs contextual information. A new interim table has to be created manually, which requires developers to make sure that this interim table can be mapped to the to be changed table and vice versa. This mapping needs to be specified explicitly. Besides, the developer needs to specify what objects and properties to copy, such as constraints and table statistics. This is error-prone and also adds more information to the ChangeLog: a ChangeLog can grow quite large, even though it might only have to resemble a simple change (e.g., a rename of a column). The In-Place approach does not need this much contextual data, requiring less work from the developers and resulting in smaller ChangeLogs.
- The process requires quite some interaction. The functions that execute in the Online Redefinition package may output errors, which need to be solved prior to continuing the redefinition process. Any unsolved error could lead to the process being stuck, data loss or unintended discrepancies with the original table.
  Contrastingly, the In-Place approach does not require monitoring error variables. The required changes to achieve the Expand or Contract-phase entail DDL statements that are sim-

ilar to DDL statements that were used before. This difference implies different learning curves for both approaches.

- The complete copy of a table and its dependencies requires at least double the table space. For this approach, this significant extra space is a requirement. Next to that, the process might be time-consuming, due to the vast amount of data copying. This is an aspect to bear in mind when considering the Interim Table approach when multiple software releases are planned.
  In the In-Place approach, fewer data will be copied, which is likely to take less space and time.

**R2 Detachable**: The extension can be removed as a dependency at any point. Any new changes that will contain the `auto-online-ddl` property, will then not be rewritten to their online equivalent anymore. Removing the extension would not be necessary to prevent DDL rewrites, as disabling (or omitting) the `auto-online-ddl` would suffice.
Due to Liquibase keeping tracking of the propagated changes, any change that has already been propagated will not be executed again. This is a logical consequence of the design decision to have equal checksums for original Liquibase changes and their online equivalent. Therefore, one can stop using the extension at any time or introduce it at any time. The same goes for the Expand and Contract pattern: any pattern that has already been deployed before, will not be propagated again. Detaching the Expand and Contract part of the solution, would simply yield not using the Expand and Contract pattern for new changes.

**R3 Transparent**: The Liquibase-osc extension has been designed to be transparent such that only when the `auto-online-ddl` property is enabled explicitly, reinterpretation to Online DDL will be possible output. The to be generated SQL can be inspected prior to propagating the changes, by executing the `changelogSyncSQL` command with the Liquibase CLI. This results in developers knowing up-front what the extension will do at runtime.
The In-Place Expand and Contract patterns are combinations of typical changes provided as Liquibase ChangeLog files, such that the solution itself is visible from those changes. This is no different from how Liquibase changes are now documented. The SQL of these ChangeLogs can be inspected in the same way, using the Liquibase CLI.

The `BatchMigrationChange` has been added to the Liquibase-osc extension to keep the solution self-contained and easy. From the perspective of the Liquibase ChangeLog this might be considered a black-box solution (less transparent). It is however open source and documented. The `BatchMigrationChange` class could be moved to microservices that incorporate Liquibase, instead of the extension providing the class. A reason for this, could be the need to modify the `BatchMigrationChange` on a per-case basis. It would also increase the transparency to developers that are working on those microservices.

**R4 Generalizable**: Other RDBMSes such as MySQL provide Online DDL for common DDL statements [53]. These could be adopted in the extension by expanding the added `Generator` classes that were introduced for Online DDL statements. Using pattern matching, the `Generator` classes are designed in such a way that they can cater different outputs for different databases.
The Expand and Contract pattern is a general pattern that is unrelated to database technology. Depending on the underlying database technology and the technology stack, different synchronization methods might have to be used (e.g., if triggers do not provide the same features).

**R5 Non-Blocking Schema Changes**: The performance tests have demonstrated that Online DDL and Meta-Data DDL changes do not have a significant impact on the performance of concurrent database clients. In terms of Expand and Contract patterns, the Interim Table approach did shortly block concurrent clients during the atomic swap. This however did not result in actual downtime in these examples (as transactions were still taking place). The settings of HammerDB were tuned to measure worst case scenarios, which indicates that in a real world situation, it is unlikely that downtime will be noticeable using this approach.

**R6 Schema ChangeSets**: This requirement was set up-front, but insights during the project caused a change in perspective. Online DDL and Meta-Data DDL are composable and were demonstrated to be composable in the tests. Stitching Expand patterns (and non-related changes) together seems to add a lot of complexity, where multiple triggers might need to be merged, and their ordering could be important. Being able to test and reason about large changes is difficult and time-consuming. The solution clearly indicates that

Expand and Contract patterns are already quite complex for relatively small changes.
The goal of the project is to facilitate releasing more often, therefore stimulating small changes to be deployed rapidly. Given this perspective, large ChangeSets might be considered an anti-pattern. Literature encourages aiming for smaller changes, such that complexity can be reduced and deployments are eased [18, p. 33] [11, p. 261].

**R7 Concurrently Active Schemas**: The Expand and Contract Patterns Tests demonstrate that two schemas can be used at once without breaking semantics. The Performance Tests indicate that changes that utilize Expand and Contract with Online DDL, do not affect new transactions.

**R8 Referential Integrity**: The referential integrity has not been violated in any of the use cases. Foreign key constraints did not have to be disabled, whenever the In-Place approach was used. However, it is clear that this has been a problem for common tools, which led to this requirement [19, p. 16].
The Interim Table alternative (using Online Redefinition), temporarily disables constraints when swapping the table. However, it makes sure that during the exclusive lock, the constraints will be enabled again [36]. The exclusive lock prevents updates and inserts, such that rows can not refer to non-existent records.

**R9 Schema Isolation**: The schema is fully isolated for each version. In the discussed Expand and Contract patterns, the entity code examples indicate that each application knows no other schema than its own.

**R10 Non-Invasiveness**: The solution did not require any source code of the application to change. All that changed from the application's perspective, is the Liquibase schema migration and the dependencies (adding the Liquibase-osc extension). Custom batch migration implementations might be added to the code, without requiring changes to existing files.

**R11 Resilience**: The Expand and Contract Patterns Tests demonstrate that rollbacks can take place, whilst still serving clients on the previous version. This is a consequence of leveraging Online DDL for rollbacks as well.

## 9.3 Reflection on Change Analysis

This section reflects on types of changes with respect to requiring downtime. The typed changes of both the automated and the manual analysis (Table 5.1 and 5.2) have been merged into Table 9.1 to reflect upon required downtime. In Table 9.1, the first column represents the Liquibase change and the other columns denote the change frequency per application code (*aax*, *pam* and *irs*).

| | **aax** | **pam** | **irs** | |
|---|---|---|---|---|
| **addColumn** | 6 | 79 | 10 | 95 |
| **addForeignKey** | 0 | 38 | 0 | 38 |
| **addNotNullConstraint** | 0 | 0 | 4 | 4 |
| **addUniqueConstraint** | 0 | 32 | 0 | 32 |
| **addPrimaryKey** | 0 | 34 | 3 | 37 |
| **createIndex** | 8 | 15 | 6 | 29 |
| **createSequence** | 4 | 37 | 0 | 41 |
| **createTable** | 2 | 37 | 5 | 44 |
| **dropColumn** | 4 | 3 | 1 | 8 |
| **dropForeignKey** | 0 | 3 | 0 | 3 |
| **dropIndex** | 0 | 2 | 2 | 4 |
| **dropNotNullConstraint** | 0 | 10 | 0 | 10 |
| **dropUniqueConstraint** | 0 | 7 | 0 | 7 |
| **dropSequence** | 2 | 0 | 0 | 2 |
| **dropTable** | 1 | 4 | 1 | 6 |
| **modifyDataType** | 0 | 2 | 1 | 3 |
| **renameColumn** | 3 | 0 | 5 | 8 |
| **renameTable** | 0 | 0 | 1 | 1 |

Table 9.1: Combined Frequency Analysis

- `addColumn`: Adding a column happened most frequently. Whenever a column is added to an existing table that has data, it has to be nullable or contain a default value, because initially it will not be filled. The Performance Tests indicate that it is safe to add nullable and default columns to existing tables. Adding a column to a new table does not affect downtime, because the previous application version will be oblivious to that table.
- `addForeignKey`: Adding a foreign key constraint to an existing column can be done without downtime by disabling integrity checks

on the existing data (similar to `addNotNullConstraint`). This requires the developer to trust the existing data to already meet the constraint, or that future updates will guarantee so. New and updated data will still be checked by the constraint. This change works without downtime, because of it being Meta-Data DDL (see: the Meta-Data DDL statements section).
Adding a foreign key to a new/empty table is safe and can be done without this concession.

- `addNotNullConstraint`: The same applies as for `addForeignKey`.
- `addUniqueConstraint`: The same applies as for `addNotNullConstraint` and `addForeignKey`.
- `addPrimaryKey`: The same applies as for `addUniqueConstraint`, `addNotNullConstraint` and `addForeignKey`.
- `createIndex`: With the extension, an index can be created without blocking by leveraging Online DDL.
- `createSequence`: Create sequence creates a new schema object, which the previous application version will be oblivious to. It is therefore safe to do online.
- `createTable`: Create table creates a new schema object, which the previous application version will not be aware of. It is safe to do online.
- `dropForeignKey`: With the extension, a foreign key can be dropped without blocking by leveraging Online DDL.
- `dropIndex`: With the extension, an index can be dropped without blocking by leveraging Online DDL.
- `dropNotNullConstraint`: Columns can be modified to nullable without blocking since Oracle 12c [44]. This is possible due to the change resulting in a Meta-Data DDL statement.
- `dropUniqueConstraint`: With the extension, a unique constraint can be dropped without blocking by leveraging Online DDL.
- `dropSequence`: This can only be done online whenever the sequence is not used anymore. A deprecation period could be helpful (similar to the Expand-phase in the Expand and Contract pattern).

- `dropTable`: This can only be done online whenever the table is not used anymore. A deprecation period could be helpful (similar to the Expand-phase in the Expand and Contract pattern).

- `modifyDataType`: The 2 identified cases in PAM both entailed the expansion of the `varchar2` vector type. No information is lost in such case, and this change showed little performance impact in the Performance Tests. The previous application version could still continue to occupy the smaller amount of space. The effect of having more data in a column than the previous version could handle is case specific.
  The single case in the **irs** project has not been identified.

- `renameColumn`: For renaming a column, this project presents viable solutions that leverage Expand and Contract. It depends on the use case whether the current patterns suffice, or whether they need to be adjusted (e.g., in case the column is part of a composite foreign key). In case of the latter, the solution and testing approach can be considered as a framework to expand upon.

- `renameTable`: For renaming a table, this project presents a zero-downtime solution that leverages the Expand and Contract pattern. The applicability of the solution, depends on whether the use of public synonyms is an option for the concerning application.

Historical releases often contained multiple changes at once, of which most of them seem to be non-breaking (additional). Renaming or modifying of existing columns did not take place often. The Expand and Contract pattern can be avoided most of the time in case this trend continues. The composition of non-breaking changes, leveraging Online DDL and Meta-Data DDL, should allow engineers to do these structural changes without downtime.
The lack of knowledge of the evolution of data per release has been a limitation for the validation of the solution. The Recommendations chapter suggests improvements regarding data management, such that data evolution will be more visible in the future. To counter this limitation, the Expand and Contract Patterns Tests test both with and without starting data. Secondly, the Performance Tests propagate schema changes on a populated and actively used database.

# 10 Recommendations

This chapter serves as a bundle of recommendations, such that the solution can be applied in practice. The future work will also be laid out.

The solution can directly be adopted, which is a result of the strong focus to integrate well with the current department tooling. Introducing Liquibase-osc to an existing or new project only requires adding it as a dependency. Any new ChangeSets that has the `auto-rewrite-ddl` property enabled, will result in Online DDL where possible.
Secondly, the Expand and Contract patterns can also be applied as of now. The In-Place approach is easier to get started with due to it not requiring intervention. It is desirable to do conduct extra tests to ensure the implementations of the Expand and Contract patterns fit the concerning application. This can be done by adopting the test architecture depicted in Figure 8.1 in the concerning application.

## 10.1 Testing

### 10.1.1 Integration Tests

It is a common practice to use in-memory databases in integration tests. The tests initiate the in-memory database and its schema prior to running the test methods. In-memory databases often have limited features and might not be able to cater for synchronization using triggers. Because these tests are executed on a single version, it is fine to run blocking DDL statements and to neglect schema compatibility. A Liquibase ChangeSet can provide an alternative approach for integration tests. The Expand-phase of the rename table scenario, will then look as follows:

```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-3.10.xsd">

    <property name="auto-online-ddl" value="true"/>

    <changeSet id="rename_table_oracle" author="jorryt"
       dbms="oracle">
      <!-- Oracle expand pattern goes here -->
    </changeSet>

    <changeSet id="rename_table_non_oracle" author="
      jorryt" dbms="!oracle">
      <renameTable oldTableName="old_table"
        newTableName="new_table" />
    </changeSet>
</databaseChangeLog>
```

For the Contract-phase the same conditional logic applies: only when it is Oracle, cleaning up has to happen.

## 10.2 Maintenance



It is important to point out that schema migrations are not database maintenance work. These migrations are often a result of application maintenance, where new requirements lead to a change in application code and its corresponding schema. The database however also needs to be maintained in order to keep performance up and to prevent using more space than needed. These type of maintenance activities should not be part of the Liquibase scripts. Liquibase does not have native types for those, which is a clear indication that this is a separate concern.
The rest of this section discusses some recommendations on database maintenance work.

### 10.2.1 Reclaiming Disk Space

Some migration strategies result in additional maintenance work. Any change that uses Online Redefinition, requires to drop the original table after it has been exclusively swapped with the interim table.
Besides, disk space occupied by unused columns as a result of `mark unused` remains reserved. Oracle keeps track of these unused columns. To get insights in unused columns, one can query the `USER_UNUSED_COL_TABS` table. There are two explicit ways to reclaim this space:

1. By executing a DDL statement to drop the unused columns:

   ```
   ALTER TABLE `my_table` DROP UNUSED COLUMNS;
   ```

   This can take a long of time, especially when the column used to have a lot of data. This can be optimized by increasing the `checkpoint` size to cut down the amount of undo logs and to prevent undo space exhaustion [54].

2. By replacing the table using Online Redefinition (by excluding the unused column from the mapping). The swap will cause the original table to be out of use, such that it can be safely deleted afterwards.

Note that reclaiming unused column space is implicitly triggered whenever any column is dropped in the same table [54].

### 10.2.2 Other Maintenance Activities

Some database maintenance can be done without downtime these days. For each task that needs to be done, it might be worthwhile to use an Online DDL alternative instead. Examples of activities that can be done using Online DDL are [54][55]:

- Creating/Rebuilding an index
- Moving a table, partition or subpartition to a different tablespace
- Changing storage parameters of a table
- Toggling compression on a table

Alternatively, Online Redefinition can be used for some of these activities. Note that some Online DDL constructs and Online Redefinition scenarios have limitations. The Oracle documentation states all the details.

## 10.3 Deployment Pipeline

It is worthwhile to read the book "Continuous Delivery" by Humble et al. The paragraph "*Decoupling Application Deployment from Database Migration*" in the chapter "*Rolling Back Databases and Zero-Downtime Releases*" describes what the Expand and Contract strategies do.

### 10.3.1 Propagating Schema Changes

Even though Spring Boot can propagate Liquibase changes, it is a good practice to have a dedicated step for the propagation of schema changes in the deployment pipeline. This separates the application and schema deployment. It makes it easier to go through logs and track down problems. It also can help in preventing deadlocks in Liquibase, which is not uncommon (related to what was mentioned in the Version Tracking section)[1]. The cause of this is a race condition, which may surface when multiple instances of the application try to change the schema simultaneously.

## 10.4 Interactive Online Redefinition



Due to the interactive nature of Online Redefinition and the gained trust in Oracle tooling, the department might be more comfortable to use the Online Redefinition package directly on the database without Liquibase in between. The interim tables would not affect the current running schema until the final swap takes place.
To facilitate this, one can adopt the offline version of those changes in Liquibase ChangeLogs (similar to what was done for Integration Tests). In such ChangeLog, there would be no need to have a condition for database type. For the rename column example, the Liquibase ChangeLog using this strategy would look as follows:

[1] See https://github.com/liquibase/liquibase issues 1036 and 1584

```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-3.10.xsd">
  <changeSet id="rename_column" author="jorryt">
    <renameColumn tableName="customer"
      oldColumnName="name"
      newColumnName="full_name" />
  </changeSet>
</databaseChangeLog>
```

This change should not be run using Liquibase, as this change is then done directly by the database administrator. After successfully swapping the tables, the Liquibase change can be considered obsolete. The Liquibase change can be marked as ran by running the Liquibase CLI and providing the `markNextChangeSetRan` command as an argument. This makes sure that the `rename_column` change will never be executed on this database, but it is documented whenever the application has to run on a new environment or whenever integration tests have to run. The Liquibase ChangeLog has to be semantically equivalent to what the Expand and Contract pattern breaks up in two steps.
The Contract-phase should also be done directly on the database.

The above paragraph states how Online Redefinition can be used interactively, whilst keeping the Liquibase history relevant and up to date. This bridges the gap between the declarative use of Liquibase and Online Redefinition. The downside is that this requires Online Redefinition changes to be semantically equivalent to Liquibase ChangeLogs.



## 10.5 Configuration & Data Management

This section discusses some recommendations on both configuration management and data management.

### 10.5.1 Application Configuration

The application configuration of PAM is currently stored in the database. This observation was mentioned in the Manual Findings section. The configuration is currently not versioned, and any change to this table might affect previous running application versions. The Continuous Delivery book by Humble et al. provides best practices in Chapter 2, which can help to overcome this.

### 10.5.2 Oracle Compatibility

It is important to make sure the Oracle compatibility parameter is set to the highest version possible. This allows newer optimizations to be leveraged, which in some cases makes a lot of difference in terms of performance[1].

### 10.5.3 Incremental Data

Database changes should be incremental [5, p. 327]. Currently, the data changes might be incremental, but the scripts do not reflect whether data is new or not (due to the idempotency of upserts/merge-statements). It would be more transparent and easier to interpret when only the actual changes would be stored as Liquibase scripts.

## 10.6 Future Work



### 10.6.1 Building New Patterns

The book "Refactoring Databases: Evolutionary Database Design" by Amber et al. describes several change patterns that preserve schema compatibility. The patterns are described with Oracle SQL examples. The book does not however take zero-downtime in mind, such that it provides no guarantees on changes being non-blocking. Besides, it does not address

[1] `https://hourim.wordpress.com/2018/03/20/ddl-optimisation-is-not-working-in-12cr2-really/`

the fact that in between DDL statements, integrity constraints might be violated. Modifying those patterns to use Online DDL, Meta-DDL or Online Redefinition, could convert them to zero-downtime alternatives.

## 10.6.2 Extending Proposed Patterns

This subsection describes suggestions on how to extend proposed patterns. These were out of scope in the project and still need to be thoroughly tested.

### Rename Composite Foreign Key Column

It is possible that the to be renamed column is part of a composite foreign key. The same steps would apply as in the scenario of renaming a single foreign key column. The difference is in the addition of the foreign key constraint. Let's consider the scenario where column `A` needs to be renamed to `D`, in which `A` is part of the foreign key constraint: $\langle A, B, C \rangle$. In such scenario, the newly added foreign key constraint should encompass the triple $\langle D, B, C \rangle$.

### Modify Data Type

Whenever a data type has to be modified (other than extension as shown for `varchar2`), it is likely that previous application versions can not deal with the new type, resulting in schema incompatibility. The recommendation is to instruct developers to add a new column for the new data type instead. Then the Expand and Contract pattern can be used similarly as to what was done for renaming a column. The difference would be that the triggers that synchronize the old and new column need to take care of the type conversion.

### 10.6.3 Batch Migrations

#### Cross-Table

In some scenarios, the `BatchMigrationChange` might also need to migrate data from one table to another. This is currently not implemented, but could be an added feature. Whenever batch migration to another table is done online, it is likely that triggers are also needed, to make sure that new and changed data is also reflected to the other table.

#### Auto-Scaling

Currently, the `BatchMigrationChange` operates based on a fixed chunk size. Whenever it tries to update rows that are being modified by another transaction, it might be that a deadlock occurs. Currently, no strategy is implemented to overcome that, demanding that chunk sizes are granular enough not to make this occur. A good approach to balance the time it takes and the amount of items it locks during high load, is to implement a mechanism that can find the optimal chunk size.

### 10.6.4 Performance Tests

HammerDB is not designed to do migrations whilst measuring performance. The presented solution does not facilitate to test this on custom/own schemas. A solution that is more maintainable and customizable could be built on top of OLTPBenchmark, an open source benchmarking framework[1].



[1] `https://github.com/oltpbenchmark/oltpbench`

# References


[1] E. Union, "Directive (eu) 2015/2366 of the european parliament and of the council," vol. 58, no. L337, Dec. 2015, ISSN: 1977-0677. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2015:337:TOC.

[2] M. Noctor, "Psd2: Is the banking industry prepared?" *Computer Fraud & Security*, vol. 2018, no. 6, pp. 9–11, 2018, ISSN: 1361-3723. DOI: https://doi.org/10.1016/S1361-3723(18)30053-8. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1361372318300538.

[3] M. Fowler, J. Highsmith, *et al.*, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.

[4] R. Shaydulin and J. Sybrandt, *To agile, or not to agile: A comparison of software development methodologies*, 2017. arXiv: 1704.07469 [cs.SE].

[5] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[6] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond continuous delivery: An empirical investigation of continuous deployment challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 111–120. DOI: 10.1109/ESEM.2017.18.

[7] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[8] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Dimensions of devops," in *Agile Processes in Software Engineering and Extreme Programming*, C. Lassenius, T. Dingsøyr, and M. Paasivaara, Eds., Cham: Springer International Publishing, 2015, pp. 212–217, ISBN: 978-3-319-18612-2.

[9] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*, 1st. IT Revolution Press, 2018, ISBN: 1942788339.

[10] J.-P. Arcangeli, R. Boujbel, and S. Leriche, "Automatic deployment of distributed software systems: Definitions and state of the art," *Journal of Systems and Software*, vol. 103, pp. 198–218, 2015, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.01.040. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215000308.

[11] M. T. Nygard, *Release it! design and deploy production-ready software*, Second edition, ser. The pragmatic programmers. Raleigh, North Carolina: Pragmatic Bookshelf, 2018, OCLC: ocn982532892, ISBN: 9781680502398.

[12] M. Kleppmann, *Designing data-intensive applications*, 2015.

[13] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker, "Interface evolution patterns: Balancing compatibility and extensibility across service life cycles," in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, 2019, pp. 1–24.

[14] S. Raemaekers, A. Van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2014, pp. 215–224.

[15] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 7th ed., ser. Always learning. Pearson, 2016, ISBN: 9781292097619. [Online]. Available: https://books.google.nl/books?id=3z4KrgEACAAJ.

[16] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed., ser. Always Learning. Pearson Education Limited, 2013, ISBN: 9781292024479. [Online]. Available: https://books.google.nl/books?id=8rzLngEACAAJ.

[17] C. Coronel and S. Morris, *Database Systems: Design, Implementation, & Management*, 13th. Cengage Learning, 2018, ISBN: 9781337627900. [Online]. Available: https://books.google.nl/books?id=j69EDwAAQBAJ.

[18] S. Ambler and P. Sadalage, *Refactoring Databases: Evolutionary Database Design*, ser. Addison-Wesley signature series. Addison Wesley, 2006, ISBN: 9780321293534. [Online]. Available: https://books.google.nl/books?id=puBQAAAAMAAJ.

[19] M. d. Jong and A. v. Deursen, "Continuous deployment and schema evolution in sql databases," in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 2015, pp. 16–19.

[20] M. de Jong, A. van Deursen, and A. Cleve, "Zero-downtime sql database schema evolution for continuous deployment," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 143–152, ISBN: 9781538627174. DOI: 10.1109/ICSE-SEIP.2017.5. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.5.

[21] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, Jan. 2018, ISSN: 1559-6915. DOI: 10.1145/3183628.3183631. [Online]. Available: https://doi.org/10.1145/3183628.3183631.

[22] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. "O'Reilly Media, Inc.", 2016.

[23] C. Richardson, *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019, OCLC: on1002834182, ISBN: 9781617294549.

[24] JetBrains, *Calling java from kotlin | kotlin*, https://kotlinlang.org/docs/java-interop.html#using-jni-with-kotlin, (Accessed on 04/21/2021), 2021.

[25] Liquibase, *Liquibase online docs*, https://docs.liquibase.com/home.html, (Accessed on 12/10/2020).

[26] Oracle, *Types of sql statements*, https://docs.oracle.com/database/121/SQLRF/statements_1001.htm#SQLRF30001, (Accessed on 12/08/2020).

[27] Liquibase, *Liquibase docs changeset*, https://docs.liquibase.com/concepts/basic/changeset.html, (Accessed on 12/11/2020).

[28] Oracle, *What is a jpa entity?* https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm, (Accessed on 05/13/2021).

[29] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, 2020, pp. 150–153. DOI: 10.1109/MEMSTECH49584.2020.9109514.

[30] M. Fowler, *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*, ser. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012, ISBN: 9780133065213. [Online]. Available: https://books.google.nl/books?id=vqTfNFDzzdIC.

[31] Google, *Google cloud devops tech: Database change management*, https://cloud.google.com/solutions/devops/devops-tech-database-change-management, (Accessed on 12/08/2020).

[32] GitHub, *Triggerless design*, https://github.com/github/gh-ost/blob/master/doc/triggerless-design.md, (Accessed on 05/17/2021).

[33] M. Fowler, *Refactoring: Improving the Design of Existing Code*, ser. A Martin Fowler signature book. Addison-Wesley, 2019, ISBN: 9780134757599. [Online]. Available: https://books.google.nl/books?id=o69NtAEACAAJ.

[34] T. Kyte and D. Kuhn, *Oracle Database Transactions and Locking Revealed*. Apress, 2014.

[35] A. Choi, "Online application upgrade using edition-based redefinition," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '09, Orlando, Florida: Association for Computing Machinery, 2009, ISBN: 9781605587233. DOI: 10.1145/1656437.1656443. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/1656437.1656443.

[36] Oracle, *Dbms_redefinition*, https://docs.oracle.com/database/121/ARPLS/d_redefi.htm#ARPLS67525, (Accessed on 04/23/2021).

[37] M. de Jong, "Zero-downtime sql database schema evolution for continuous deployment," M.S. thesis, TU Delft, Sep. 2015. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%5C%3Aaf89f8ba-fc34-4084-b479-154be397718f.

[38] Oracle, *Data concurrency and consistency*, https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-concurrency-and-consistency.html#GUID-E8CBA9C5-58E3-460F-A82A-850E0152E95C, (Accessed on 04/23/2021).

[39] M. Overeem, M. Spoor, and S. Jansen, "The dark side of event sourcing: Managing data conversion," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 193–204. DOI: 10.1109/SANER.2017.7884621.

[40] M. Overeem, M. Spoor, S. Jansen, and S. Brinkkemper, "An empirical characterization of event sourced systems and their schema evolution—lessons from industry," *Journal of Systems and Software*, p. 110 970, 2021.

[41] Oracle, *Edition-based redefinition*, https://docs.oracle.com/cd/E11882_01/appdev.112/e41502/adfns_editions.htm#ADFNS99917, (Accessed on 04/23/2021).

[42] Liquibase, *Overview - liquibase extensions - liquibase*, https://liquibase.jira.com/wiki/spaces/CONTRIB/pages/2424879/Overview, (Accessed on 05/13/2021), Dec. 2019.

[43] Oracle, *Managing tables*, https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-tables.html#GUID-74A86E52-E2D2-405E-B888-94164E3973B9, (Accessed on 05/16/2021), 2021.

[44] ——, *Oracle database 12c release 1 (12.1.0.1) new features*, https://docs.oracle.com/database/121/NEWFT/chapter12101.htm#NEWFT280, (Accessed on 05/24/2021), 2021.

[45] ——, *Data integrity*, https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-integrity.html#GUID-97D4BB23-FD4B-4FB6-BE21-E5F8C43BD94F, (Accessed on 05/14/2021).

[46] ——, *Automatic locks in dml operations*, https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Automatic-Locks-in-DML-Operations.html#GUID-3D57596F-8B73-4C80-8F4D-79A12F781EFD, (Accessed on 05/14/2021).

[47] ——, *Schema object dependencies*, https://docs.oracle.com/cd/B28359_01/server.111/b28318/dependencies.htm#CNCPT1869, (Accessed on 12/15/2020).

[48] M. Papadakis and K. Sagonas, "A proper integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '11, Tokyo, Japan: Association for Computing Machinery, 2011, pp. 39–50, ISBN: 9781450308595. DOI: 10.1145/2034654.2034663. [Online]. Available: https://doi.org/10.1145/2034654.2034663.

[49] Liquibase, *Updatesql command | liquibase docs*, https://docsstage.liquibase.com/commands/community/updatesql.html, (Accessed on 05/14/2021), 2021.

[50] L. Wevers, M. Hofstra, M. Tammens, M. Huisman, and M. van Keulen, "Analysis of the blocking behaviour of schema transformations in relational database systems," in *Advances in Databases and Information Systems*, M. Tadeusz, P. Valduriez, and L. Bellatreche, Eds., Cham: Springer International Publishing, 2015, pp. 169–183, ISBN: 978-3-319-23135-8.

[51] HammerDB, *Configuring driver script options*, `https://www.hammerdb.com/docs/ch04s05.html`, (Accessed on 05/16/2021), 2021.

[52] ——, *Understanding the tproc-c workload derived from tpc-c*, `https://www.hammerdb.com/docs/ch03s05.html`, (Accessed on 05/19/2021), 2021.

[53] M. O. DDL, *Mysql :: Mysql 5.7 reference manual :: 14.13.1 online ddl operations*, `https://dev.mysql.com/doc/refman/5.7/en/innodb-online-ddl-operations.html`, (Accessed on 05/15/2021), 2021.

[54] Oracle, *Altering tables*, `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/ALTER-TABLE.htm#GUID-552E7373-BF93-477D-9DA3-B2C9386F2877`, (Accessed on 05/18/2021), 2021.

[55] ——, *Managing indexes*, `https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-indexes.html#GUID-BD4B0F57-6E2A-4A73-A0A7-99AB1F69F142`, (Accessed on 05/18/2021), 2021.