DMB

DATABASE MANAGEMENT
AND
BIOMETRICS

.06871

ZERO DOWNTIME SCHEMA MIGRATIONS IN HIGHLY AVAILABLE DATABASES

Coen van Kampen

MASTER'S ASSIGNMENT

Committee:
Maurice van Keulen
Tim Soethout (ING)
Kevin van der Vlist (ING)

June, 2022

2022DMB0003
Data Management and Biometrics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Abstract

Nowadays, online services are becoming more important in every day life. Some services are so important that they require to be online at all times. They are expected to have (near) zero-downtime. Downtime can either be planned for or unplanned for. To account for unplanned downtime, services can be made highly available by replicating the service across multiple nodes and balance the workload. If a node crashes or the service errors, the remaining nodes will take over their workload. Planned downtime however is difficult to diminish. It is a result of deploying a new software version. To have zero-downtime, both the old and new application versions have to live beside each other. Blue-green deployments make use of this practice to ensure zero-downtime in a stateless application.

However zero-downtime deployments are not as straightforward for stateful applications. To migrate a database schema along the application deployment, another strategy is required. This research is based around the expand-contract pattern that can be used in combination with the blue-green deployments. In earlier work from Dijkstra, he proposes the use of this pattern and applies it to an Oracle database using the Liquibase versioning tool. In this research, the pattern has been applied to a Postgres database and a Liquibase plugin was implemented to simplify deployments. In addition, the pattern is tested on a data-intensive application that aims for high availability by replicating the database. A benchmarking tool has been used to measure the latency and throughput of the database. From the results, it can be concluded that the expand-contract pattern provides zero-downtime.

Acknowledgements

Before I started this research, I did not have a topic in mind myself. I started looking for topics on the university website under open master thesis assignments. There were several assignments that caught my eye. After weighing the pros and cons of each of them, I concluded that the topic that most suits my knowledge and interest is about zero-downtime schema migrations at ING. Unfortunately, I was notified that ING was not able to provide a workspace for me to do my research. Nevertheless, that did not change my mind about my research topic. Even throughout the research, I never lost interest in the topic.

Throughout my research, I had weekly meetings with my supervisor Maurice van Keulen, and Tim Soethout and Kevin van der Vlist from ING. Overall, we were always able to find a date for a meeting. Sometimes we were not able to meet with the whole group, but that did not add any difficulties to my research. During these meetings, I was able to tell how my research was going and ask questions when necessary. However, the reason I especially felt these meetings were important to me, is because they provided me with motivation to continue my research. Even if I still had questions after the meeting, I was always able to contact Tim and Kevin, and quickly receive a reaction. I would like to thank Maurice, Tim and Kevin for their effort in providing me with support for my research. In the end, my research would not have been where it is without their help.

Finally, I would like to thank Jorryt Dijkstra, since his research provided the perfect direction for mine. I was able to use a lot of concepts from his paper in my implementation which would not have been possible without his efforts.

Contents

Glossary

availability A database is available if it is online and can be accessed by the user.

blue-green deployment A blue-green deployment is an application deployment technique that enables developers to install a new application version and transition users to instances of the new application without having them experience downtime.

consistency A database must always be in a consistent state. Data integrity constraints, such as data types or nullability, cannot be violated.

convergence Convergence is used in the context of distributed systems and means that each replicated instance of an object returns the same last updated value.

data-intensive application "We call an application data-intensive if data is its primary challenge the quantity of data, the complexity of data, or the speed at which it is changing as opposed to compute-intensive, where CPU cycles are the bottleneck." [22].

database A database is a structured collection of data that is commonly stored on a server.

deployment Deployment is the process that installs a software unit on a server and exposes it for use.

distributed system "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system." [31].

downtime Downtime is the amount of time that an application is offline or unavailable to its users.

eventual convergence Eventual convergence is used in the context of distributed systems and means that, provided no additional updates are made, each replicated instance of an object will eventually have the same last updated value.

fault tolerance A fault tolerant system is able to resist failure of a component and continue operating.

full replication Full replication is one of the possible database replication techniques. It involves replicating all available data in a database to another one. This is different from sharding where only parts are replicated.

high availability High availability requirements describe guarantees about the availability of a service. Highly available services aim for a small amount of downtime.

latency Latency describes the time it takes to send a request for a query, process it and send back a response.

linearizability Linearizability guarantees that a write operation to a single object is instantly visible to any later (in real-time) read operation of the same object.

load balancer A load balancer is a node that efficiently distributes the incoming requests across multiple servers running the same application or database instance.

microservice A microservice is a software unit that encapsulates the functionality for a single business capability. Microservices are small, independent and loosely coupled.

mixed-state A database is in a mixed-state if it supports two distinct schema versions.

one-copy serializability A non-serial schedule of concurrent transactions across a replicated database system is one-copy serializable if the result after execution is equivalent to the result of a serial schedule on a single database instance.

replication Database replication is the process of continuously replicating data from one database to another in order to keep both databases in sync.

schema A schema describes the logical structure of the database.

schema migration A schema migration is a change operation that updates the logical structure of the database.

serializability A non-serial schedule of database transactions is serializable if the result is equivalent to the result of a serial schedule of the corresponding transactions.

snapshot isolation Once a transaction starts, it takes a snapshot of the current data and only operates on the snapshot. Read operations are only able to see changes that were applied before the start of the transaction. Write operations are only applied to the snapshot until the transaction is committed.

strict serializability The combination of serializability and linearizability.

synchronization Synchronization is the process of keeping data between two data stores equivalent to ensure consistency. Changes on one end are automatically updated on the other end and vice versa.

throughput The throughput of an application defines how many transactions are executed within a unit of time.

transaction A transaction is a group of database operations that either executes as a whole or does not execute at all. In case of failure, all operations will be rolled back.

zero-downtime Zero-downtime describes that a construct does not entail any downtime that is perceived by the user as a change in latency or throughput.

Chapter 1

Introduction

1.1 Motivation

Traditional applications often used a monolithic architecture. These resulted in vast mainframe applications. The advantage of these applications is that they were straightforward to set up. Unfortunately, they were also hard to manage and not reusable. Additionally, a fault lead to failure of the entire system. In the meantime, several new application architectures have been designed to deal with these problems.

Nowadays, microservices architectures are becoming more and more popular. microservices architectures are better able to fulfill the requirements that stakeholders ask for. Applications that make use of this architecture are often more fault tolerant, maintainable, scalable and highly modular. Consequently, they produce less downtime. The latter requirement is the focus of this research. The advantage of (nearly) zero-downtime is that users do not observe any interruptions. This is less frustrating for the user and results in a better user experience overall, and therefore more trust in the company. Downtime can be unplanned for, in the case of a server crash, and therefore requires a resilient system. High availability clusters add redundancy, in the form of additional standby servers, to quickly recover from a failure. It is also possible to provide high availability in a database cluster. This is done through replication: a process that creates a snapshot of the database on a separate server and synchronizes the data in both instances.

On the other hand, there is also planned downtime. This happens when application features are updated or added and the new application code has to be deployed. For a large number of companies, the procedure is to let the customer know at what time the system will be updated. The system is then taken down, resulting in downtime, and the updated application is deployed over a small time frame. This experience might be undesirable for the customer. In the case of a banking application for example, it should be ensured to the customer that they can make a transaction whenever they want. Therefore, applications should be deployed without downtime. This is where blue-green deployments come in. A blue-green deployment ensures zero-downtime by making sure that at least one application instance is online at all times.

For stateless applications, blue-green deployments are mostly straightforward. In a data-intensive application, however, schema migrations also have to be deployed. This can result in quite complex blue-green deployments, since the application and database are coupled by a domain model. Both the application and database require an updated version. Therefore, a blue-green deployment has to be coordinated with care and requires some form of backwards compatibility. Dijkstra has implemented the expand-contract pattern for Oracle databases, using the Liquibase versioning tool, which shows how this can be done [18]. The pattern is divided in two steps: the expand step adds features to the schema, which results in a mixed-state that supports both database versions, while the contract step removes features to end up with the intended version. This is quite a robust pattern. However, additional complexity has to be taken into account when working with replicated databases such as in a highly available database cluster.

To build on Dijkstra's implementation, this research implements the expand-contract pattern for Postgres and extends its functionality to support more change types. Finally, the implementation is tested on a replicated database in a Kubernetes cluster. Postgres is chosen since it is widely-used, open source and has a large community. Additionally, ING has mentioned their interest for an implementation of the pattern for Postgres databases.

1.2 Problem definition

As explained, this research is focused around building a zero-downtime schema migration pattern. A tool has been requested by ING that can deploy schema migrations in a zero-downtime fashion. In previous research advocated by Dijkstra, an expand-contract pattern has been proposed. Several migrations were implemented for Oracle databases, using the Liquibase versioning tool. The deliverables of Dijkstra include, among others, a Liquibase extension that adds support for Oracle's Online DDL, and a set of expand-contract change templates that can be deployed without downtime. Since ING has mentioned their interest in using a Postgres database, this research will build upon the expand-contract templates to support Postgres. Another goal is to incorporate these templates into a Liquibase plugin. This would simplify the definition of a schema migration; the database administrator is not required to scholar themselves to understand each expand-contract migration.

Additionally, data-intensive applications nowadays require to be highly available. This is achieved by replicating application and database instances across several server nodes. This ensures that the system is fault tolerant. Once a node experiences a crash, it is expected that another node takes over their workload. In addition, the total workload will be distributed across all instances and therefore increases overall performance. To replicate database instances, data has to continuously be synchronized across all instances. This ensures that each instance contains a consistent state. Database replication can be done in many different ways and these will be discussed in this research.

Based on the aforementioned research goals, the following research questions have been formulated:

RQ1 How can the change templates from Dijkstra's research be extended to support more change operations that can be deployed on a Postgres database in a zero-downtime fashion?

RQ1.1 How can a Liquibase plugin be built that incorporates the change templates?

RQ2 To what extent can the expand-contract pattern be applied to a data-intensive system with a fully-replicated database to support zero-downtime deployments?

RQ2.1 How can the expand-contract pattern be applied when using master-slave replication?

RQ3 How much does the expand-contract pattern affect the latency and throughput of a data-intensive system?

RQ3.1 What tool is able to simulate a data-intensive system and can be used to test the Liquibase plugin and monitor the latency and throughput?

RQ3.2 To what extent can the tool be used to test and monitor a data-intensive system with a fully-replicated database?

RQ3.3 How does the latency and throughput for an asynchronously replicated database differ from the latency and throughput for a synchronously replicated database?

1.3 Research method

This research was conducted in several phases. The research started with a preliminary study during which required knowledge of the topic was acquired. This phase has been projected in Chapter 3. After the preliminary study, the research questions were defined. The next step was to do a full literature review in order to gain more detailed knowledge in the field and current state-of-the-art techniques that handle zero-downtime. Chapter 2 was written based on the obtained literature. An important aspect is the list of requirements for our Liquibase plugin which was taken from Dijkstra's research [18].

The next step in our research was to implement the Liquibase plugin which we have called `liquibase-zd`. While implementing the expand-contract pattern, the usage requirements were taken into account. The plugin was debugged in a Kubernetes cluster. Once the implementation was ready to be tested, HammerDB benchmarks were setup and run within the Kubernetes cluster. The benchmarks evaluate the performance requirements. In addition, the functional correctness of the implementation was evaluated by deduction.

1.4 Paper structure

In this section, we shortly explain the structure of this paper. This chapter has introduced the research and defined the research questions. In the next chapter, some related work will be discussed. Background information is provided in Chapter 3 to get a better understanding of the theory. In Chapter 4, the used technologies are discussed and the implementation of `liquibase-zd` is described in Chapter 5. The tests and benchmarks are presented and the results are examined in Chapter 6. Finally, the research paper is concluded by answering the research questions based on the results.

Chapter 2

Problem investigation

This chapter investigates the problem by first defining the requirements for a zero-downtime schema migration strategy. Next, literature related to the topic of this research is discussed. In Section 2.2, several state-of-the-art zero-downtime schema migration techniques are described and their limitations, with respect to the requirements, are considered. Furthermore, the method proposed by Dijkstra is presented and some additional research with respect to schema migrations is discussed. In Section 2.3, zero-downtime deployment techniques are described, and the concept of high availability is applied to databases by means of replication and is reported in Section 2.4. Finally, a list of change operations and their usage frequency from the research of Dijkstra is depicted in Section 2.5.

2.1 Requirements

In this section, the requirements of a zero-downtime schema migration strategy are described. These are mainly taken from the research of Dijkstra [18] who took **R5** onward from de Jong et al. [15]. **R8** has slightly been adjusted to comprise any constraint instead of only foreign key constraints.

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 **Integrity**. Constraints, such as foreign key constraints, should be supported. Both during regular use and while migrating the schema, constraints should not be invalidated.

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-invasive**. 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.

2.2 Schema migrations

2.2.1 State-of-the-art

There are many frameworks that try to solve the zero-downtime problem. Most of these make use of an interim or ghost table which is a copy of the original table. Essentially, a ghost table is created by copying the structure of the original table. The next step is to apply schema changes and finally copy the data from the original to the ghost table.

Examples of frameworks that use this strategy are `gh-ost` and `pt-online-schema-change`. Another example is `QuantumDB` which provides a wrapper for the JDBC driver that rewrites SQL queries by replacing the table name with the name of the ghost table depending on which table structure the application version depends on [14, 15]. Therefore, `QuantumDB` is very well applicable to zero-downtime schema migrations where a mixed-state is required (**R7**). A similar system, called `Ratchet`, has been developed by Zhu which creates a view for each table in the updated database schema and redirects queries to either the original table or the migrated view based on the query [35]. The advantage is that the data is not required to be copied to a ghost table, but the method does need an external proxy that sits between the application and database server. The same can be said for `PRISM` which also uses a proxy to rewrite queries in order to support the current schema version [11, 12, 13].

Additionally, Sheng proposes a lazy schema migration approach [29]. The idea is to apply the schema change to a copy of the original relation. The copy initially does not contain any data. The copy is only filled with data when a query cannot be applied to the original. A similar approach is used in `BullFrog`, an open source Postgres extension, which is able to achieve minimal changes in throughput and latency [4].

2.2.2 Shortcomings

The aforementioned frameworks have some shortcomings and limitations, and are therefore not the preferred approach for this research. First of all, most of these frameworks make use of ghost tables, hence require additional resources. Only renaming a column requires an additional table and copying of all the data. In a million-record table, this can be an expensive operation and should be eliminated where possible. The approaches proposed by Sheng [29] and Bhattacherjee et al [4] overcome this problem by lazily copying the original table. Systems such as `Ratchet` and `PRISM` require an additional proxy between the application and database server. This requires additional setup which we are trying to prevent (**R3**, **R10**). Finally, most of these frameworks do not support the enforcement of foreign key constraints during schema migration, which is considered a requirement (**R8**) [14, 15, 18]. For these reasons, a different solution has been chosen.

2.2.3 Expand-contract pattern

As described in the previous section, current state-of-the-art solutions do not fit the requirements. There are several other strategies to guarantee zero-downtime schema migrations as described by Dijkstra [18].

One of these strategies involves event sourcing. The approach requires a proxy that records all changes as events while the database schema is being migrated. Finally, the events that queued up during migration are forwarded to the updated database. Aside from requiring a whole new framework, as discussed by Dijkstra, earlier research has shown that this approach is not a serious contender for zero-downtime migrations.

The strategy that Dijkstra eventually decides upon, is called the expand-contract pattern, also referred to as parallel change [28]. This pattern has broadly been explained in 3.6. It has the advantage that the pattern is applied in-place, and therefore takes up minimal resources. Additionally, after the expand step, while the database is in a mixed-state (**R7**), the system can be tested and rolled back without much effort (**R11**). The challenge is to transform blocking changes into non-blocking alternatives (**R5**). Other strategies that Dijkstra refers to either require a NoSQL database, or are specific to Oracle, and are therefore not applicable to this research (**R1**). Furthermore, some of the techniques that Dijkstra uses to overcome the blocking behavior of schema changes are only supported by Oracle. Therefore, other solutions have to be found in this research.

2.2.4 Additional research

This section discusses additional research with respect to schema migrations that is interesting to look into.

Delplanque et al. figured that, specifically for PostgreSQL, schema migrations can be a hassle [16]. It requires extra steps to modify a table that is used by a view. Additionally, Postgres's procedural language offers the ability to create stored procedures. However, schema migrations do not migrate the code for these procedures. The paper proposes a tool that is able to identify dependencies and recommend changes to the developer that should be applied alongside the original schema change. Such a tool has not been developed yet. Aside from dependencies between database entities, the database is often coupled to an application by means of a domain model. A tool that proposes changes in the application code, based on a schema migration, has been developed by Meurice, Nagy and Cleve [24]. This tool will not be used during this research, but can be useful to developers that work with (zero-downtime) schema migrations.

Wevers et al. [32] have researched the effects of blocking operations on the throughput for several databases. The research shows that PostgreSQL performs rather well in comparison to MySQL and Oracle. PostgreSQL is able to add and remove columns, and only blocks when adding a column with a default value. Oracle is the only database that also blocks when dropping columns. Additionally, a bulk update operation, i.e. an operation that updates a large number of rows, blocks both PostgreSQL and MySQL. A solution, as implemented by Dijkstra [18], is to bulk update in steps of small batches. The bulk update operation was not possible on Oracle due to concurrency conflicts. All in all, the research shows that PostgreSQL performs well when it comes to blocking behavior.

2.3 Zero-downtime deployments

When it comes to application architectures, the micro services architecture has taken over from monolithic architectures. Micro services ensure more fault tolerance and are less demanding. Nevertheless, a software update would still require the application to be redeployed, resulting in a short time frame of downtime. To eliminate any downtime that the user would experience during a software update, zero-downtime deployments are required. For continuous delivery, zero-downtime deployments are a must. To be able to make the transition from the old to the new application, it cannot be avoided that both versions need to be online for a short period of time. Rudrabhatla discusses several deployment techniques to ensure zero-downtime [27]. One of the techniques that is also applied to micro services in other research [8, 9, 17, 26, 34] is the blue-green deployment [19]. Rudrabhatla has researched three different strategies to switch clients between the old and the new application versions. It is concluded that the load balancer strategy performs best in terms of switch-over time. However, it requires constant adjustment of the load balancer configuration when applied to a canary style blue-green deployment. Yang et al. – researchers from IBM, Amazon and Uber – propose a blue-green deployment strategy that utilizes automatic service discovery, dynamic routing and automated application deployment for improved continuous delivery [34]. Facebook has developed their own zero-downtime framework that seamlessly hands over connections, as well as current state, to the new server [25]. These papers also describe that containerization is very well applicable to micro services [9, 17, 27], because of their ability to launch quickly and provide lightweight environments. Kubernetes is mentioned as a suitable system to manage containerized applications.

2.4 Database replication

In a highly available data-intensive application, high availability is applied to databases through replication. Database replication techniques are often distinguished as being synchronous or asynchronous. Most papers refer to these variants as eager and lazy replication. A large amount of research has been done in this field. Lazy replication has been introduced in 1992 [23]. It aims to improve performance by requiring less overhead than eager replication. Lazy replication does result in a replication lag and should therefore only be used in case eventual consistency is allowed. Even though, eager replication does guarantee consistency, in practice, database replication is often applied lazily [33]. In the end, it depends on what the system permits. Wiesmann et al. discuss several protocols for eager replication and describe where overhead can be reduced [33]. In this research, both techniques will be discussed and evaluated for zero-downtime schema migrations.

2.5 Change operations

One of the deliverables from the research of Dijkstra is a tool that analyzes Liquibase changelogs and produces a frequency table. Using this tool, Dijkstra has put together a table with the frequencies of Liquibase changes used within a set of ING projects. This table is depicted in Table 2.1. From the results of the *pam* application, it can be seen that most operations involve adding or creating a database object while it happens less frequent that database objects are also removed. This table will be used as a guideline as to how urgent it is to implement the zero-downtime variant of certain changes.

	aax	pam	cha	irs	srv	cpa	nfe	**total**
addColumn	6	79	0	10	0	0	0	**95**
addNotNullConstraint	0	0	0	4	0	0	0	**4**
addForeignKey	0	38	0	0	0	0	0	**38**
addPrimaryKey	0	34	0	3	0	0	0	**37**
addUniqueConstraint	0	32	0	0	0	0	0	**32**
createIndex	8	15	0	6	0	0	0	**29**
createSequence	4	37	0	0	0	0	0	**41**
createTable	2	37	0	5	0	0	0	**44**
dropColumn	4	3	0	1	0	0	0	**8**
dropIndex	0	2	0	2	0	0	0	**4**
dropSequence	2	0	0	0	0	0	0	**2**
dropTable	1	4	0	1	0	0	0	**6**
dropNotNullConstraint	0	10	0	0	0	0	0	**10**
dropUniqueConstraint	0	7	0	0	0	0	0	**7**
modifyDataType	0	2	0	1	0	0	0	**3**
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**
$N_{ChangeSets}$	10	6131	4083	38	7	1571	572	**12412**
$N_{RollbackableChangeSets}$	7	1545	1353	24	4	111	267	**9335**
$N_{Changes}$	31	6131	4083	50	7	1571	590	**12463**
$N_{RollbackableChanges}$	23	739	646	34	2	60	109	**1613**

Table 2.1: Frequency analysis of Liquibase change types for several ING applications. For the *pam* project, a separate manual analysis has been done on SQL code changes to discover their respective change types. These numbers have been included in the table.

Chapter 3

Background

This section is meant to provide some background theory that is necessary to understand the research topic. The theory can be grouped into three parts. We start off by explaining database-specific theory after which we continue to application-specific concepts. The final section explains the expand-contract pattern as a schema migration technique for zero downtime deployments in the context of a stateful application.

3.1 Relational databases

3.1.1 Database schemas

A database schema defines an architecture for the database. It is a template of how the data is structured. There are different classes of databases based on how the structure is defined. In this research, we focus on relational databases. In a relational database, the structure describes the tables, and other associated database objects, and their relationships. The database schema is defined through a formal language like SQL. A database can have multiple schemas. Oracle, for example, creates a schema per database user. Nonetheless, often one schema is used in the context of an application.

3.1.2 Schema migrations

Now we have an understanding of what a schema is, we need to know what schema migrations are. A schema migration can involve adding a column to a table, renaming a column, or changing the data type of a column. These are only some examples. There are much more migrations possible. Schema migrations are executed programmatically through SQL queries which are received and processed by the DBMS.

3.1.3 Database management system

The database schema is defined by the database management system or DBMS. This is a system that allows for the definition and modification of a schema, but also for addition, manipulation and retrieval of the data. The DBMS provides a language that enables us to apply these operations. This language is known as SQL for structured query language and can be deconstructed in the following groups of statements:

- **Data definition language (DDL)**. DDL consists of statements that operate on the database schema. This includes statements such as CREATE, ALTER, DROP, RENAME and TRUNCATE. These statements modify the database schema and are therefore used for schema migrations. These operations are often blocking in nature.
- **Data query language (DQL)**. To read data from the database, DQL provides a single statement called the SELECT statement.
- **Data manipulation language (DML)**. Aside from reading data, there are also statements that manipulate the data such as INSERT, UPDATE and DELETE statements. These comprise the data manipulation language.
- **Transaction control language (TCL)**. Several operations can be combined to form a database transaction. This is a series of queries which are executed sequentially as a single unit. The transaction control language provides us with statements to manage transactions. This includes statements such as COMMIT, to save the results of the transaction, and ROLLBACK, to undo a transaction and restore the last database state.
- **Data control language (DCL)**. Finally, the data control language allows us to manage the privileges of database users. We can GRANT or REVOKE a privilege. We grant privileges in the form of statements that a user is allowed to execute or not.

3.1.4 ACID properties

A database is a critical construct. It can be used to store important data. The DBMS provides us with operations to update, insert, delete and retrieve data. We are relying on the DBMS to ensure that data does not go corrupt, as a result of a transaction, so we can focus on the application logic. Therefore, the DBMS must provide us some guarantees about database transactions. These guarantees are described in the ACID properties:

- **Atomicity**. A database transaction should either be executed entirely, or the state is rolled back in case of a failure. It can be seen as an all-or-nothing operation. An example that is often used to explain this property is a banking system. If an account A transfers money to an account B, then the amount has to be subtracted from account A and added to account B. These steps have to be executed atomically, i.e. they are seen as a single operation. If atomicity is not guaranteed, then it could be possible that the transaction fails after the subtract operation. As a result, some amount of money disappeared and this should be reversed. Generally, atomicity is implemented by keeping a copy of the data prior to execution of the transaction.

- **Consistency**. It must be guaranteed that the database is in a consistent or correct state at all times. That means that data integrity constraints cannot be violated. These are restrictions on the data such as data types or nullability constraints. A column that is specified to be *NOT NULL*, should be guaranteed not to contain a row where the value is *NULL*. A transaction that tries to insert a *NULL* value in this column, should be rolled back. Application-specific constraints cannot be guaranteed by the DBMS. An example would be to ensure that an account balance can never be negative and a transaction that tries to subtract an amount that would result in a negative balance, should never be executed. These restrictions have to be enclosed in the application code.

- **Isolation**. The DBMS needs to guarantee that the database is safe to concurrent executions. In other words, the database state after concurrent execution of transactions and after sequential execution are the same. There are several phenomena that can occur when reading data:

  - **Dirty read**. A dirty read occurs when uncommitted data is being read. This should never happen.
  - **Non-repeatable read**. If a row is read at time T1 and read again at time T2, the data in that row might have changed.
  - **Phantom read**. If a query is executed at time T1 and re-executed at time T2, additional rows may have been added that might affect the results.

  Based on these phenomena, four isolation levels are defined that provide their own guarantees:

Isolation level	**Dirty read**	**Non-repeatable read**	**Phantom read**
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Table 3.1: The isolation levels described by the possibility of the three phenomena.

  The serializable isolation level makes the best guarantees to ensure consistency. Serializable transactions are all executed one after the other to ensure that a transaction cannot read an inconsistent state as a result of a partial execution of another transaction. A negative consequence is that serial executions are less performant than parallel executions. In many situations serializability is not even required to keep a consistent state and a lower isolation level would actually improve performance [2, 3].

- **Durability**. All committed transactions must be persisted in the database and a system failure or restart should not affect the data. Data should not be kept in non-volatile memory, but should be written to persistent memory, such as a hard drive, instead.

3.1.5 Concurrency control

The ACID properties are implemented through concurrency control. Concurrency control ensures that transactions can be executed concurrently, i.e. in parallel, without producing incorrect or corrupt results.

Concurrency control protocols can be categorized as being optimistic or pessimistic. Pessimistic concurrency control prevents two transactions from modifying the same record at the same time by blocking operations if there is a possibility that the ACID properties are violated. The blocking results in a performance overhead. Additionally, pessimistic mechanisms are prone to deadlocks. A deadlock occurs when two transactions are waiting for each other to unblock some portion of the data. Consequently, they each wait for an infinite amount of time and cannot finish their execution. On the other hand, optimistic concurrency control does not block concurrent operations. Instead, it allows for concurrency conflicts to happen. If a conflict happens, it is detected by the DBMS and one of the transactions is rolled back. The transaction is restarted and re-executed. Evidently this means that conflicts are expensive. Therefore, optimistic concurrency control should be used in case conflicts are expected not to happen too often. Otherwise, pessimistic concurrency control can result in better performance.

Locking

The conventional concurrency control protocol involves locking. This is a pessimistic method. Any transaction that tries to access the database has to acquire a lock. There are two kinds of locks: *exclusive* and *shared* locks. In case of a write operation, the transaction acquires an exclusive lock on a database object and releases it again after the operation finishes execution. The shared lock is used for read operations. It is possible for multiple transactions to acquire a shared lock and perform a concurrent read operation. On the other hand, an exclusive lock can only be acquired if no other lock is held on an object. Additionally, locks can be held on rows, or an entire table. Holding a row-level lock instead of a table-level lock allows for multiple transactions to concurrently write to separate rows.

The mechanics of these two locks ensure that ACID properties remain intact. However, it does not guarantee the exclusion of deadlocks. For example, assume transaction T1 holds an exclusive lock on object A and transaction T2 holds an exclusive lock on object B. If T2 also tries to acquire an exclusive lock on object A and T1 tries to acquire an exclusive lock on object B, both transactions need to wait on each other to release the initial locks, resulting in a deadlock. There are many ways to handle a deadlock situation. Generally, it is difficult to prevent deadlocks, but it can be done by tracking resource allocation and reason which resources a process requires in the future[1]. Another approach is to allow deadlocks, detect their occurance by tracking the resources and recover from the undesirable state.

[1]https://en.wikipedia.org/wiki/Deadlock

Multiversion concurrency control

Multiversion concurrency control or MVCC is a concurrency control protocol that has been widely employed by database management systems such as Postgres and Oracle. It is an optimistic method. Moreover, it is a timestamp-based protocol. That means that it assigns a timestamp to each transaction. The older the transaction is, the higher its priority. A new transaction has to wait until older transactions finished execution. Additionally, as opposed to locking, MVCC prevents read and write operations from blocking each other. This is done through snapshots. Each query works on a snapshot of the current state at the time the query starts. This is where the term snapshot isolation comes in. Once a transaction starts, it takes a snapshot of the current data and only operates on the snapshot. Read operations are only able to see changes that were applied before the start of the transaction. A write operation only updates the snapshot to ensure that concurrent reads can still access the old state. As a consequence, read transactions never have to wait. On the other hand, timestamps ensure that write operations block each other and therefore do not result in conflicts.

3.1.6 Strict serializability

Serializability speaks about the correctness of the execution of concurrent transactions. A set of concurrent transactions is executed in a schedule. It is possible to execute one transaction completely after the other which is referred to as a serial schedule. Serially scheduled transactions prevent any data inconsistencies. A non-serial schedule comes to mind when transactions are not serial and instead individual operations of the transactions are interleaved. This can lead to concurrency issues. A non-serial schedule can however still be serializable. A non-serial schedule is serializable if the result is equivalent to the result of a serial schedule of the corresponding transactions. Non-serializability of a schedule leads to conflicting operations and an inconsistent database state.

Linearizability is another term used in databases and it is important that the reader has a good understanding of this term. It provides a real-time guarantee about single operations on single (database) objects. That means that a write operation to a single object should instantly be visible to a read operation of the same object. Any read operation that is executed after the write operation, in real-time, should read the written value or the value of a later write. Another term for linearizability is atomic convergence.

The combination of serializability and linearizability is called strict serializability[2]. It puts real-time constraints on the execution order of the transactions. If a transaction A starts before transaction B, then A is scheduled before B and the result after execution is equivalent to the result of a serial schedule.

[2]`http://www.bailis.org/blog/linearizability-versus-serializability/`

3.2 Software architectural styles

An application consists of several layers that are working together to provide the user a functioning product. These are usually defined in order as:

1. Presentation layer
2. Business logic layer
3. Data access layer

A stateless application does not persist any user data and does therefore not require a data layer. However, most applications are stateful and use a database to persist their data. Each layer builds logic on top of the other layer and is only allowed to request information from the layer below it. The business layer consists processes or tasks that use data from the data layer and perform some operation on it. The presentation layer translates information from the business layer into something that the user can understand. In a web application, this usually means that information is displayed in the form of an HTML page. An HTTP request is sent from the presentation layer to the business layer to request any stateful information.

3.2.1 Monolithic architecture

A monolithic application is an application in which there is no clear separation of the three layers, and the layers run in a single environment and on a single computing device. An advantage of monolithic applications is that they are easy to develop, and easy to test and deploy. This makes it a good use case for a proof of concept or a small project. However, monolithic applications tend to be more difficult to scale horizontally; it requires to deploy all the code on multiple machines. Moreover, maintaining a monolithic application is not straightforward either, since a small change can affect the whole system [20]. Another disadvantage is that updating a single component requires the entire application to be redeployed.

3.2.2 Layered architecture

By splitting up the parts that comprise an application, we can make it more modular, more maintainable and easier to scale. Therefore, an improvement on the monolithic architecture is the layered architecture. First of all, the presentation, business and data layer are separated from each other. Each layer can even run on a separate device. There is no need for the presentation layer to know how the business layer functions. All it needs to know is that there is a business layer that provides information to display. Furthermore, the business layer does not need to know how the presentation layer is going to display this information. Second of all, the business layer can be split up into a coarse-grained set of services. The idea behind this is that an application is meant to serve distinct business processes. For example, a login service provides the user with a means to authenticate themselves to the application. The login service does not have to know about any other services and can therefore be structured as a separate component. There might also be services that do require communication with each other. This usually involves data passing or communication through middleware.

3.2.3 Microservices architecture

Microservices architecture goes a step further than the layered architecture. The services are split up into even smaller more fine-grained services. Each service is self-contained and implements a single business capability within a bounded context. Bounded contexts separate business domains. For example, a checkout service has its own bounded context and is separated from the catalogue service. Information can still travel between these bounded contexts, but the business logic is divided into features with well-defined boundaries. The microservices are therefore small, independent and loosely coupled by hiding implementation details from each other [10]. An important consequence is that small teams can work on their own microservice. This avoids a vast overhead of communication within large teams. Microservices can be deployed independently without requiring to redeploy the entire application. Additionally, they are responsible for persisting their own data, but can communicate information to each other through APIs or middleware such as an event bus. Finally, each microservice can be written in their own technology stack. In other words, they do not need to share programming language or dependencies.

All of this does come at a cost: additional constructs are required to orchestrate the microservices which increases the complexity of applications. The upcoming sections will discuss some of the most important constructs.

Load balancer

The first component that is required is an API gateway or an enterprise service bus. Clients do not need to know about the different services, but instead call the API gateway with a request. It is then the job of the gateway to forward this request to the appropriate services. This is another example of a construct that is used to decouple parts of the application and make it more modular. An API gateway usually also implements load balancing. This is a way to efficiently distribute the incoming requests across multiple servers. For simplicity, the term load balancer is often used when referring to an API gateway that implements load balancing.

Service registry

In order for the load balancer to forward a request to a microservice, it needs to know about the existence of these microservices. Moreover, it needs to know the IP addresses. These can be configured in a configuration file. However, cloud-based applications assign addresses dynamically. In such cases, a service registry is used where microservice instances can register themselves. The service registry consists of a database of microservices, their instances and their IP addresses. Microservice instances register themselves with the service registry when launched and deregister themselves before shutting down.

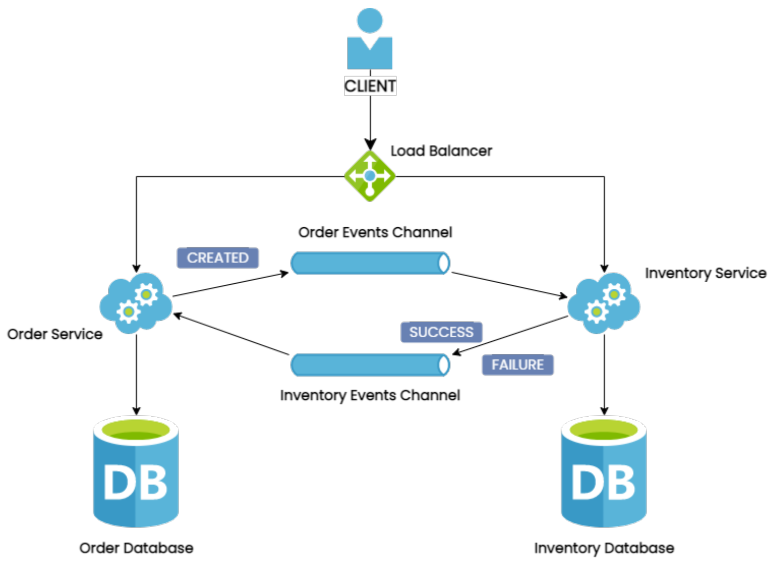


Figure 3.2.1: An example of a microservices architecture for an online store. The architecture implements the database-per-service and sagas pattern.

Shared database anti-pattern

In a stateful microservices architecture, the decision has to be made whether to use a single database that is shared across all microservices, or whether to use one database per service. It is possible to use a shared database where each service can access data from other services using local ACID transactions. However, it is strongly advised not to, and it is even considered to be an anti-pattern. The reason is that it could result in microservices blocking each other when trying to access the same data. Additionally, it breaks the idea of separating the concerns of development teams. Teams have to communicate database schema migrations with each other.

Database-per-service pattern

As explained in the previous section, a shared database is an anti-pattern. It is preferred that each microservice owns its own database instead. This has the advantage that microservices are kept decoupled. Database schema migrations will not have to be coordinated with other teams, and database queries do not block other services.

Sagas

Even though the database-per-service pattern is preferred, microservices might still have to access and update data owned by other microservices. Think for example of an online store. A request for an order that is received by the order service needs to remove items from the inventory database which is owned by the inventory service. There are a couple reasons why we cannot simply tell the inventory service to remove these items. First of all, the inventory might not contain enough of these items. Secondly, if the transaction from the inventory service fails, the order service also needs to rollback its transaction.

The solution is to implement sagas. A saga is a sequence of local transactions that live across multiple services. Each service commits a local transaction and sends an event to trigger the next local transaction. If one transaction fails, then an event is sent back down the saga to reset the state. In case of the online store, the order service creates an order and tells the inventory service to update the inventory as shown in Figure 3.2.1. If there are not enough items, or the transaction fails, the inventory service tells the order service to cancel the order. Otherwise, the inventory service sends a success event. The previously described coordination method is called choreography. Another method uses an additional component, called an orchestrator. Instead of sending events, the orchestrator sends commands to the next microservice to tell what transaction to perform.

The additional complexity that comes with a microservices architecture is often outweighed by the flexibility. Therefore, most companies implement a microservices architecture and even legacy monolithic applications are migrated to use microservices.

3.3 Distributed systems

In the previous section, we have already seen that we can either have a single machine on which all components live such as in a monolithic architecture. On the other hand, a system can be distributed. It is difficult to find a single interpretation of a distributed system, but van Steen and Tanenbaum provide an accurate definition [31]:

> "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system."

In other words, in a distributed system, components are spread across multiple machines and they all work together as a system. A distributed system can also be geographically scattered. A microservices architecture (or layered architecture) is a good example of a distributed system. Each microservice runs on its own server and communicates with other microservices across the network to appear as a single coherent unit.

In contrast to a distributed system, a monolithic application has one single point of failure. This means that a fault will result in a system crash. Single points of failure should always be avoided. Therefore, a distributed system is much more attractive, since the workload is not put on a single machine, but spread across several servers. If one microservice fails, other microservices are still available to the user. This is called a partial failure. This is of course an advantage, but there is also a disadvantage to it: there is no way for the other components to know that a machine failed, let alone which machine. After all, the network is unreliable. This means that requests may be lost or delayed, and vice versa for responses. Another possibility is that a machine's clock is out of sync with the rest of the system. There are solutions out there that provide fault tolerance for distributed systems such as health checking or the circuit breaker pattern.

3.3.1 High availability clusters

The availability of a system is described by uptime and downtime. Uptime is the amount of time that the system is working correctly and is available for requests. When the system is not up, it is down and we refer to the amount of time that the system is down as downtime. Generally, we try to minimize the downtime, so our system can be used without (many) interruptions. In other words, we aim for a high availability, meaning there is only a very small percentage of downtime. Promises about uptime and downtime are described in service level agreements (SLAs).

A high availability cluster (or failover cluster) is a distributed system that ensures high availability. Therefore, the system needs to be fault tolerant. That means that a failure of one component does not result in a failure of the entire system. There are three principles to high availability that ensure fault tolerance:

HA1 **Failure detection**. If a component fails, it needs to be detected by the system. Usually, this is done by adding a monitor that sends health checks to the system parts.

HA2 **No single point of failure**. If the system has a single point of failure, we rely on that part of the system. If this component crashes, the whole system might crash. Therefore, high availability clusters add redundancy by copying parts of the system.

HA3 **Reliable crossover**. When a health check fails, the system needs to restart the application on another machine. This is called failover or crossover.

The last two principles, **HA2** and **HA3**, are mainly about failure recovery. When one part of the system fails, it needs to be recovered on an additional machine. There are two main architectures or strategies for a high availability cluster to ensure this:

1. **Active-passive clusters**. Alongside the primary server that serves the application, there is an additional redundant server that is only activated when the primary server fails.

2. **Active-active clusters**. Several servers run in parallel behind a load balancer that distributes the workload. In case of a failure, the workload is distributed across the remaining servers.

Active-active clusters are often preferred, since it ensures that all servers are active, instead of having a redundant server that is only active in case of a failure of the primary server. Additionally, this means that all workload is load balanced which improves overall performance.

High availability for microservices

In the context of a microservices architecture, we might have several servers serving the same microservice. These can even be geographically distributed. A geographically distributed system consists of clusters across several regions. This results in a lower latency by redirecting clients to the closest regional cluster using a global load balancer. Each cluster runs one or more instances of every microservice. By having multiple active instances of a microservice, requests can be redirected to a different instance in case of a failure. In case the entire cluster fails, the requests will be forwarded to the next nearest regional cluster. These configurations ensure that there is no single point of failure in the system.

3.4 Database replication

Database replication is the process of creating a copy of a database and keeping the data in both instances synchronized. The original database is often called the primary or master database, while a replica is called a standby or slave. Each database instance lives on a separate server to ensure that a crash only results in failure of a single instance. There are two different forms of replication:

- **Sharding.** Sharding or horizontal partitioning is understood as partitioning the data across several database instances. Instead of keeping all rows in one database instance, they are spread across multiple instances. Sharding is often used to geographically distribute the data. Data objects that contain regional information are sharded and located at the database of the corresponding region. These data objects are most often accessed at their respective regions, hence sharding decreases latency. Additionally, since each database instance is of limited size, queries require less lookup time.

- **Full replication.** Another form of replication is full replication which replicates the entire database across all instances. This shall research focus on full replication and full replication is to be understood when referring to replication.

It is important to note that database replication is different from duplication. A duplicate of a database is an exact copy of the database at a specific moment in time and is often used to keep a back-up. Replication, on the other hand, keeps the data synchronized such that the replicas are consistent with the primary database. Replication is implemented to fulfill two main goals:

DR1 **Availability**. A consequence of database replication is an increase in availability of the data. The data is available as long as one instance is online and is accessible. Of course, this requires the system to be fault tolerant. That is, the system should still function correctly if part of the system fails. As we have seen, high availability is an important concept in order to implement fault tolerance in a distributed system. The same principles can be applied to database replication to ensure fault tolerant databases.

DR2 **Scalability**. It is possible to load balance the workload across several replicas to ensure a higher throughput. This makes the database more scalable, even though it requires more storage space.

3.4.1 CAP theorem

Replication can be implemented in many different ways. In order to understand which technique might be preferred, we need to know about the CAP theorem by Brewer [7]. Its definition can be quite unclear and has been critiqued throughout the years [6, 21]. The CAP theorem states that any distributed database can only guarantee two of the following three properties: consistency, availability & partition tolerance. The critique discusses several aspects that are wrong or unclear in the definition. First of all, the last property is unclear. It essentially means that the system should be tolerant against network failures between distributed nodes. In practice, this property should always be preserved and the theorem comes down to choosing between consistency and availability. It means that the database cannot provide both consistency and availability in case of a network failure. However, as the critique states, this is not a system-wide decision. It is possible that one subsystem prefers consistency over availability, and vice versa for another subsystem.

Abadi provides more critique to the CAP theorem and discuss that, when no network partition is present, there is still a tradeoff between consistency and latency [1]. Latency describes the time it takes to send a request for a query, process it and send back a response. Essentially, an unavailable system has very high latency, while a low latency indicates availability. Availability does not mean no latency at all. Moreover, as Abadi describes, for a replicated database to be consistent, more latency is unavoidable. Abadi captures this tradeoff in a variant of the CAP theorem called the PACELC theorem: in presence of a network partition (P) in a distributed system, there is a tradeoff between availability (A) and consistency (C), but else (E), when there is no network partition, latency (L) or consistency (C) should be weighed up.

Additionally, to the above critique towards the CAP theorem, we should critique the use of the word consistency. In this case, the definition of availability is as described before. However, we have seen the use of the word consistency in two concepts: ACID properties and the CAP theorem. Most research uses the same name for both definitions. Nonetheless, they are considered two different constructs. Therefore, it is important to make the distinction. Consistency, in the context of ACID, is a guarantee that data integrity constraints are not violated. On the other hand, we have consistency in the context of a distributed database and the CAP theorem. In this context, consistency states that all replicas read the same database state. That is, a write operation is instantly applied to all database instances. To distinguish both constructs, we will use a different name for the latter definition: convergence[3]. In section Section 3.4.4, it will become clear how the CAP theorem is applied in practice. There we will also distinguish another form of convergence called eventual convergence.

3.4.2 One-copy serializability

In Section 3.1.6, we have discussed the meaning of serializability in the context of a non-replicated database. The terminology behind serializability can also be extended

[3]`https://pathelland.substack.com/p/dont-get-stuck-in-the-con-game-v3`

to replicated databases. In that case we are talking about one-copy serializability. A non-serial schedule of concurrent transactions across a replicated database system is one-copy serializable if the result after execution is equivalent to the result of a serial schedule on a single database instance.

3.4.3 Database architecture for replication

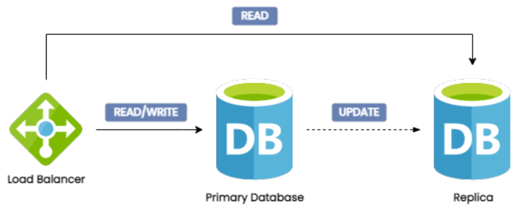
Before implementing the replication mechanism, an architecture has to be designed. For instance, it has to be decided how many replicas the database will have. Most of all though, a decision has to be made whether write operations are only allowed to be received by the primary database or also by the replicas. This question categorizes two architectures:

- **Master-slave**. In a master-slave architecture, the primary or master database receives write operations and replicates them to one or more replicas. For a master-slave architecture, we can additionally decide between a warm standby or a hot standby. A warm standby is a replica that only tracks changes of the master and updates accordingly. On the other hand, a hot standby is, in addition, online and available for read-only transactions. Since we aim for high availability and scalability, the better option is to use hot standby servers. The disadvantage is that it has to be ensured that the hot standby has processed all updates. This could lead to added latency when waiting for the standby to be in-sync with the primary database. In Section 3.4.4, this will be discussed in more detail. An example of the master-slave architecture can be found in Figure 3.4.1a. It is also referred to as unidirectional replication, since the changes are always replicated from the master to a slave.

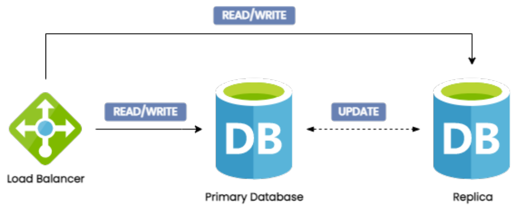
- **Master-master**. In a master-master setting, every database is considered equivalent. That means that any database can receive write operations or read operations. A write operation will have to be synchronized with the rest of the databases. The main disadvantage, in comparison to a master-slave setting, is that concurrent write operations to multiple databases can result in conflicts. Therefore, there needs to be a conflict resolution mechanism in place.

There are reasons to choose master-slave over master-master and vice versa. A master-slave architecture is more straightforward to setup and does not have to deal with conflict resolution. On the other hand, all write operations are forwarded to a single database node. For that reason, in a system where lots of writes are expected, a master-master replication mechanism is preferred. Conflicts will have to be dealt with in an orderly manner. For the remainder of this section, we assume a master-slave architecture since this simplifies our replication process.

It should be noted that databases are different from software systems. Additionally to a system crash, it is possible that the underlying hardware has been physically damaged and corrupted the data. Therefore, it is always important to create a backup if decided against replication.



(A) An example of a master-slave replication architecture with a hot-standby.



(B) An example of a master-master replication architecture. Replica control is bidirectional.

FIGURE 3.4.1: A comparison of the master-slave and master-master replication architectures.

3.4.4 Replica control

As discussed, replication can be used to provide high availability for databases. Replication is the process of creating a copy of a database and keeping the data in both instances synchronized. While previous principles focused on the architecture of a replicated database system, this section puts emphasis on the synchronization step. The synchronization step is called replica control. There are many mechanisms to implement replica control. A replica control mechanism can either synchronously or asynchronously replicate the data:

- **Synchronous replication**. Any write transaction waits for the transaction to be committed to the replica before committing itself. With respect to the CAP theorem, synchronous replication provides convergence across all replicas. However, this means that a transaction might take longer to finish and therefore reduces availability.

- **Asynchronous replication**. Write transactions are replicated after being committed. This is done asynchronously. Therefore, there is a short time frame, called the replication lag, during which the replica database has not converged. While asynchronous replication can promise availability, convergence cannot be guaranteed. Instead, it provides eventual convergence: if no additional data is written, the system will eventually converge across all replicas. In some systems, eventual convergence can be enough. A disadvantage of asynchronous replication is that data can be lost if the system crashes during replication.

Deciding between synchronous and asynchronous replication means deciding between convergence and availability. Synchronous replication can provide better fault tolerance. If the primary database fails, a replica is guaranteed to converge and can directly catch any incoming workload. Asynchronous replication on the other hand does not provide any replication overhead. Instead of deciding between synchronous and asynchronous replication on an application-level, we can also decide per database. Some data requires to be consistent all the time, while other data can do with eventual convergence.

There are many replication mechanisms, and it depends highly on the DBMS which technique should be used. Next, we present some of these solutions:

- **SQL-based replication**. The SQL statements can be intercepted and forwarded to the replicas. This way, the same SQL query will be executed on all servers. However, non-deterministic query statements such as *CURRENT_-TIMESTAMP* can result in different data across the replicas. Furthermore, it must be ensured that a transaction is committed on all servers. This can be done using an atomic commit protocol such as two-phase commit. For these reasons, SQL-based replication is not preferred.

- **Trigger-based replication**. Replicates data based on SQL triggers. An SQL trigger can listen for update or insert operations and act upon them. By placing triggers on the primary server, it can send data changes to replicas. This has as an advantage over the SQL-based replication that the SQL query is only executed once, resulting in less overhead. The results are directly copied across all replicas.

- **Log shipping**. Trigger-based replication replicates the data through SQL. On the other hand, databases keep a transaction log. This is a file that records all the modifications to the database. These logs are mainly used to recover your database to a correct state after a failure. However, they can also be used for replication. In log shipping, the primary server sends changes as log records to the replicas. The replica then updates the database by applying the changes seen in the log record. The advantage to trigger-based replication is that we offload the replication process from the database. In addition, it does not require us to write SQL triggers. Overall, log shipping is the preferred method. It can be implemented asynchronously, or synchronously by waiting for approval from all the replicas.

3.4.5 Conflicts

Even log shipping replication still has some challenges to overcome. These challenges mostly arise from conflicts between the primary and standby servers. If a write happens on the primary server, simultaneous to a read operation on the standby server that accesses the same data, there are two choices that we can make. One is to wait for the read query to finish before publishing the changes to the replica. However, if the query takes a long time, the replica can run behind on the primary. The second choice is to forcefully cancel the read query and maybe try to re-execute it after applying the log records.

3.5 Application deployments

When releasing a new software version, the new application needs to be deployed. There are several ways to come about this. This section talks about deployment pipelines, gives a definition for zero-downtime, and discusses several deployment strategies to overcome downtime.

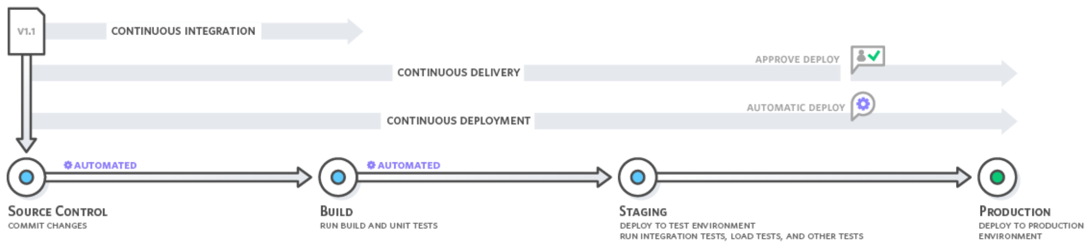


Figure 3.5.1: A comparison of the continuous integration, continuous delivery and continuous deployment pipelines.
Source: `https://aws.amazon.com/devops/continuous-delivery/`

3.5.1 Deployment practices

Application deployments are often automated in a pipeline. A code change has to be committed by the developer after which the pipeline executes a sequence of automated steps to check for correctness of the application:

1. First of all, the pipeline runs a build script to ensure that the application compiles. The unit tests are run as well.
2. If the first step succeeds, the build is deployed to the testing environment in order to run additional tests such as integration tests.
3. Finally, assuming all tests succeeded, the application is deployed to the production environment.

It is not required to run all these steps. There are several deployment practices that each have their own pipeline as shown in Figure 3.5.1. One of the deployment practices is continuous integration where only the first step is executed. Continuous integration is used in case developers commit changes often. It ensures that the code always functions correctly. An extension of continuous integration is continuous delivery. A continuous delivery pipeline executes the first two steps described before. This ensures that there is always a release version ready which has to be manually deployed to the production environment. If we include automatic deployment into the pipeline, we speak of continuous deployment.

3.5.2 Zero-downtime

It is important to give a definition of zero-downtime, since it can be interpreted in different ways. The term can be deconstructed into ’zero’ and ’downtime’. Downtime is defined as the time that an application is offline and thus zero-downtime would mean that the application is always online and available for requests. Generally, this would be a correct definition for zero-downtime and can be achieved. However, a will always result in some (additional) latency and a change in the throughput. The current definition of zero-downtime does not tell us anything about how much latency or throughput a user might observe. Therefore, if we speak of in this research, we extend it to the perception of the user. In other words, we try to minimize the perceived change in latency and throughput.

3.5.3 After-hours deployment

The following sections describe some deployment strategies and discuss their properties with respect to zero-downtime.

The most straightforward deployment strategy is to take down the current application, apply database migrations, and deploy the new application. This can be scheduled to be done during low usage hours. Such an after-hours deployment can often be recognized by the maintenance notification that users receive. Of course, this results in some downtime. Hence a different deployment pattern needs to be used to guarantee zero-downtime.

3.5.4 Blue-green deployment

A blue-green deployment is more robust and does not entail any downtime [27]. All in all, to be able to support zero-downtime when transition from one application version to another, there can never only be one application instance running. In other words, we always have to run both application versions in parallel for some time frame. For this concept to work, a load balancer is required. A blue-green deployment initially only has one or more instances of the old application running behind the load balancer; all requests are forwarded to the old application. The old instances are called the "blue" instances. After the updated application has been developed, we deploy several instances in parallel with the blue instances and call them "green" instances. Once the green instances are active and have been thoroughly tested, we tell the load balancer to direct any requests to the green instances. As soon as all clients are moved to the green instances, it is safe to take down the blue instances.

As we can see, during this entire process, clients will still be able to use the application. At any point in time, at least one of the application instances is active. Therefore resulting in zero-downtime. However, the user might experience some latency when the client is moved to the green instance.

When it comes to blue-green deployments, there are some best practices to keep in mind. First of all, it is good to do several deployments by repeatedly deploying small changes and switching from blue to green until all changes are applied. This ensures that we do not directly deploy a new application version that is not compatible with the old version. Furthermore, the application instances should always be monitored and the new instance should be thoroughly tested. If the new instance fails or results in unforeseen behaviour, it is still possible to take down the green instance while keeping the blue instance running.

Besides, the standard blue-green deployments, there exist some variations such as:

- **Canary deployment.** In a canary deployment, we only redirect a small subset of users to the green instance. This is great for performance monitoring and user testing, and makes it less demanding to rollback. When the green instance has been thoroughly tested, the rest of the clients can be redirected to use the new application version.

- **Rolling deployment.** Blue-green deployments instantly switch all service instances from the old to the new version. In a rolling deployment, we swap instances out one after the other. This results in slower deployments, but requires less resources since it only needs one extra instance to run in parallel at a time.

3.6 Expand-contract pattern

The aforementioned blue-green deployments work in a stateless application. However, applications often require state and a database to store that state. The state is incorporated in the application as the domain model. Essentially, it is a definition of the database structure inside the application. The database is coupled to the domain model and any database query loads the data into the domain model. When the database schema requires a migration, the domain model that depends on that schema also needs to be updated. To facilitate zero-downtime schema migrations, the application deployment and schema migration have to be coordinated in order to stay coupled. It is inevitable that two application versions, conforming to two versions of the domain model, require to be online at the same time. This asks for a mixed-state database [14]. In other words, the database is required to support both application versions. The expand-contract pattern, as described by Dijkstra, supports a mixed-state [18].

The expand-contract pattern consists of two steps: expanding the schema and then contracting it. This is best explained by looking at an example. If we want to rename a database column, the expand step creates a new column with the updated name. The database is now in a mixed-state. That means that both the old and new application version are supported. Subsequently, the new and old column are synchronized by applying update and insert operations to both columns and copy any remaining data from the old column to the new column. This synchronization step can be done through application code or database triggers. Finally, the contract step removes the old column.

There are several advantages to the expand-contract pattern. First of all, it does not require a great amount of extra insight from the developer. Secondly, the pattern is backwards compatible while the database is in a mixed-state. This also means that, after the expand step, the migration can easily be rolled back to the original state. The expand-contract pattern will not block access to the database either. Finally, this pattern can be interwoven into the blue-green deployment pipeline without much trouble. After all, blue-green deployments are a form of parallel change providing zero-downtime [28], similar to the expand-contract pattern. The steps of both methods can be alternated to ensure zero-downtime schema migrations. An example of this is described below. In addition, the steps are illustrated in Fig. 3.6.1.

Expand-contract blue-green deployment pipeline

1. Start with application version 1.0.0 and database version $v1$.

2. Apply the expand step to the database. The database is now in a mixed-state supporting both version $v1$ and $v2$.

3. Deploy one or more application instances of version 2.0.0.

4. Ensure the new application functions correctly.

5. Redirect all clients to the new application instances and wait until no client is using application version 1.0.0 anymore before moving on. It might be a good idea to monitor the new application. If the application fails and needs to rollback, then now is the time. For these reasons, a good alternative to the blue-green deployment could be a canary deployment where only a small portion of users is redirected to the green application instances while being monitored. This ensures that faults in the system are detected early on and can be managed more easily.

6. Take down the old application instances.

7. Finally, apply the contract step to the database. The database is now in version $v2$. After this, the migration cannot be rolled back anymore. Instead, a separate migration that reverts to the old state would have to be applied. This is called a forward roll.

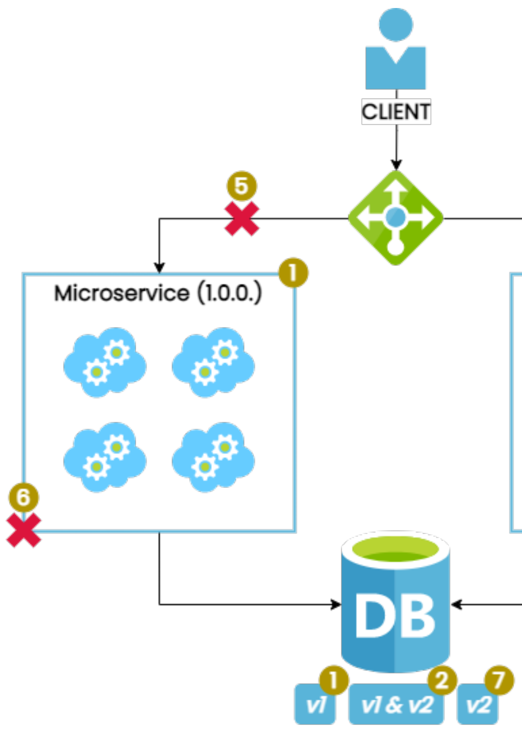


FIGURE 3.6.1: An explanation of how the expand-contract pattern is applied in a blue-green deployment. Each number represents a step in the expand-contract blue-green deployment pipeline as defined in Section 3.6.

3.7 Physical and logical replication

Postgres supports two types of replication: physical and logical replication. Physical replication is the default and deals with replication on a file system level [5]. Data and schema changes are replicated in a binary format by means of write-ahead logging (WAL). The WAL records are streamed to the standby server which applies them to the database replica.

Physical replication is worried about having the same file structure. On the other hand, logical replication only ensures that the data is copied correctly on the basis of SQL statements [5]. It does not care about how the underlying file structure is implemented. Therefore, an important advantage of logical replication is that it allows replication between two different (major) Postgres versions. Logical replication makes use of a publisher and subscriber pattern. The primary database sets up one or more publications. A publication can be applied to the entire database, or limited to a subset of tables. Furthermore, it is possible to specify which DML operations it should be restricted to. Once a publication exists, the standby server can create a subscription that listens to changes from the publication. Each change is then applied to the database.

Both physical and logical replication have their advantages and disadvantages [5]:

1. Logical replication does not support DDL changes while physical replication does. This is a major advantage of physical replication for our use case, since schema migrations only have to be applied once to the primary database.

2. Logical replication is unable to stream changes. A transaction is only send to the replica on commit. Therefore, there could be a large delay before the replica converges.

3. Logical replication does not support replication of sequences, truncate operations, or large objects.

4. On the other hand, it is possible to use logical replication in a multi-master architecture while physical replication does not support this.

5. Moreover, as mentioned, logical replication enables replication between two major Postgres versions.

Even though, DDL statements are not replicated by logical replication, it is still feasible to apply the expand-contract pattern on a replicated database. Since logical replication copies changes by means of SQL statements, changes in the data can be replicated to an expanded (mixed-state) database. Consequently, the example in Section 3.6 can be extended for a logically replicated database. In that case, step 2 should be applied to the slave instance before applying it to the master database. Any write operations coming in on the master database can still be forwarded to the slave that is in the mixed-state. Similarly, the contract phase in step 7 should first be applied to the master database, since the contracted state is compatible with the mixed-state of the slave.

For this research, it has been decided to use physical replication, since it is much more dependable and simplifies the deployment pipeline. Logical replication has been around since Postgres version 10. Therefore, it is still quite new and has not been widely adopted. Moreover, physical replication is the default replication type and is also a technique found in other DBMSs while logical replication is specific to Postgres.

Chapter 4

Technology stack

4.1 Liquibase

Liquibase is an open-source database schema versioning tool [1]. The tool records each schema migration. This enables the user to keep track of the current database schema and ensure it lines up with the application code. In addition, we can easily revert to a previous schema version by rolling back the last schema change. Schema changes are defined in a Liquibase specification in the form of, for example, XML. The specification is translated into SQL statements for the target database. Liquibase supports a range of different databases.

Liquibase defines several constructs that encapsulate the schema changes. On the top level, a changelog is composed of schema changes called changesets. A changeset, on the other hand, captures several schema changes that are intended to be applied in a single transaction. Some examples of schema changes that Liquibase supports include adding, dropping or renaming a column. For each schema change, Liquibase also creates rollback statements to revert changes with ease.

The master changelog keeps track of all schema changes. Liquibase uses the master changelog to determine which changes have and which changes have not been applied yet. Any changelog that should be applied has to be included in the master changelog as follows:

```
<?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-
   4.7.xsd">


    <!-- each changelog corresponds to a new version of the database schema -->
    <include file="db/changelog/db.changelog-v1.xml"/>
```

[1] https://www.liquibase.org/

```
    <include file="db/changelog/db.changelog-v2.xml"/>
    <include file="db/changelog/db.changelog-v3.xml"/>

</databaseChangeLog>
```

The schema changes are recorded in a separate table called *databasechangelog*. Before migrating the database, Liquibase has to acquire a lock using the *databasechangeloglock* table. This ensures that only one Liquibase instance performs operations at any moment in time. After applying a schema change, the change is registered in the *databasechangelog* table which tells Liquibase that this change does not need to be applied anymore. Additionally, a rollback request looks at the last schema change to determine the rollback statement.

Liquibase is open-source and exposes endpoints that can be extended to implement new schema changes or add implementations for an unsupported database. This research produces a Liquibase plugin that overrides the base schema changes. To override an existing schema change, we need to specify a higher priority level. The plugin needs to be added to the lib folder to be able to use it. When reading a change specification, Liquibase scans the lib folder for implementations of that schema change and executes the one with the highest priority.

4.2 Kubernetes cluster

For this research, a Kubernetes cluster has been constructed [2]. It runs a Postgres cluster that is configured by Crunchy Data's Postgres Operator (PGO) [3]. PGO manages the database, ensures replication and provides automatic failover. In the master-slave replication architecture, a load balancer is required to forward write queries only to the master database. This functionality is provided by Pgpool-II. To test the expand-contract pattern, a Spring application sits in front of the Pgpool-II instance and receives HTTP requests through a REST API. The step-by-step approach to perform a blue-green deployment, as described in Section 3.6, will be applied to ensure correctness of the applied pattern.

The entire architecture of the Kubernetes cluster is depicted in Fig. 4.2.1. The figure shows that high availability is achieved in the form of redundant containers. The services act as load balancers and distribute incoming requests equally across all instances. The architecture is meant to be as representative as possible. A similar architecture is used in the production environment at ING. This ensures that the results of our tests and benchmarks are accurate and trustworthy.

It should be noted that limited resources were available for this research. The Kubernetes cluster was run on a single machine, being a laptop, using `minikube` to simulate the different nodes [4]. Consequently, the overall performance could be influenced by other processes running on the machine in parallel. We take this into account when evaluating the results. The laptop specifications can be found in Appendix A.

[2] `https://github.com/coenvk/kubernetes-postgresql`

[3] `https://access.crunchydata.com/documentation/postgres-operator/latest/`

[4] `https://minikube.sigs.k8s.io/docs/`

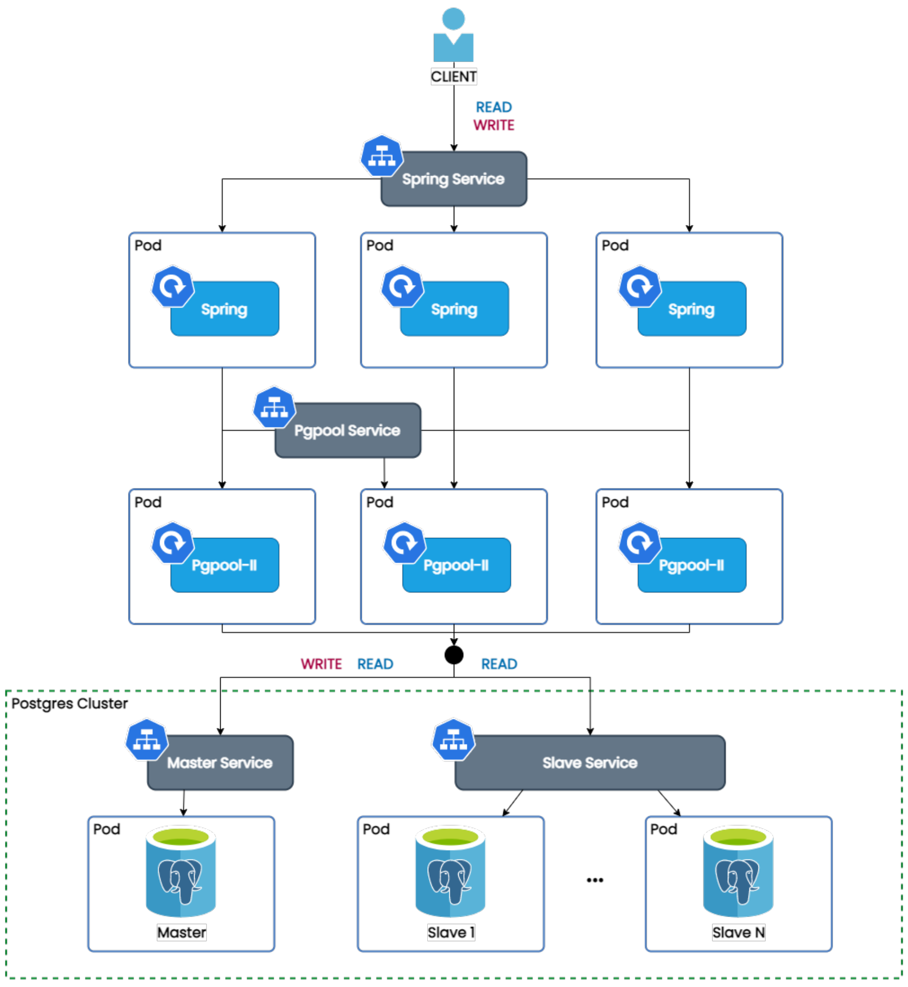


FIGURE 4.2.1: Architecture of the Kubernetes cluster that is used to test the expand-contract pattern.

4.3 HammerDB benchmarks

HammerDB is a benchmarking tool that we use to measure the latency and throughput of database transactions [5]. The Transaction Processing Performance Council (TPC) has specified a benchmark for online transaction processing (OLTP) called TPC-C. Based on the TPC-C specification, the TPROC-C benchmark has been created that is suitable for HammerDB. Dijkstra has also made use of HammerDB for his benchmarks [18].

Before running the benchmark, HammerDB creates a Postgres database. The database represents a warehousing system where customers send in orders. The benchmark automatically generates orders in bulk to provide a constant workload. The orders are processed and update the stock of the warehouses. The TPROC-C database is, in addition to benchmarking, used to debug `liquibase-zd`.

[5]`https://www.hammerdb.com/`

Chapter 5

Expand-contract pattern implementation

This chapter describes how several schema migrations have been implemented in `liquibase-zd`. The first section describes features specific to the plugin and how they can be used. The next sections describe the schema changes that have been implemented. The findings in this research and the research by Dijkstra can be used to provide implementations of the expand-contract pattern for other schema migrations or other versioning tools.

5.1 Plugin features

`liquibase-zd` replaces the original schema change by a custom change that executes the required individual changes.

5.1.1 Switch between expand and contract

By means of a Liquibase property, we can specify whether we are in the expand or in the contract phase. Based on this property, the corresponding changes are generated and executed. The property can take on one of the following values: *disabled*, *expand*, *contract*. The first value specifies that the original Liquibase implementation of the schema change should be applied. The property can be used 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-
        4.7.xsd">

    <property name="zd-strategy" value="expand" global="false"/>
```

```
5       <changeSet id="rename.column.expand" author="Coen">
6           <renameColumn tableName="customer" oldColumnName="name"
            newColumnName="fullname"/>
7       </changeSet>
8   </databaseChangeLog>
```

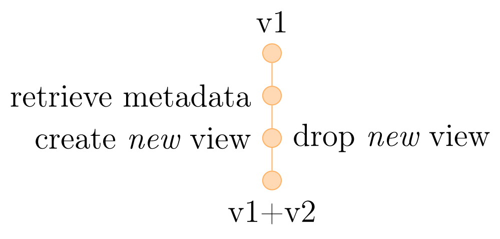
The property is defined in line 4 as a non-global property. This ensures that the value of the property can be changed in a different file. If the property is excluded, it is assumed that the zero-downtime plugin is disabled. Generally, the idea is to create two files: one for the expand step and one for the contract step. The definition of the schema change should be exactly the same. The only difference is the property value and the identifier of the changeset.

5.1.2 Metadata retrieval

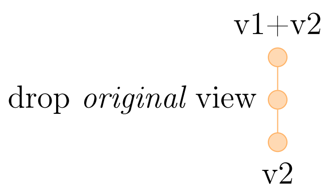
Some schema changes require additional information about database objects. When renaming a view, as explained in Section 5.2, the expand phase consists of adding the same view with a different name. The definition of the original view is required to create an exact copy. There are two options to acquire this information. The first option is that the software engineer provides the definition, or other metadata, to the schema change. However, this has a lot of disadvantages. The metadata has to be exactly the same as the original specification. Moreover, this goes against the transparency requirement **R3**. Instead, prior to the schema migration, the view definition is retrieved from the Postgres system catalogs. Other metadata, such as table structures and column constraints, can also be obtained from the system catalogs. The upcoming sections will refer to metadata retrieval when information is required about a database object in order to complete a specific schema change.

5.1.3 Rolling back changes

After applying the expand step, the database is in a mixed-state. The database supports both the old and the new database version. From this state, it is still possible to rollback the schema change and revert to the old database version. However, after the contract step has also been processed, the database only supports the new version and cannot be reverted as described in the example in Section 3.6. `liquibase-zd` implements this behavior. Liquibase keeps track of the schema changes in the *databasechangelog* table. A rollback looks at the last schema change and tries to revert the operation. If the last schema change was applied using the contract strategy, the rollback operation will throw an exception. Not only does this prevent the contract operation from rolling back, it also ensures that the expand operation cannot be reverted. Instead, a forward roll can be applied to revert the database back to the old state.



(A) Expand and rollback steps



(B) Contract step

Figure 5.2.1: Expand-contract pattern for renaming a view.

5.2 Rename view

Before going into more complicated schema changes, we start with a schema change for which the expand and contract steps are more straightforward, to show how the pattern is applied. The expand-contract pattern is the simplest for renaming a view. In Fig. 5.2.1, the pattern has been illustrated. We refer to the original view and the renamed version as *original* and *new* respectively. The same names will be used when describing other schema changes. The expand phase for this schema change consists of two steps:

1. The first step is to retrieve metadata about the view. Specifically, we want to know the definition: the query that retrieves the data.
2. Finally, a new view is created with the updated name based on the retrieved definition.

After the expand step, the database contains a view with the original name and the same view with the new name. The rollback operation corresponds to the schema changes displayed on the right in Fig. 5.2.1a, applied in reverted order. The operation can be rolled back by dropping the renamed view, while the contract step drops the original view.

5.3 Rename column

The next change that has been implemented involves renaming a column. The pattern is applied similarly to Dijkstra's implementation. However, as Dijkstra also describes in his future work section [18], his implementation does not work for composite foreign key constraints. In our implementation, we solve this issue by acquiring metadata about the database column that describes its foreign key constraints.

Expand

The expand phase consists of the following steps:

1. The first step is to retrieve metadata about the original column. This can include the data type, default value, and constraints such as nullability, uniqueness, foreign keys and primary keys.

2. Using the retrieved metadata, we can now add the new column. The new column is required to be nullable, since it will not contain any values for the existing records.

3. A unique constraint on the original column is applied to the new column. This can already be done before synchronizing the two columns, since Postgres allows multiple null values in a unique column.

4. The new column needs to contain the same data as the original. To make this guarantee, we use triggers to synchronize the two columns. An update to the original column is also propagated to the new column. An insert operation is propagated in both directions.

5. Any records that have not been synchronized to the new column are copied. This is done is small batches to ensure that this operation does not block access to the table for too long. The plugin provides the ability to adjust the batch size. Between each batch update, Liquibase waits for a specified amount of time to not overload the database. A custom Liquibase change, as illustrated in Section 5.6, implements this functionality.

6. The next step is to copy any remaining constraints from the original column to the new column. The retrieved metadata contains all the information we need for this. Special care is taken when adding foreign keys. In the foreign key definition, the plugin replaces the name of the original column with the updated name. Consequently, a composite foreign key constraint would also be added correctly. Moreover, by default, adding a foreign key validates the existing records against the constraint. In a million-row table, this can be a computationally expensive operation. Therefore, Postgres provides functionality to disable validation on existing data [1]. This feature is currently only available for foreign key and check constraints. The plugin makes use of this functionality to improve performance.

7. In addition, if the original column has a default value, the same default value is added to the new column. By doing this after converging the new column, there are no empty records that Postgres would have to initialize with the default value. This process would keep a lock on the table and could last a substantially long period of time.

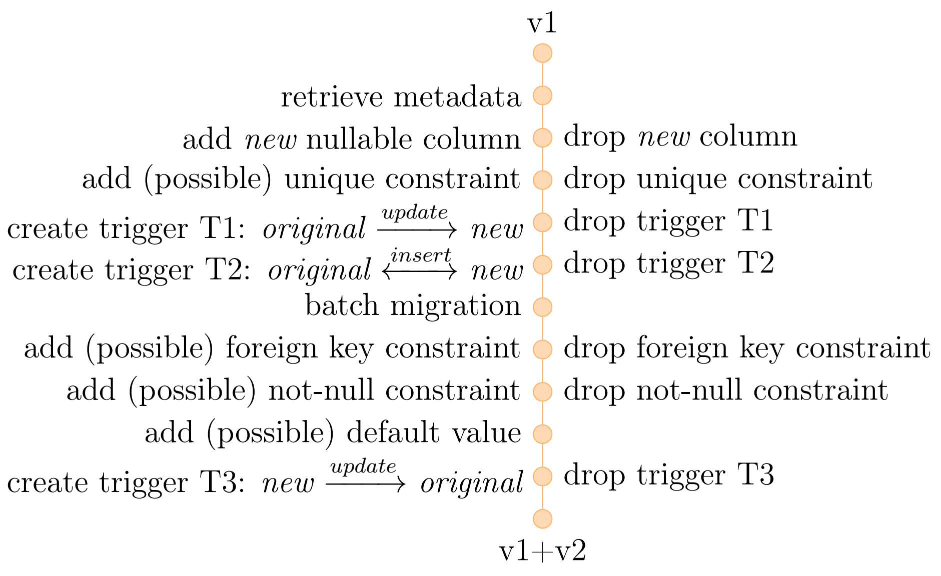
8. Finally, a trigger is added that updates the original column when the new column is updated.

Rollback

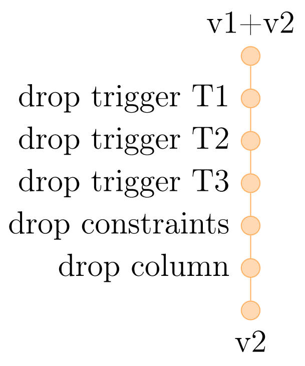
While the contract phase has not been processed yet, it is possible to rollback from the mixed-state. This is done in the following manner:

1. The triggers that were created during the expand phase are dropped.

[1] `https://www.postgresql.org/docs/current/sql-altertable.html`


v1
retrieve metadata
add *new* nullable column
drop *new* column
add (possible) unique constraint
drop unique constraint
create trigger T1: *original* $\xrightarrow{update}$ *new*
drop trigger T1
create trigger T2: *original* $\xleftrightarrow{insert}$ *new*
drop trigger T2
batch migration
add (possible) foreign key constraint
drop foreign key constraint
add (possible) not-null constraint
drop not-null constraint
add (possible) default value
create trigger T3: *new* $\xrightarrow{update}$ *original*
drop trigger T3
v1+v2

(A) Expand and rollback steps



(B) Contract step

Figure 5.3.1: Expand-contract pattern for renaming a column.

2. Additionally, the constraints on the new column are dropped.
3. Finally, the new column is dropped in order to return to the initial database state.

Contract

After the expand phase has successfully been processed, as described in Section 3.6, and the operation is not rolled back, the original column can be removed. As a result, the database version is incremented. As mentioned before, the contract phase can be enabled by setting the *zd-strategy* property to *contract*. The migration is similar to the rollback operation, but is applied to the original column instead:

1. All the triggers that synchronize the two columns are dropped.
2. Additionally, the constraints on the original column need to be dropped before we can drop the column.
3. Finally, the original column is dropped.

5.4 Modify data type

Data type modification is one of the schema migrations that Dijkstra was not able to implement yet. As he suggests [18], it is recommended to do this similarly as to how a column can be renamed. The step that adds the new column would have to specify the new data type. This is how `liquibase-zd` implements data type modification.

There are some caveats to this implementation however:

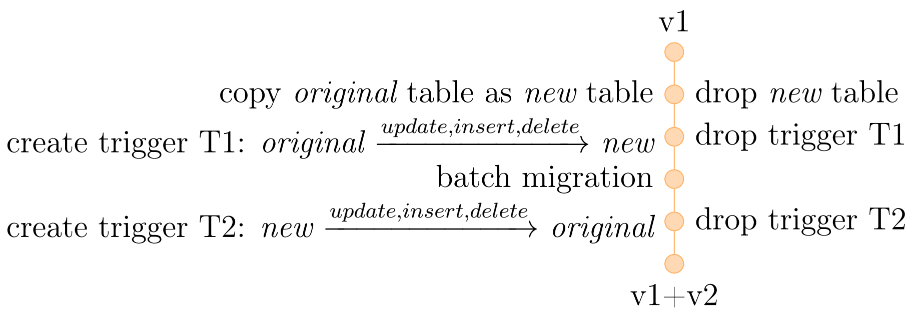
- Modifying the data type of a column requires it to be renamed. It is impossible for a table to have two columns with the same name, hence a new column is required to have a different name. This is not a large downside, but has to be taken into account. It means that any reference in the application code has to be updated as well. Another possibility is to work around it by renaming the column back to the original directly after modifying the data type, but this requires an additional (possibly long-lasting) operation.
- Additionally, it might not be possible to convert between data types. Postgres is able to convert *text* to *varchar* and vice versa. However, it is not possible to change the data type from *varchar* to *int*. An important use case for this schema migration is expanding the size of a *varchar* column. If a $varchar(10)$ column is expanded to 20 characters, the expand step can copy the data to the new column. However, copying data from the new column to the original could go wrong if a value of more than 10 characters is inserted. This situation would cause the triggers, that keep the columns synchronized, to raise an error. A possible workaround for this caveat is to put a limit on the size of the variable in the updated application code. Once the contract step has been applied, this limit can be eliminated.

5.5 Rename table

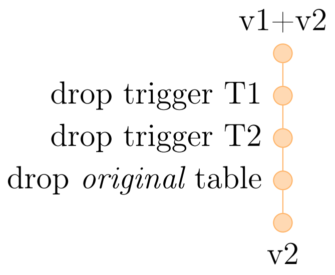
The final change that `liquibase-zd` covers involves renaming a table. Dijkstra has described how the expand-contract pattern can be applied to rename a table in an Oracle database. It is done using a table synonym. Unfortunately, Postgres does not have functionality for synonyms or aliases. Therefore, the pattern has to be applied in a different manner. In this section, we show two techniques that have been implemented.

5.5.1 Rename table using copy

Similarly to renaming a column, it is possible to create a renamed version of a table and keep the data synchronized. The main disadvantage is that, as opposed to single columns, it takes much more computational power to converge an entire table. Therefore, depending on the table size, it can take a large amount of time to apply this schema migration. It should be noted that the triggers also forward a delete operation to remove a record in both tables. Since a delete statement removes entire rows, this is not required for intra-table synchronization between columns.


v1
copy *original* table as *new* table
drop *new* table
create trigger T1: $original \xrightarrow{update,insert,delete} new$
drop trigger T1
batch migration
create trigger T2: $new \xrightarrow{update,insert,delete} original$
drop trigger T2
v1+v2

(A) Expand and rollback steps



(B) Contract step

FIGURE 5.5.1: Expand-contract pattern for renaming a table using a copy.

Some Postgres magic is required to duplicate a table. The below statement creates a new table with the structure of another table. In other words, all columns, constraints (except for foreign keys) and indexes are copied. The foreign key constraints are added by querying the metadata of the original table.

```sql
CREATE TABLE newTable (LIKE oldTable INCLUDING ALL);
```

After the new table has been constructed, the triggers and batch migration (see Section 5.6) ensure that both tables are kept in sync. It should be mentioned that an update statement is propagated as an upsert operation. In other words, if the row exists in the new table, it is updated. However, it is inserted if the row has not been added to the new table yet.

During testing, a problem was discovered that was not accounted for. The triggers, referred to as T1 and T2 in Fig. 5.5.1a, were recursively calling each other. When a record is updated in the original table, trigger T1 propagates it to the new table. Consequently, trigger T2 is fired which again activates trigger T1. To resolve this issue, we need to verify that a trigger is not fired from within another trigger. Postgres provides a system function called *pg_ trigger_ depth* which returns the current trigger nesting level. The trigger recursion can be prevented by verifying that the current depth is equal to zero before executing a trigger function.

5.5.2 Rename table using view

Another approach is to rename the original and create a view with the name of the original table in the expand phase. This method is much faster, but will block the table for a short period of time. The operation can be rolled back by dropping the view and renaming the table back, again blocking access to the table. The contract phase simply drops the view.

`liquibase-zd` implements both aforementioned techniques, referred to as *renameTableUsingCopy* and *renameTableUsingView* respectively, making it the responsibility of the developer to choose the one that best fits their needs. The first approach is more time consuming, but does not put a lock on the table. Vice versa for the latter approach.

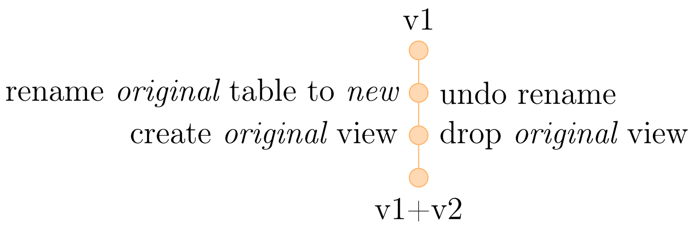
5.6 Batch migration

The batch migration change that is mentioned in Section 5.3 and Section 5.5, ensures that the copy of a database object converges to the original. While the triggers take care of new data, the batch migration change copies existing data. The result of this operation is that both database objects contain the same data. The plugin implements the batch migration change for columns and tables. The operation is similar to an *update* statement, but prevents Postgres from blocking access for an extensive amount of time. Instead, the batch migration change performs *updates* in small batches by limiting the number of records to update. This limit is set by the *chunkSize* parameter. In addition, the *sleepTime* can be specified as the time between consecutive executions of a batch update (in milliseconds). This feature has been added to prevent overloading the database and provide a time window for regular access from the application. Accordingly, the application is able to access the database parallel to the batch migration.

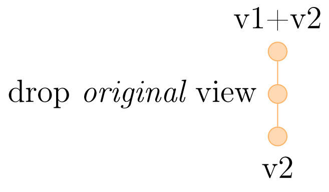
5.6.1 Intra-table batch migration

Dijkstra has implemented a batch migration change for columns in Oracle. This implementation has been adjusted for Postgres. It is a custom Liquibase change and can be used in a changeset as follows:

```
<changeSet id="batchMigration" author="Coen">
    <customChange
    class="liquibase.ext.change.update.BulkColumnCopyChange">
```



(A) Expand and rollback steps



(B) Contract step

Figure 5.5.2: Expand-contract pattern for renaming a table using a view.

```
        <param name="tableName" value="tableName"/>
        <param name="fromColumns" value="oldColumn"/>
        <param name="toColumns" value="newColumn"/>
        <param name="chunkSize" value="1000"/>
        <param name="sleepTime" value="0"/>
    </customChange>
</changeSet>
```

The source and destination columns can be specified as comma-separated lists. The number of source and destination columns need to be equal and they cannot overlap. In other words, it is not possible to copy data from columns (A,B) to (B,C).

The query that a batch migration executes requires a row identifier. In an Oracle table, each record can uniquely be identified by a $rowId$. They should never be used as primary keys, since they can be reused when a row gets deleted. Nonetheless, in our use case, it can be guaranteed that no problems arise [18]. Postgres has a similar identifier called *ctid*. However, the *ctid* of a row changes when its content gets updated [2]. Therefore, a more reliable row identifier is the primary key. A metadata query is used to retrieve the primary key of a table. The following statement is an example of what the batch migration change executes:

```sql
UPDATE tableName
SET newColumn = oldColumn
WHERE (primaryKey) IN
(
    SELECT primaryKey FROM tableName WHERE (newColumn IS NULL AND
    oldColumn IS NOT NULL) LIMIT 1000
);
```

5.6.2 Inter-table batch migration

The batch migration change is slightly different for entire tables. We have seen that copying data between two columns requires updating an existing record. To copy data from one table to another, insert statements are used. Postgres allows us to insert records found in a SELECT query. Therefore, the idea is to retrieve all rows from the original table which do not exist in the new table. These are records that were not caught by the synchronization triggers. Inserting these records into the new table ensures that it converges. Correspondingly, the SQL query looks as follows.

```sql
INSERT INTO newTable
SELECT a.* FROM oldTable a
LEFT OUTER JOIN newTable b USING (primaryKey)
```

[2]https://www.postgresql.org/docs/current/ddl-system-columns.html

```sql
WHERE b.primaryKey IS NULL
LIMIT 1000;
```

Chapter 6

Evaluation

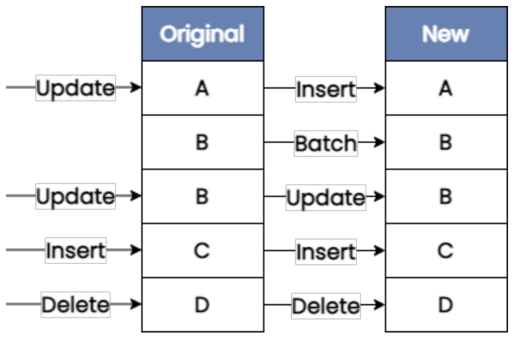
`liquibase-zd` has been evaluated. In the next section, we show by deduction that the schema migrations function as expected. This will help to verify requirement **R7**. The sections that follow, describe how the HammerDB benchmarks are run and use the results to verify **R5**. Non-blocking schema changes ensure that application deployments, and specifically blue-green deployments, provide zero-downtime. That is, the user does not experience a change in application performance. The final section, discusses how the requirements from Section 2.1 have been tested against the expand-contract pattern. All in all, this shows us that the expand-contract pattern correctly implements the schema migrations.

6.1 Functional correctness

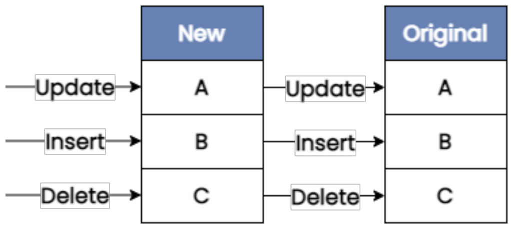
The expand-contract pattern has been implemented for several schema migrations with zero-downtime in mind. Before we can evaluate the blocking behavior of the schema changes, it is important that we prove that the schema changes are functionally correct. That is, the result after applying the expand-contract pattern is equivalent to the result of executing the original schema change. We have seen that the expand phase tries to create a copy of the original database object with the required schema changes. The contract phase continues by removing the original database object. Therefore, showing functional correctness of the expand-contract pattern is equivalent to proving that the original and new database object are identical before the contract step is applied. The database objects have to both contain the same data and be identical in schematic structure. We prove both requirements in the next subsections.

6.1.1 Data equivalence

Most of the implemented schema changes use triggers and a batch migration change to synchronize data between the two database objects. Only when renaming a table using a view this is not required, since a view by definition contains the same data as the table it is defined from. Data can be manipulated in three different ways: updating existing data, inserting new data, and deleting data. Triggers are added that propagate each of these data changes to the new database object. The batch



(A) Copy data from original database object to newly created database object.



(B) Copy data from new database object to original database object.

Figure 6.1.1: Data change propagation to prove functional correctness of implemented schema migrations.

migration change takes care of data that was inserted or updated before the triggers were created.

There are several executions of DML statements that can result from this construction which are described in Fig. 6.1.1a. Data insertions are directly caught by a trigger and propagated to the new database object. They can happen before or after the batch migration change. The same goes for deletion operations. On the other hand, data updates should be handled differently before and after the batch migration change. If we update data before the batch migration change is executed or has propagated that data, the updated data is inserted into the new database object. If the batch migration change has already added the corresponding data, the same update is applied on the new database object. From these statements, we can deduce that all DML changes on the original database object are forwarded to the new database object.

Finally, triggers are created that ensure data manipulations on the new database object are registered by the original database object. These triggers did not miss any data changes, since the new database object can only be manipulated after the expand phase has finished. Consequently, the two database objects contain equivalent data which concludes the prove.

6.1.2 Identical schematic structure

Secondly, we show that the schematic structure of the two database objects is identical. This is achieved by retrieving metadata about the original database object, such as constraints and indexes, and applying them to the newly created database object. Although, the current implementation does have some shortcomings. For example, when modifying the data type of a column, the column is renamed as well. This is unavoidable, since a table cannot contain two columns with the same name. In addition, when renaming a primary key column, the current implementation does not apply the primary key constraint to the new column. It should be possible to implement this feature, but it is quite cumbersome and is out of the scope of this research. Foreign keys can depend on the primary key and it can therefore not easily be changed without breaking requirement **R11**. Moreover, a primary key cannot simply be altered, but needs to be removed and recreated with the adjustments.

6.2 Experimental setup

As mentioned earlier, HammerDB has been used to run performance benchmarks and monitor the blocking behavior of the implemented schema changes. From the results, we can determine whether the schema changes ensure zero-downtime deployments. This section describes how the benchmarks have been configured to get reliable results.

The benchmarks have been tested on an asynchronously replicated Postgres database, as well as a synchronously replicated database. The isolation level is kept at the default value of *read committed*. To determine the blocking behavior of the schema changes, the throughput and latency have been measured during execution of the migrations. The throughput or TPM is defined as the number of transactions that are processed within a minute. HammerDB queries the Postgres Statistics Collector [1] every second for the number of transactions committed and rolled back on the database using the following query:

```
SELECT SUM(xact_commit + xact_rollback) FROM pg_stat_database
```

The TPM is calculated as the difference between two subsequent results of this query, multiplied by 60.

The latency is measured in microseconds and defines the time between receiving a transaction request and sending back a response. The first benchmark that we run is a baseline. The baseline runs without a schema migration and shows database performance under regular use. The rest of the benchmarks can be compared with the baseline to determine the effects of the schema migration. Each benchmark runs for about 20 minutes and conforms to the following procedure:

1. Before each benchmark, the database schema needs to be re-initialized to get an equivalent starting state. The initial database schema is called *v1*. The TPROC-C database has been instantiated with ten warehouses.

2. Once the database is ready, HammerDB starts a warm-up period that builds up the workload to reach a constant transaction rate. Immediately when the warm-up period has finished, the TPM and latency are recorded every second. This is when the benchmark officially starts. HammerDB is configured with a read-write ratio of 3:1. The following steps are also depicted in Fig. 6.2.1.

3. After five minutes, the schema migration is executed. Depending on the type of schema change and the size of the database, it can take some time to finish up the migration. The migrated database is in the mixed-state conforming to both version *v1* and *v2*. By examining the TPM and latency during the schema migration, we can verify **R5**.

[1] `https://www.postgresql.org/docs/current/monitoring-stats.html`

4. The schema migration is followed with five minutes of rest during which we can examine whether the mixed-state is correctly implemented; the transactions fired by HammerDB should not fail. This validates **R7** in one direction: it shows that the mixed-state captures *v1*, but not necessarily *v2*.

5. Finally, the database is rolled back to version *v1* to determine the blocking behavior of the rollback operation as well.

Similarly, the contract phase has been benchmarked by starting from schema version *v2*. HammerDB generates a workload for *v2* while the migration contracts the database schema from a mixed-state to *v2*. In step 4 of the above enumeration, we can validate that the mixed-state captures *v2*.

In addition to the aforementioned benchmarks, the performance of the batch migration change has been measured in a separate test. The duration of the schema migration is highly dependent on the duration of the batch migration change. For different sizes of the database, based on the number of warehouses, the batch migration change was executed and its execution time was recorded. The goal of these benchmarks is to determine whether the relation between table size and execution time is linear or exponential. In case the relation is exponential, it could lead to long-running schema migrations for vast databases. Even though, we are aiming at non-blocking schema changes, the duration of the migration is an important factor as well. Generally, small changes are deployed and will be deployed often. Therefore, the deployment pipeline should not take up an immense amount of time.

6.3 Results

This section discusses the results of the benchmarks. Every schema migration has been tested on an asynchronously replicated database as well as a synchronously replicated database. For each benchmark, the recorded TPM and latency are depicted in a chart. The dotted lines indicate the start or end of a migration using the same color scheme as in Fig. 6.2.1. Some results have been excluded in this section, since they either are not interesting enough to discuss, or they do not deviate from the priorly discussed results. All of the results can be found in Appendix B.

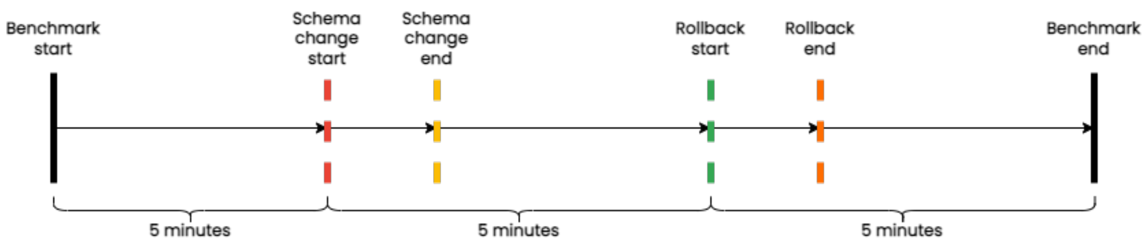


Figure 6.2.1: Execution process of the HammerDB benchmarks.

6.3.1 Asynchronously replicated database

Baseline

Before testing each schema change, a baseline has been measured. The baseline can be found in Fig. 6.3.1. The top chart displays the TPM over time, while the bottom chart shows the latency. The baseline is used to compare the other results to. Therefore, no schema change is executed during this benchmark; the results show the database performance under normal use.

From the results, it can be determined that the TPM fluctuates a lot. The reasoning behind this could be that the Kubernetes cluster is running on one machine that is also processing other programs. However, the baseline does show a constant trend and never goes to zero. Moreover, we can argue about the results and draw conclusions.

The chart that depicts the latency shows several metrics. The minimum and maximum latency speak for themselves. In addition, the $P50$, $P95$ and $P99$ latency are presented. The $P50$ latency, for example, tells us that 50% of the incoming requests is handled faster than that. Similar definitions can be provided for the $P95$ and $P99$ values. The baseline shows that the maximum latency is peaking sometimes. However, the $P99$ latency is at a constant rate. Therefore, these are merely outliers which might be caused by vacuuming or other automatic database processes.

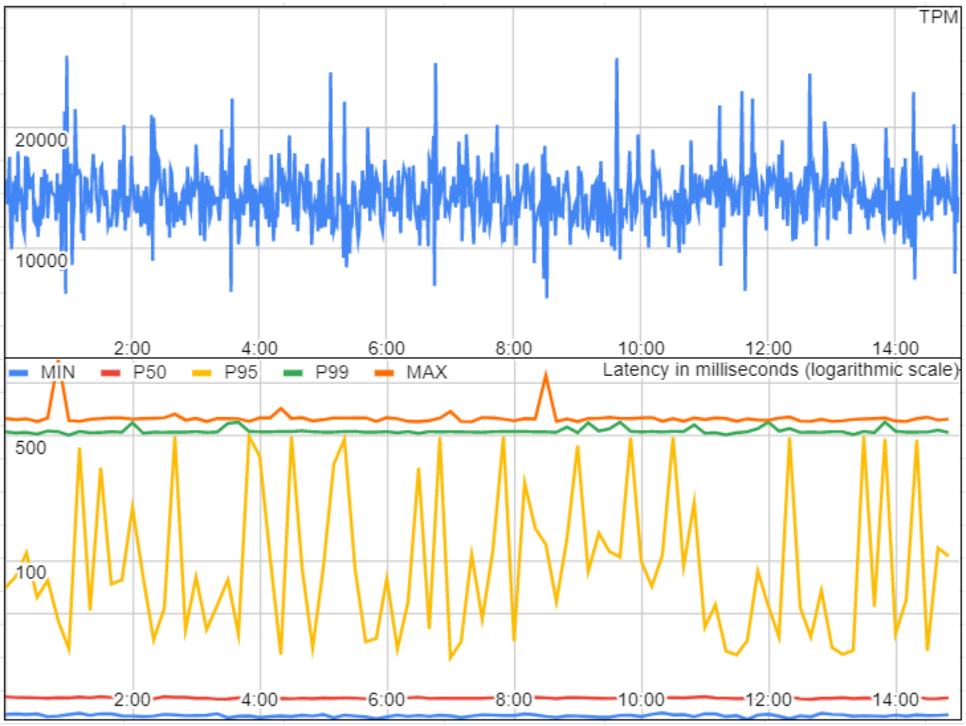


Figure 6.3.1: Baseline

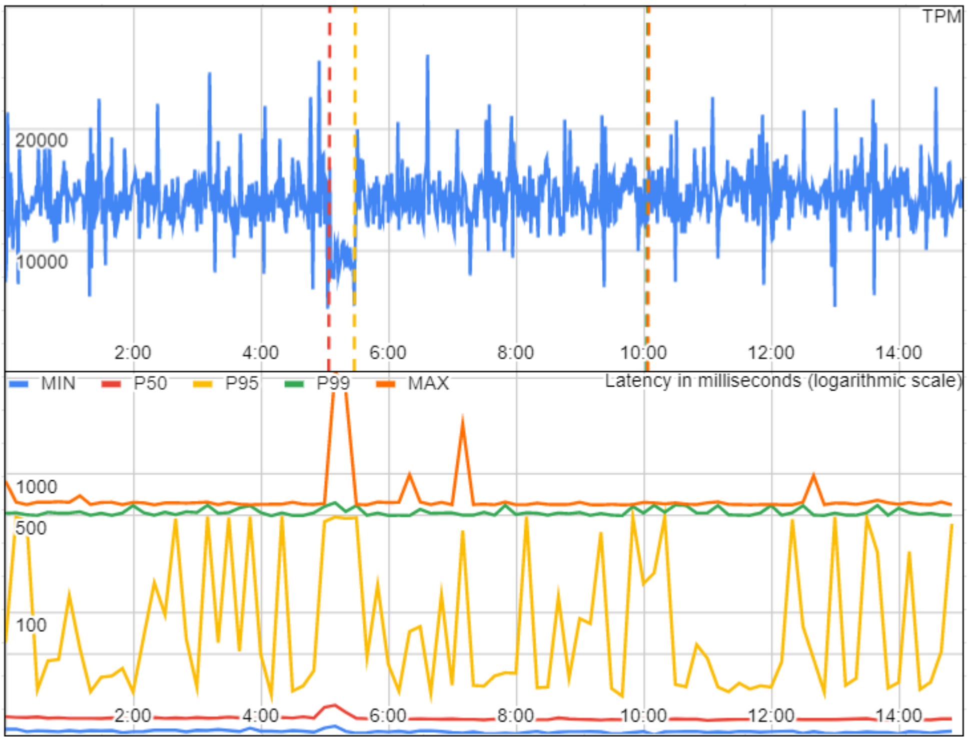


Figure 6.3.2: Rename column with a batch size of 1.000

Rename column

The schema change to rename a column is the first schema change that has been benchmarked. The results are presented in Fig. 6.3.2. During the schema migration, the throughput decreases by about 35%. In other words, the change is non-blocking. The schema change involves adding a new column which does acquire a lock on the corresponding table, but this operation takes close to no time. In addition, the synchronization triggers and the batch migration possibly have an effect on the throughput. As we can see in Fig. 6.3.3, the throughput deduction can be reduced by decreasing the batch size for the batch migration. For a batch size of 250, the throughput only decreases by about 15% while the migration only takes a couple of seconds longer.

The rollback operation involves dropping the newly created column which acquires an exclusive lock on the table blocking all access. Nonetheless, Postgres does not completely remove the column, but rather marks it as unused. Vacuuming operations take care of eventually removing unused data. Consequently, the rollback operations is quite brief and should not be noticeable. Moreover, there is no approach to drop a column without locking.

Similarly to the baseline, the latency chart does show several high maximum values. There is a very large peak at five and a half seconds during the schema migration. Nevertheless, this is only a maximum value, and the $P99$ value is at a constant level. The $P95$ value is peaking during the schema migration. All in all, it can be deduced that the schema migration has an effect on the latency. However, the effect is quite minimal, since we are talking about less than half a second increase. Moreover, these effects are not applicable anymore after the schema

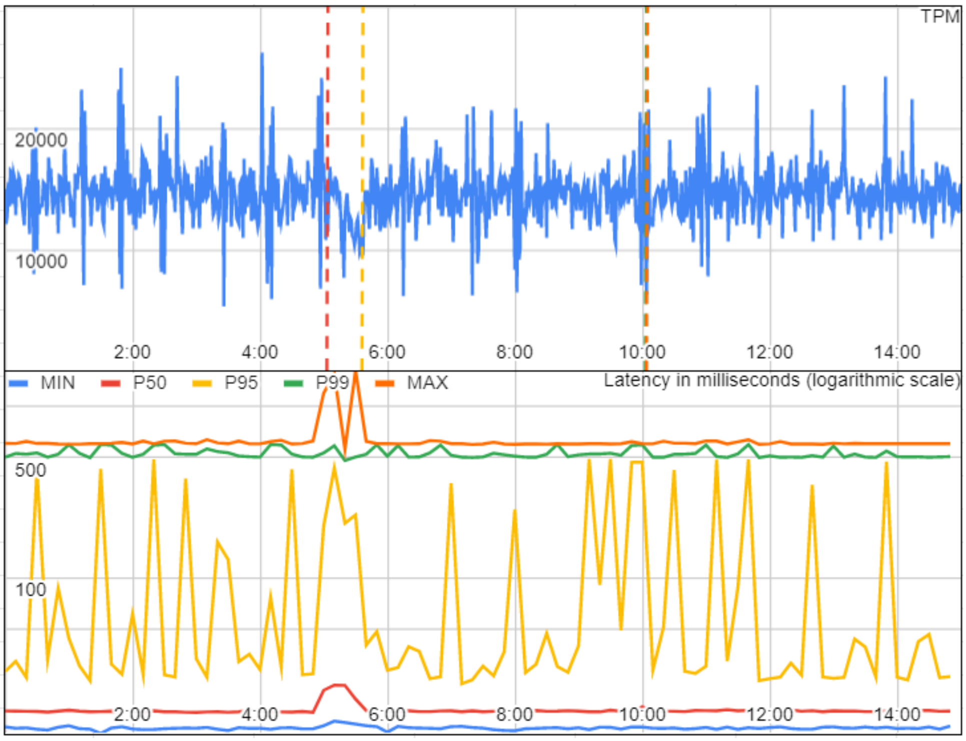


Figure 6.3.3: Rename column with a batch size of 250

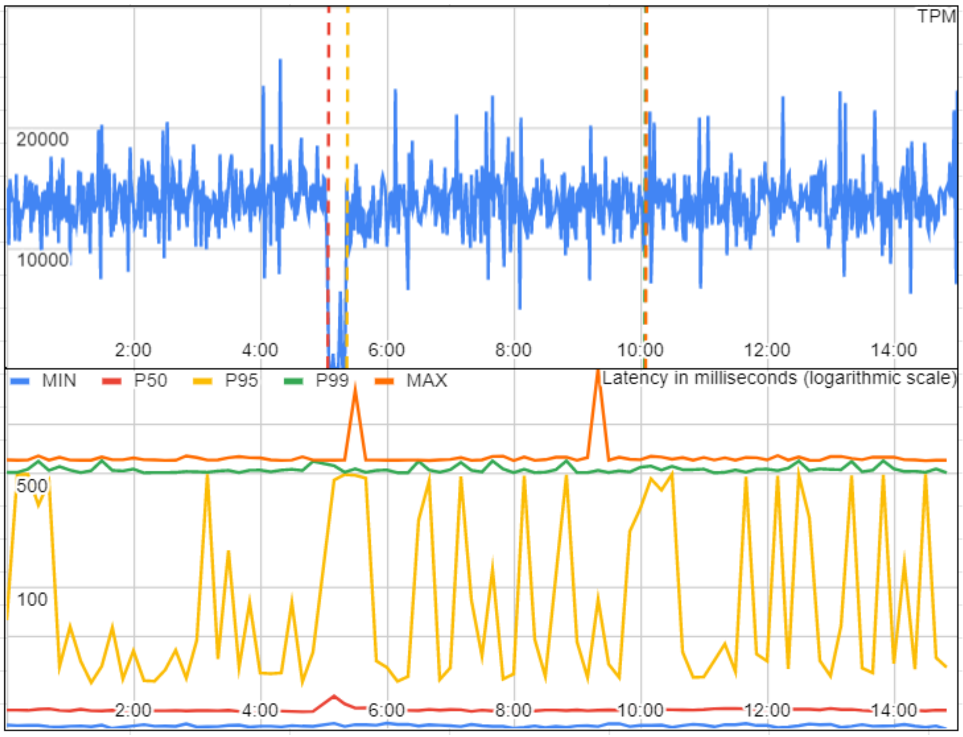


Figure 6.3.4: Rename column with a batch size of 100.000

migration completes.

To see the effects of a blocking schema change, the same schema migration was executed with a vast batch size of 100.000 for the batch migration. The results are

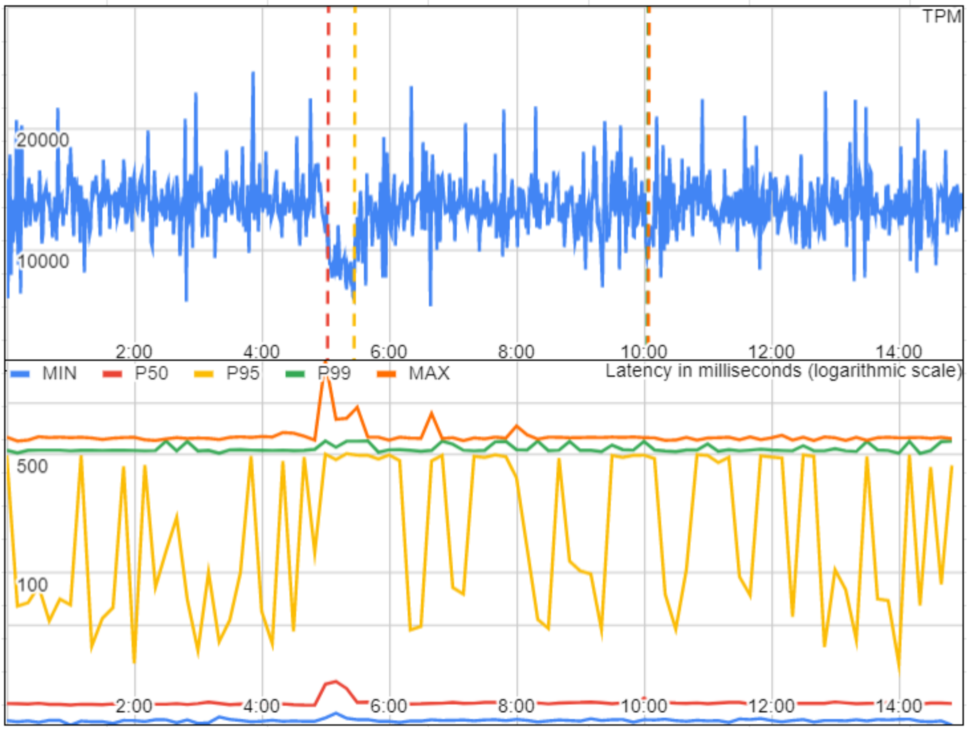


Figure 6.3.5: Modify data type with a batch size of 1.000

shown in Fig. 6.3.4. One batch covers one third of the table, and therefore blocks most transactions. The chart shows that the TPM goes almost all the way to zero. The effects on the latency are similar to using the default batch size of 1.000.

Modify data type

Modifying the data type of a column has been benchmarked next. A column with the *charactervarying* datatype has been expanded by double the number of allowed characters. It makes sense to compare the results against the results for renaming a column, since the schema migration is so similar. As seen in Fig. 6.3.5, it seems that modifying the data type has similar effects on the throughput and latency.

Rename table using copy

Renaming a table can be done in two ways: using a copy of the original table or using a view. We started with benchmark the prior technique. The results in Fig. 6.3.6 show that the TPM has a similar reduction as to renaming a column or modifying the data type of a column. However, the throughput deduction is not as large. The reasoning behind this could be that a different kind of SQL statement is executed. The batch migration change copies that data to another table. Therefore, it performs a SELECT statement on the original table and inserts this data in the newly created table. Since the SELECT operation does not require a lock, the original table can still be used in parallel. The throughput still experiences a slight deduction, which could be a result of the additional load on the table.

There is no significant impact on the latency: the $P95$ value shows a small peak,

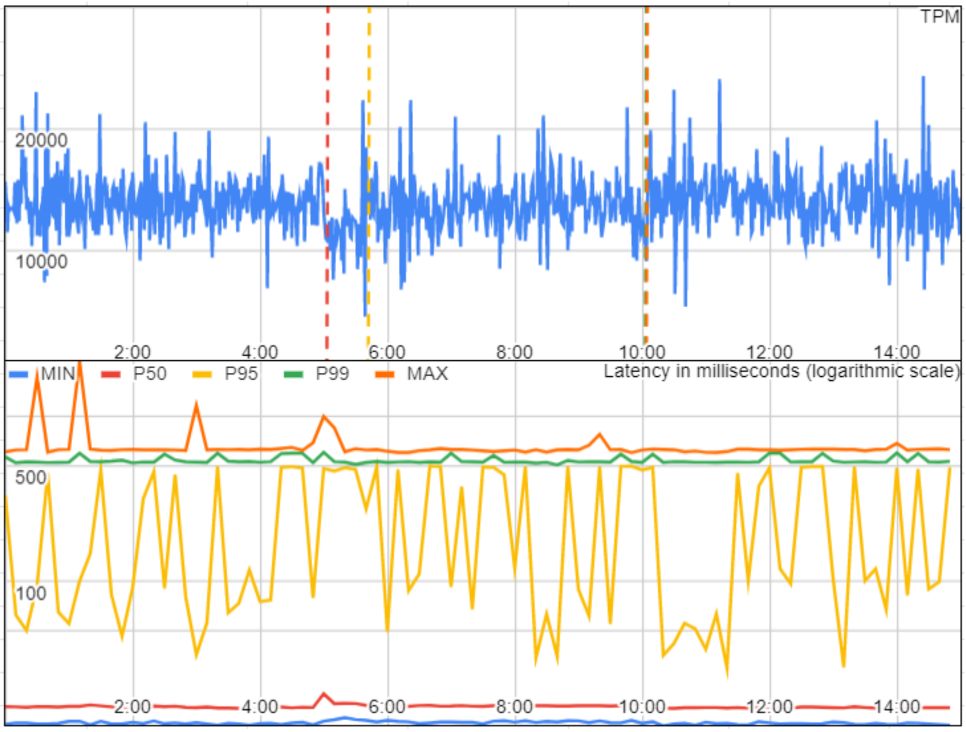


Figure 6.3.6: Rename table using copy with a batch size of 1.000

but is not worrisome. Similar peaks appear under regular operation of the database. Moreover, 95% of the requests are processed faster than this.

The rollback operation drops the new table. The HammerDB workload does not touch the new table, since it only exists in schema version $v2$. Therefore, the schema migration can be rolled back unnoticed.

Rename table using view

Finally, we look at the benchmark results for renaming a table using a view. The schema migration acquires a lock when renaming the table, but even under a high workload this operation is not perceived. Moreover, the charts in Fig. 6.3.7 show that the throughput and latency are not influenced.

The rollback operation drops the view that is only visible in schema version $v2$. Accordingly, the throughput and latency are not affected by the rollback operation.

The previous observations lead us to the following conclusion: it is preferred to rename the table in combination with a view, instead of creating a copy. Copying the table drops the throughput and ultimately leads to a longer migration time, while directly renaming the table and creating a view is almost instant.

6.3.2 Synchronously replicated database

Baseline

The synchronously replicated database seems to behave very distinctively. The baseline in Fig. 6.3.8 shows large fluctuations in both the throughput and latency.

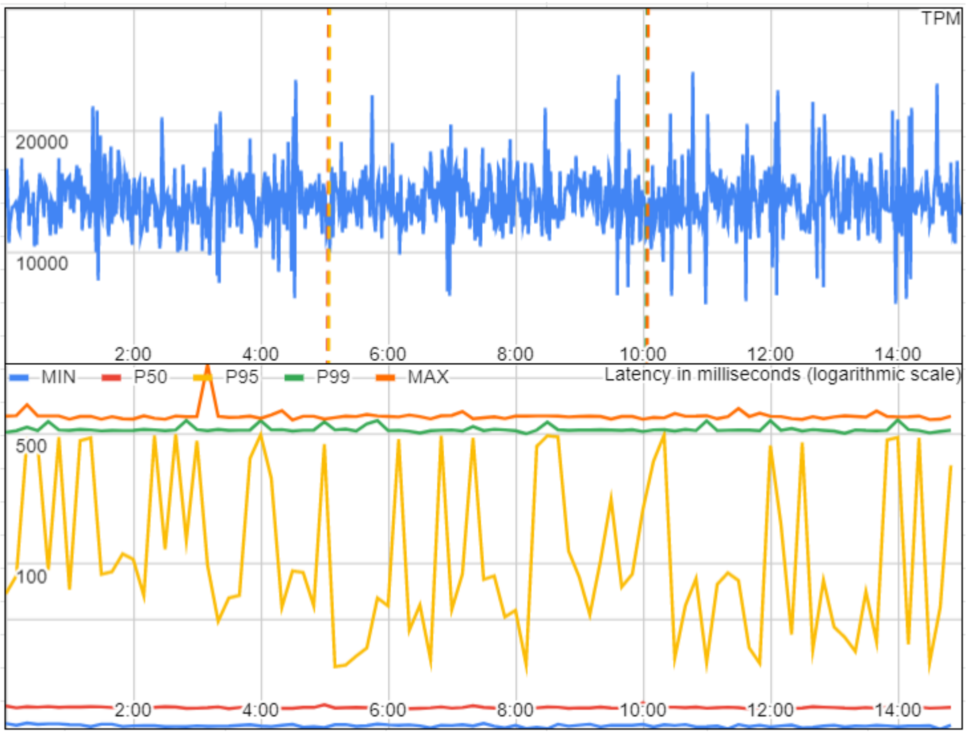


Figure 6.3.7: Rename table using view with a batch size of 1.000

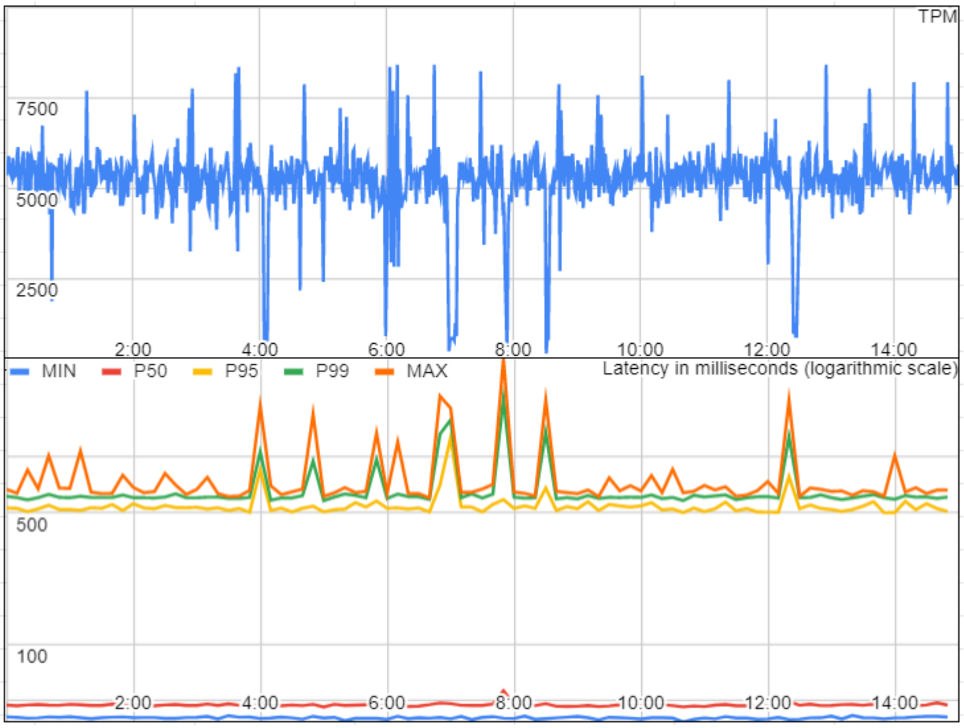


Figure 6.3.8: Baseline

Overall, the TPM is lower than the TPM in Fig. 6.3.1. Similarly, the *P*95 latency does not go below half a second. Sometimes the throughput sinks to a couple hundred transactions per minute. Moreover, the exact moment that the TPM drops,

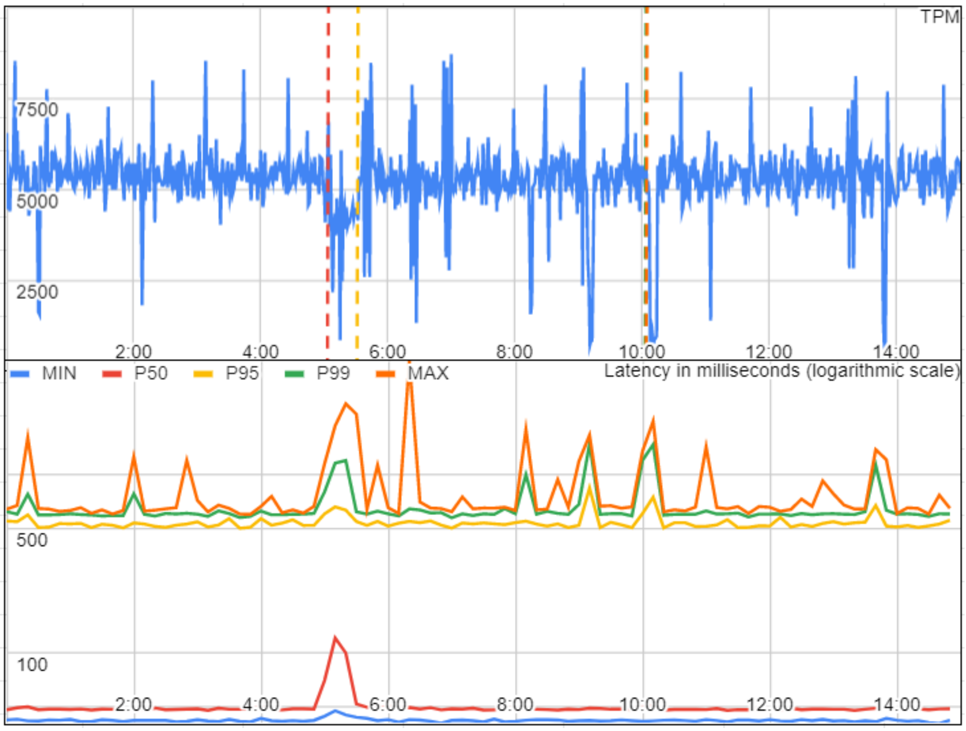


Figure 6.3.9: Rename column with a batch size of 1.000

the latency peaks. The reason could be that a long running transaction is executed which keeps a lock. The master database has to wait for the slaves to commit the result, before committing itself. Consequently, other transactions that access the same table are blocked, increasing their respective latency.

Rename column

The results for renaming a column in a synchronously replicated database are depicted in Fig. 6.3.9. Overall, the schema migration affects the database performance in a similar manner as to in an asynchronously replicated database. There is a modest decrease in the TPM. All the latency metrics, even the $P50$ have a small peak during execution of the schema migration. It can be concluded that the schema change is non-blocking. Nonetheless, the user might experience an extra second of processing time for their requests.

Remaining schema changes

The results for the remaining schema changes can be found in Appendix B.2. We have chosen not to discuss these results here, since they show a similar pattern as to the results in the asynchronously replicated database. The aspects that do differ are comparable to the differences for renaming a column which have been discussed in the previous section. Additionally, Appendix B can be referred to for an overview of all benchmark results.

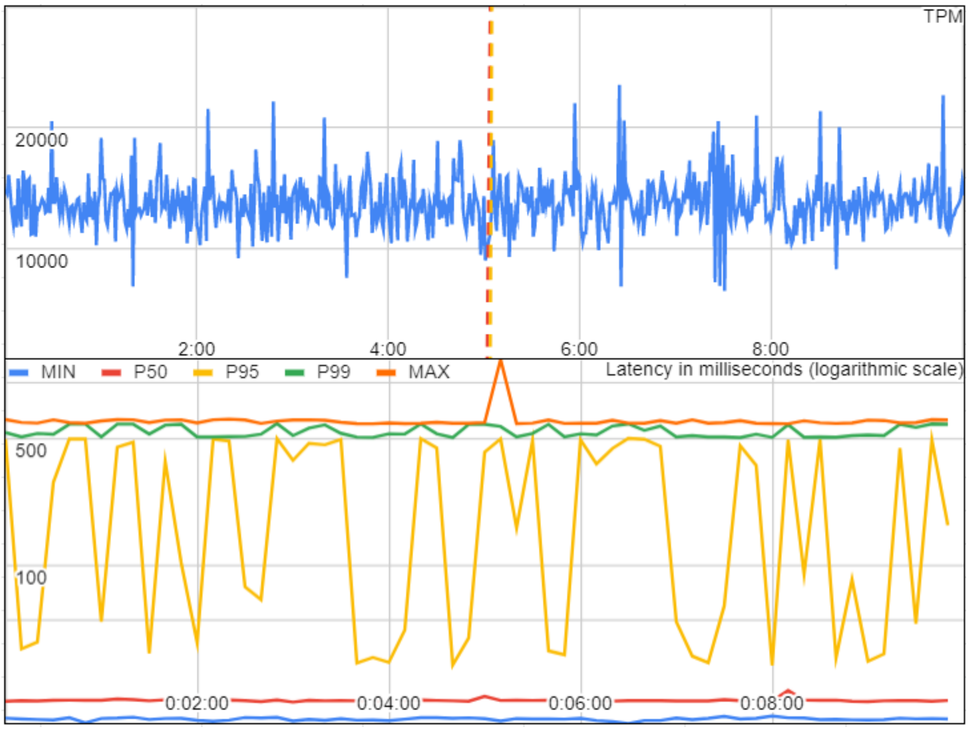


FIGURE 6.3.10: Rename column

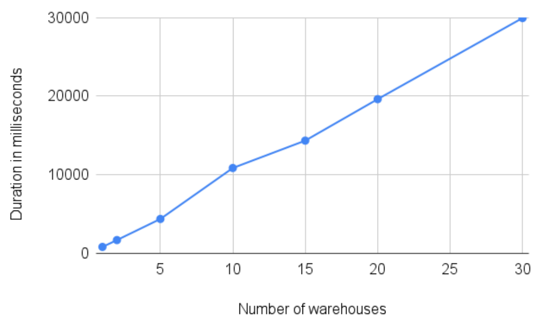
6.3.3 Contract from mixed-state

The previous results show that the mixed-state, as a result of the expand step, captures schema version *v1*. To ensure the mixed-state also captures *v2*, we benchmarked the contract step. In Fig. 6.3.10, the results for renaming a column can be found. We can see that the throughput is at a constant level and that it remains constant after the contract step has been applied. Moreover, it can be concluded that the mixed-state is correctly implemented. The latency peaks shortly after the contraction, but there is no reason to assume that it is related.

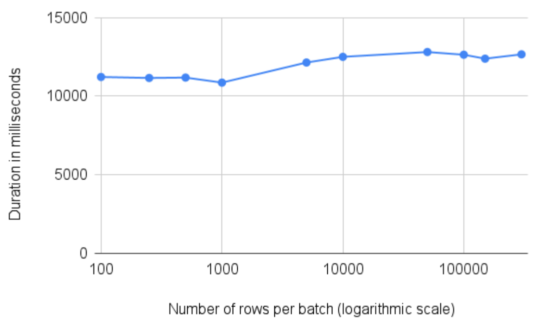
Similar results are produced for the other schema migrations. The charts can be found in Appendix B.3.

6.3.4 Batch migration change

The second set of benchmarks evaluates the execution time of the batch migration change. Specifically, the execution time has been measured for various table sizes as well as for several batch sizes. Both the intra-table and inter-table variants have been tested on the *customer* table. The table size is a multiple of the number of warehouses in the HammerDB database; there are 30.000 customers per warehouse. The results have been averaged over five benchmark runs. The reasoning behind these benchmarks is that there is a trade-off between batch size and throughput as seen in Section 6.3. However, if the run time significantly decreases for larger batch sizes, this trade-off could be acceptable. That is left as a decision to the software engineer.

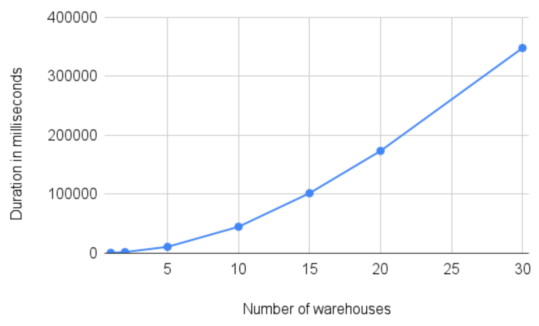


(A) Warehouses

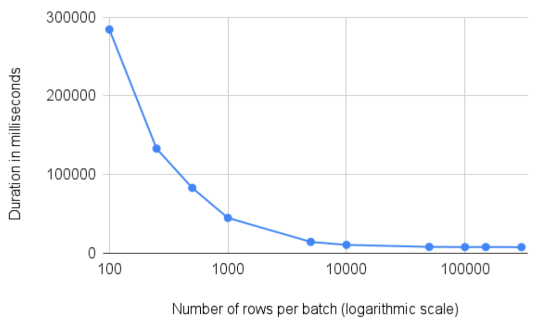


(B) Batch sizes

Figure 6.3.11: Benchmark results for the intra-table batch migration change.



(A) Warehouses



(B) Batch sizes

Figure 6.3.12: Benchmark results for the inter-table batch migration change.

Intra-table batch migration

The results of the first benchmark are depicted in Fig. 6.3.11a. The chart shows a linear relationship between the number of warehouses and the execution time. In other words, if the table is twice as large, the migration takes twice as long. Moreover, the duration of updating one batch, does not depend on table size.

The execution time seems to be constant for different batch sizes as seen in Fig. 6.3.11b. Nonetheless, the batch migration change appears to be slightly faster for smaller batch sizes where the minimal execution time has been achieved for batches of 1.000 rows. Since greater batch sizes result in less batches overall, it can be concluded that Postgres' update statement linearly increases in run time with the row count.

All in all, these benchmark results show that it is better to use the intra-table batch migration variant with a low batch size. Its execution time is not dependant on the table size or batch size. Furthermore, a lower batch size guarantees better performance as shown in Section 6.3.

Inter-table batch migration

Fig. 6.3.12 shows the results for the inter-table batch migration. Unlike for the intra-table batch migration, the relation between the number of warehouses and the execution time is non-linear. In fact, the chart shows an exponential distribution. It takes longer to insert data into a large database, because it has to be re-indexed every time. Updating the indexes can take some time when the index is vast. However, it should be taken into account that these benchmarks batch migrate the entire table. In practice, the batch migration change is used in conjunction with the expand-contract pattern which starts off with an optimized table copy operation. The batch migration change only inserts rows that are not initially copied or not caught by the synchronization triggers.

On the other hand, the run time decreases more than exponentially for larger batch sizes. It seems that Postgres has optimized insertion in bulk. Therefore, increasing the batch sizes for vast tables could be a solution to decrease the run time. Nonetheless, it should be tested how much the throughput and latency are affected to determine whether it is worth the trade-off.

6.4 Requirement validation

The requirements from Section 2.1 are listed again below. It is important that the plugin meets these requirements. Therefore, each requirement has been tested or can be verified by implementation of the plugin.

R1 **Integration**. `liquibase-zd` has been tested in a Kubernetes cluster. This cluster is representative in the sense that it follows a similar architecture to microservices that ING implements. Each of the other requirements has been tested within this cluster.

R2 **Detachable**. The plugin can be disabled by excluding the *zd-strategy* property from the Liquibase specification, or by setting its value to *disabled*. When the plugin is disabled, the original schema changes, as implemented by Liquibase, are applied.

R3 **Transparent**. It has been taken into account that the software engineer should be able to use the plugin without any additional knowledge. The plugin is designed such that the software engineer does not have to update their changeset specifications. The only alteration that has to be carried out is the addition of the *zd-strategy* property.

Nonetheless, some schema changes do require additional information from the developer. To modify the data type of a column, for example, a (new) name has to be provided for the added column. Furthermore, there are two implementations that rename a table which can be referred to as *renameTableUsingCopy* and *renameTableUsingView*.

R4 **Generalizable**. Even though, the expand-contract pattern has only been implemented for the Postgres database, the implementation can be extended to support other databases. Liquibase is database-independent and generates SQL statements based on the active database. In fact, the implementation of the plugin has been built on top of the original Liquibase changes. Solely the creation of triggers, and the retrieval of metadata, is specific to Postgres. These implementations can be adjusted to support other databases. Consequently, it is feasible to use the plugin with other databases as well. Moreover, the expand-contract pattern can similarly be implemented for other database versioning tools such as Flyway.

R5 Non-blocking schema changes. To evaluate the blocking behavior of the expand-contract pattern, the HammerDB benchmarks were set up. In the previous sections, several test cases have been determined, and the latency and throughput have been measured during execution of the schema migrations. The results have been discussed in Section 6.3.

R6 Schema changesets. It is possible to apply several schema changes in one go. First of all, the *update* command from Liquibase executes any migration that has not been applied yet. Therefore, several expand-contract changes can be applied sequentially. However, it becomes more complicated when zipping together the expand and contract phases such that all expand phases finish before contracting any change. If two changes overlap, the created triggers could interfere. Therefore, this is considered an anti-pattern. Generally, changes should be kept small. Another advantage is that bugs can be found and fixed more easily.

R7 Concurrently active schemas. In Section 6.1, we have shown that the database objects, conforming to version *v1* and *v2*, are identical. Both database versions can be used in parallel and will remain equivalent.

The HammerDB benchmarks, in addition, show that both database versions are supported in the mixed-state. The benchmarks put pressure on the database in the form of a continuous workload. The workload consists of SQL transactions that interact with the initial database version. Therefore, the benchmark should result in an error if the mixed-state is incorrect. From the results in Section 6.3, it can be determined that HammerDB did not experience any failure. In other words, the mixed-state is correctly implemented for each schema change.

R8 Integrity. The plugin ensures that integrity constraints are not violated. Triggers are added that propagate DML changes from the original database object to a copy. It is assumed that the software engineer correctly implements constraints on the original object. Therefore, if the original object does not violate any constraint, neither does the copy. When the copy converges, metadata about the constraints is acquired in order to add the same constraints to the object copy. Finally, triggers can be added in the other direction. This keeps both database objects in sync. Consequently, integrity constraints will not be violated.

R9 **Schema isolation**. The expand step puts the database in a mixed-state. In this state, the database supports both the schema prior to and after the migration. It is in the application code that the schema versions are differentiated. The old application is only able to see the old schema version while the updated application sees the schema after the migration.

R10 **Non-invasive**. Usage of the plugin does not involve any change in the application code. The plugin can be added to Liquibase in the `lib` folder and is instantly usable.

R11 **Resilience**. For each schema migration, the plugin implements a rollback operation. Rollbacks can take place while the database is in the mixed-state. In Section 6.3, it is shown that the rollback operation does not block access to the old database schema which can be queried in parallel. After contracting the database schema, it is not possible to perform a rollback.

Chapter 7

Discussion

In this chapter, some limitations are discussed that influenced the research process and possibly the results. Additionally, several recommendations are given for future work that could enhance the results of this research.

7.1 Limitations

One of the limitations that had a lot of impact on this research is that we were limited to a single laptop. We were unable to get access to a cluster of servers. Therefore, the Kubernetes cluster had to be run on a single machine. Using minikube, it was possible to simulate an entire cluster on one laptop. Nonetheless, it was inevitable that performance would be affected. During research, we noticed that the throughput goes down a lot when the laptop is processing other heavy operations parallel to the Kubernetes cluster. Moreover, the throughput went down if the HammerDB benchmark was run on multiple threads. Therefore, the benchmarks were run on a single thread.

Nevertheless, the results show that the throughput is not completely constant; it fluctuates more than expected. This particularly shows that the performance of the database cluster is influenced by the fact that everything runs on a single laptop. The latency, specifically the max-value, peaks sometimes as well with unclear cause. Moreover, these peaks happen both during and exclusive of the schema migration. Specifically, the baseline shows high fluctuations as well. During schema migrations the peaks seem to last longer. Therefore, they could be correlated to the schema migration, but this is not a conclusion we can draw as of now.

The setup used in this research is meant to be representative of the microservices architecture used at ING. Nonetheless, to be more representative, the benchmarks should be run on a multi-node cluster. This would either give us more certainty about our results, or it could lead to different results.

In addition, there are some limitations to the implementation of the expand-contract pattern. First of all, modifying the data type of a column requires renaming the column as well. This is a requirement of the implementation. However, this schema change could also have been implemented by creating a copy of the original table where the respective column takes on the new data type. A disadvantage is that the batch migration step would take longer and more storage is being used.

The final limitation is that the current implementation does not support primary key columns. When renaming a primary key column, the expand step adds a new column which would become the new primary key column. Unfortunately, it is impossible to directly modify a primary key constraint. It would require removing the current primary key constraint and adding a new one. There are various reasons why this process is not as straightforward as it may seem. The current primary key constraint might have dependent foreign keys that would have to be removed first and added back at the end. Moreover, while the primary and foreign key constraints are dropped, it is possible for invalid data to be inserted. Usually, application source code expects automatically incremented primary keys. Therefore, the application does not provide any primary key value when inserting new data. Consequently, an insert operation could fail while the primary key is absent. For these reasons, the current implementation of `liquibase-zd` throws an exception when trying to apply a schema change to a primary key column using the expand-contract pattern. This problem is out of the scope of this research, because of its complexity, and is left as future work.

7.2 Future work

There are several research problems that were either out of the scope of this research, or have not been done due to time limitations. These are left as suggestions for future work:

FW1 The first recommendation for future research involves adding support for primary key constraints to the current implementation of the expand-contract pattern. As explained in the previous section, this process could be quite complex and further research has to be done to find a good and stable solution.

FW2 During this research, we have tested performance of the expand-contract pattern in a master-slave replicated database cluster. From the results, we can formulate an answer to research question **RQ2.1**. In addition, we wanted to run the benchmarks on a multi-master replicated database to answer the following question: to what extent can the expand-contract pattern be applied when using master-master replication? Unfortunately, due to time constraints we have not been able to. Therefore, this research question is left unanswered and is considered for future work. CrunchyData's Postgres Operator (PGO) only supports master-slave replication. Nonetheless, there are several solutions that allow to setup multi-master replication in a Postgres database cluster. One solution that seems promising is `BDR` by EnterpriseDB [1]. Once the multi-master replicated database cluster is setup, similar HammerDB benchmarks can be run to evaluate performance of the expand-contract pattern.

FW3 As the limitations in Section 7.1 suggest, it is important that the expand-contract pattern gets tested on a production-ready system. A Kubernetes

[1]`https://www.enterprisedb.com/docs/bdr/latest/`

cluster should be setup across several server nodes to spread the workload and ensure that performance is not impacted by machine-related limitations. It is expected that the benchmark results will be different, but this has to be investigated in an extensive research.

FW4 Another aspect that could be tested is whether there is a difference between the performance under write operations in contrast to read operations. This research ran the HammerDB benchmarks with a read-write ratio of 3:1. Therefore, the results do not say much about performance under a high write load. Benchmarks can be setup to only pressurize the database with write operations. The results can be evaluated to come to a conclusion.

FW5 Dijkstra has already implemented the expand-contract pattern for Oracle and provides several change templates. This research has extended the pattern to Postgres and implements a Liquibase plugin that takes care of the logic. In future work, `liquibase-zd` can be extended to support other databases as well. Furthermore, there are other database versioning tools such as Flyway that are extensible, for which an implementation can be contributed.

Chapter 8

Conclusion

In data-intensive applications, schema migrations need to be deployed often. This happens alongside updates of the application code. Dijkstra suggested implementing an expand-contract pattern that can be used in combination with blue-green deployments. Moreover, a non-blocking implementation of the expand-contract pattern can ensure zero-downtime deployments.

This research has extended Dijkstra's implementation of the expand-contract pattern by producing a Liquibase plugin that implements the pattern. The plugin provides zero-downtime schema migrations for Postgres databases. In addition to the schema migrations for which Dijkstra provided change templates, the expand-contract pattern was implemented for modifying the data type of a column. Moreover, the plugin retrieves metadata about the database objects to ensure that constraints such as foreign keys are persisted. To test the performance of the expand-contract pattern, a Kubernetes cluster was setup with a master-slave replicated Postgres database. The plugin was benchmarked using HammerDB on both an asynchronously and synchronously replicated database. The benchmark results show that the expand-contract pattern is non-blocking for all schema changes. However, as explained in Section 7.1, the implementation should be further exploited to gain more certainty about its performance. In combination with a blue-green deployment, the pattern can be applied to provide zero-downtime deployments.

The final two sections describe the deliverables of this research and answer the research questions that have been specified in Section 1.2.

8.1 Deliverables

Besides this paper, this research contributes the following products:

1. A **Kubernetes cluster setup** running a master-slave replicated Postgres database. The setup has been described in-depth in Section 4.2 [1].

2. **HammerDB benchmark results** showing both the throughput and latency during execution of the schema migrations. The expand-contract pattern was

[1]The final version of the setup can be found at `https://github.com/coenvk/kubernetes-postgresql`

both benchmarked on an asynchronously as well as a synchronously replicated database. Furthermore, the batch migration change has been benchmarked separately to gain insight in its behaviour.

3. A **Liquibase plugin** called `liquibase-zd` that implements the expand-contract pattern for several schema changes [30].

8.2 Research answers

This research is an extension of Dijkstra's research. The implementation tries to improve upon Dijkstra's technique. Dijkstra has written change templates while we have written a Liquibase plugin that extends the default implementation of the schema migrations and incorporates the expand-contract pattern (**RQ1.1**). The plugin can be enabled by specifying either *expand* or *contract* for the *zd-strategy* property corresponding to the expand and contract step respectively. Dijkstra has mentioned that modifying the data type of a column and support for composite foreign keys could be improvements upon his implementation. These have been implemented in `liquibase-zd` (**RQ1**). In Section 7.1, some limitations to the plugin have been discussed.

Nonetheless, the plugin is fully functional and has shown great performance under a high workload. `liquibase-zd` was tested on a Kubernetes cluster running a master-slave replicated Postgres database (**RQ2**, **RQ2.1**). Due to time constraints, it has not been tested on a master-master replicated database. Instead, this is left as a recommendation for future research. As explained in Section 7.2, `BDR` provides a multi-master solution for replicated Postgres databases that could be used for this.

HammerDB has been used to benchmark `liquibase-zd` (**RQ3.1**). It is a widely used tool to performance test a database. Since it sends thousands of transactions to the database every minute, thus putting it under a lot of pressure. The pressure is supposed to simulate a real life scenario. Therefore, disregarding the limitations, the results are trustworthy and can very well be used to evaluate our implementation. HammerDB has no problem running benchmarks on a fully-replicated database (**RQ3.2**). Overall, the benchmark results show that the expand-contract pattern does not entail any downtime, but configuration of the batch size should be taken into account. Altogether, the latency is mostly unaffected and the throughput goes down slightly depending on the batch size. In case the TPM is set too high, the throughput can go down to zero. In general, asynchronously replicated databases show better performance and asynchronous replication affects the throughput and latency less than synchronous replication. However, the effects of the expand-contract pattern are similar on both the synchronously as well as asynchronously replicated database (**RQ3.3**).

Bibliography

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems (extended version), 2014.

[3] Peter David Bailis. Coordination avoidance in distributed databases. *Coordination avoidance in distributed databases*, 2015.

[4] Souvik Bhattacherjee, Gang Liao, Michael Hicks, and Daniel J. Abadi. *BullFrog: Online Schema Evolution via Lazy Evaluation*, page 194–206. Association for Computing Machinery, New York, NY, USA, 2021.

[5] Zoltán Böszörményi and Hans-Jürgen Schönig. *PostgreSQL Replication*. Packt Publishing, 2013.

[6] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[7] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[8] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6, 2016.

[9] Alina Buzachis, Antonino Galletta, Antonio Celesti, Lorenzo Carnevale, and Massimo Villari. Towards osmotic computing: a blue-green strategy for the fast re-deployment of microservices. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2019.

[10] Lianping Chen. Microservices: Architecting for continuous delivery and devops. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 39–397, 2018.

[11] Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. Automating database schema evolution in information system upgrades. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98, feb 2013.

[13] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, aug 2008.

[14] Michael De Jong and Arie Van Deursen. Continuous deployment and schema evolution in sql databases. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 16–19, 2015.

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

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

[17] Rodolfo Miranda Barros Diego Quirino Silva, Everton Gomede. Strategies for zero-downtime releases: A comparative study. 2016.

[18] Jorryt-Jan Dijkstra. *Zero-downtime schema changes*. PhD thesis, University of Twente, Netherlands, September 2021.

[19] Martin Fowler. Bliki: Bluegreendeployment, Mar 2010.

[20] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[21] Martin Kleppmann. A critique of the cap theorem. 09 2015.

[22] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[23] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, nov 1992.

[24] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 262–273, 2016.

[25] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. Zero downtime release: Disruption-free load balancing of a multi-billion user website. In *Proceedings of the Annual Conference*

of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 529–541, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Vinicius Feitosa Pacheco. *Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices.* Packt Publishing, 2018.

[27] Chaitanya K. Rudrabhatla. Comparison of zero downtime based deployment techniques in public cloud infrastructure. In *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 1082–1086, 2020.

[28] Danilo Sato. Bliki: Parallelchange, May 2014.

[29] Yangjun Sheng. Non-blocking lazy schema changes in multi-version database management systems. *PhD diss., Carnegie Mellon University Pittsburgh, PA*, 2019.

[30] Coen van Kampen. coenvk/liquibase-zd: v4.8.0, June 2022.

[31] Maarten van Steen and Andrew S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, 2016.

[32] Lesley Wevers, Matthijs Hofstra, Menno Tammens, Marieke Huisman, and Maurice van Keulen. Analysis of the blocking behaviour of schema transformations in relational database systems. In Morzy Tadeusz, Patrick Valduriez, and Ladjel Bellatreche, editors, *Advances in Databases and Information Systems*, pages 169–183, Cham, 2015. Springer International Publishing.

[33] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 206–215, 2000.

[34] Bo Yang, Anca Sailer, Siddharth Jain, Angel Reyes, Manu Singh, and Anirudh Ramnath. Service discovery based blue-green deployment technique in cloud native environments. pages 185–192, 07 2018.

[35] Y. Zhu. Towards automated online schema evolution. 2017.

Appendices

Appendix A

Machine specifications

Operating system	Microsoft Windows 11 Home (10.0.22000 Build 22000)
Processor	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s), 12 Logical Processor(s)
Memory (RAM)	16GB
Disks	512GB NVMe SSD + 1TB HDD

TABLE A.1: Specifications of the laptop that has been used in this research. The laptop simultaneously ran a Kubernetes cluster and the HammerDB benchmarks.

Appendix B

Benchmark results

B.1 Asynchronously replicated database

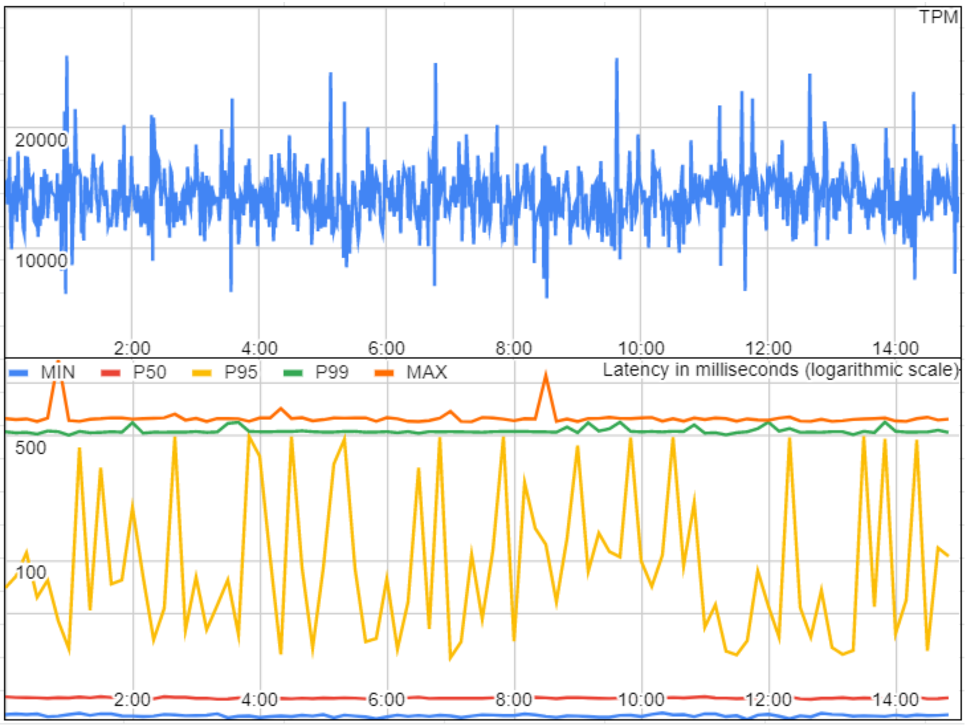


FIGURE B.1.1: Baseline

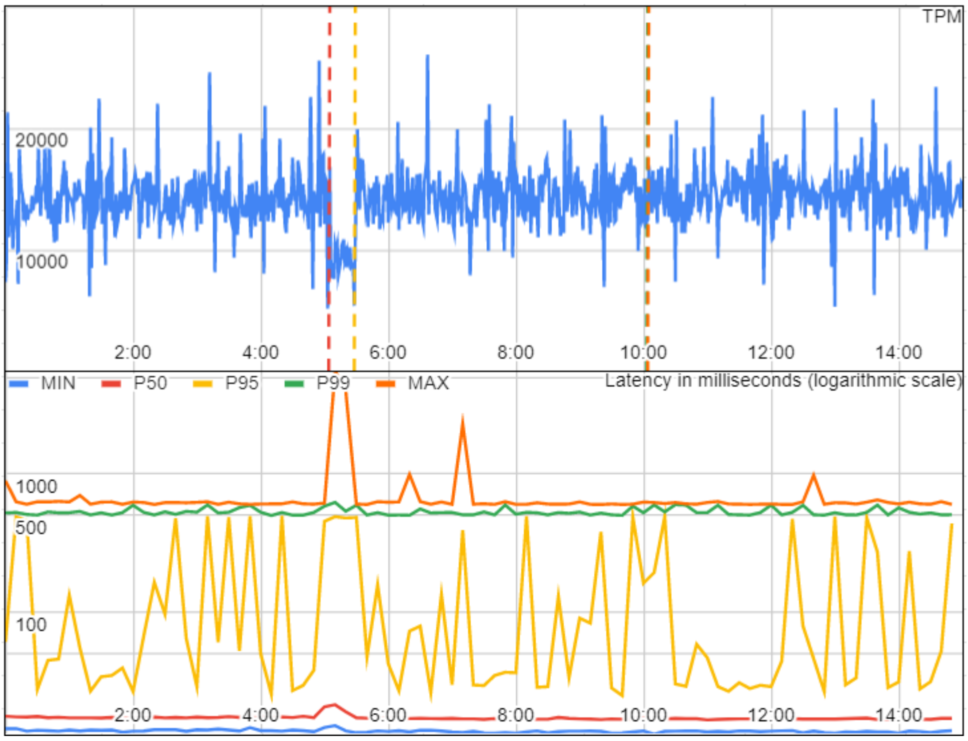


Figure B.1.2: Rename column with a batch size of 1.000

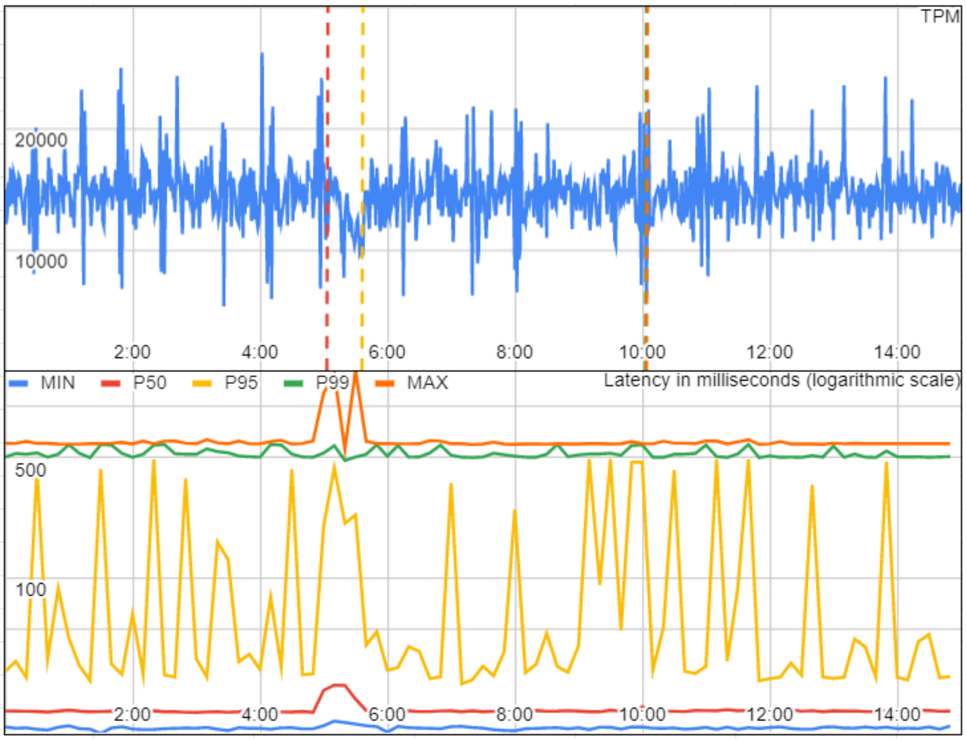


Figure B.1.3: Rename column with a batch size of 250

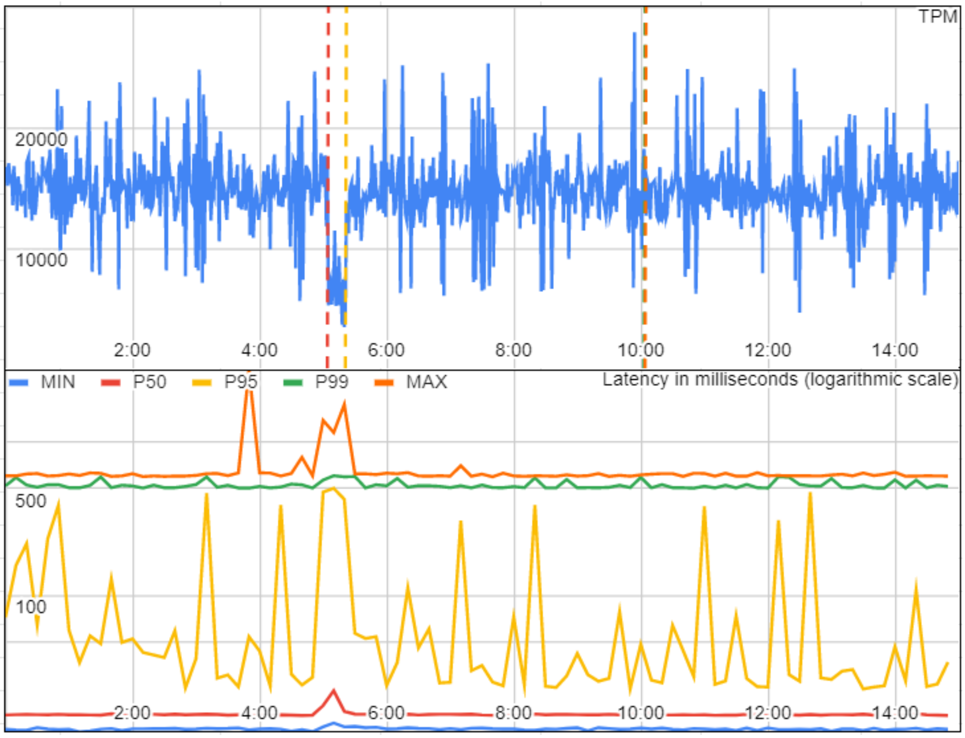


FIGURE B.1.4: Rename column with a batch size of 10.000

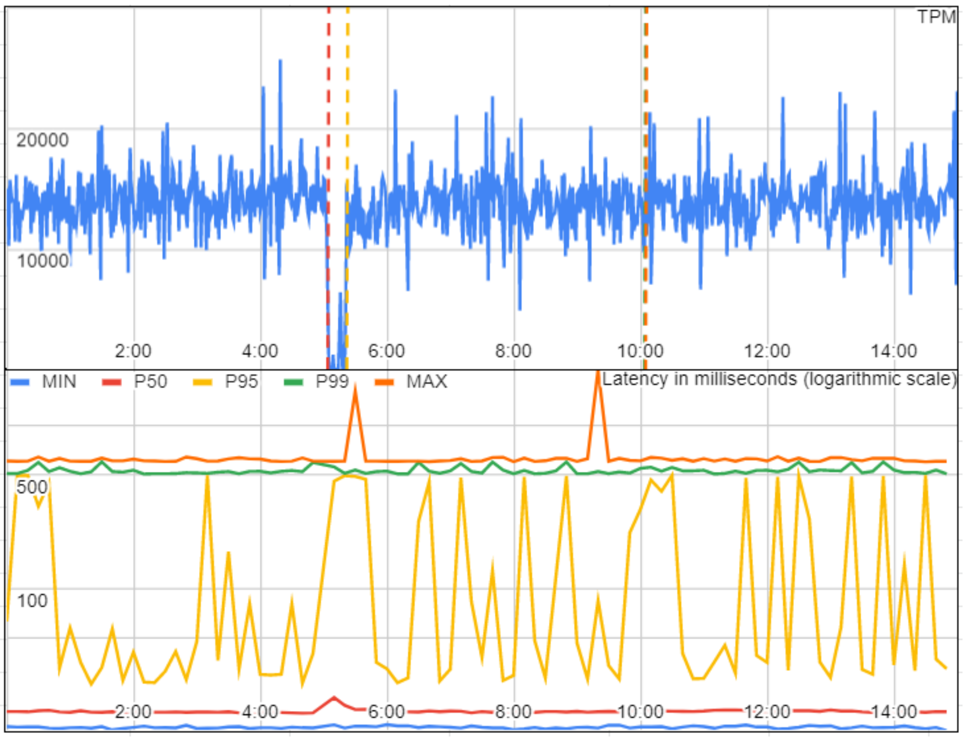


FIGURE B.1.5: Rename column with a batch size of 100.000

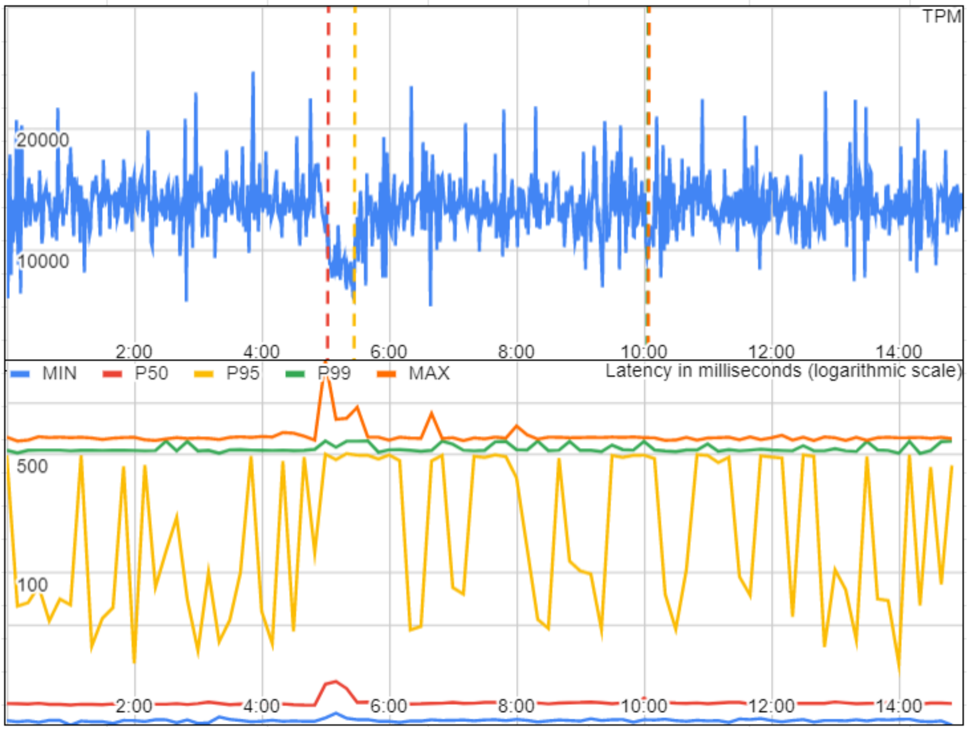


Figure B.1.6: Modify data type with a batch size of 1.000

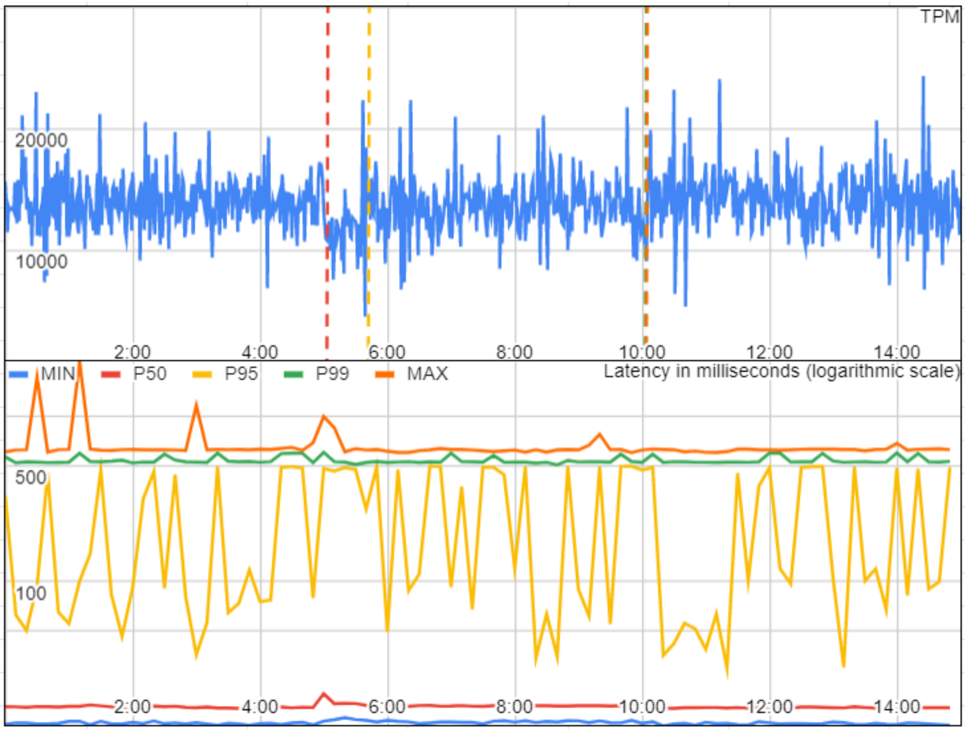


Figure B.1.7: Rename table using copy with a batch size of 1.000

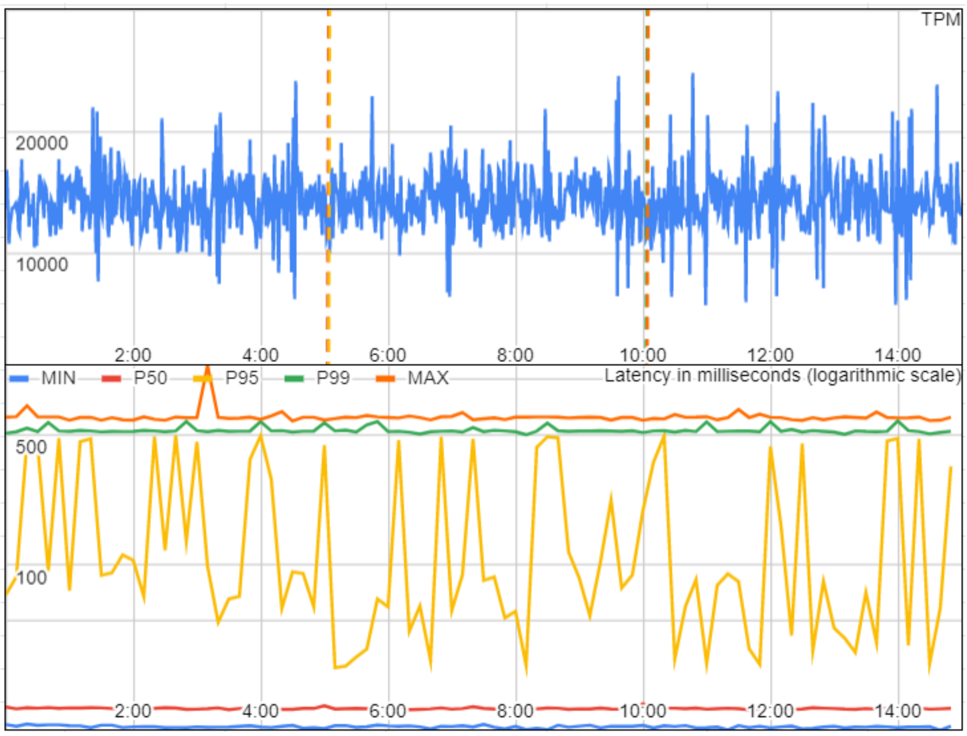


Figure B.1.8: Rename table using view with a batch size of 1.000

B.2 Synchronously replicated database

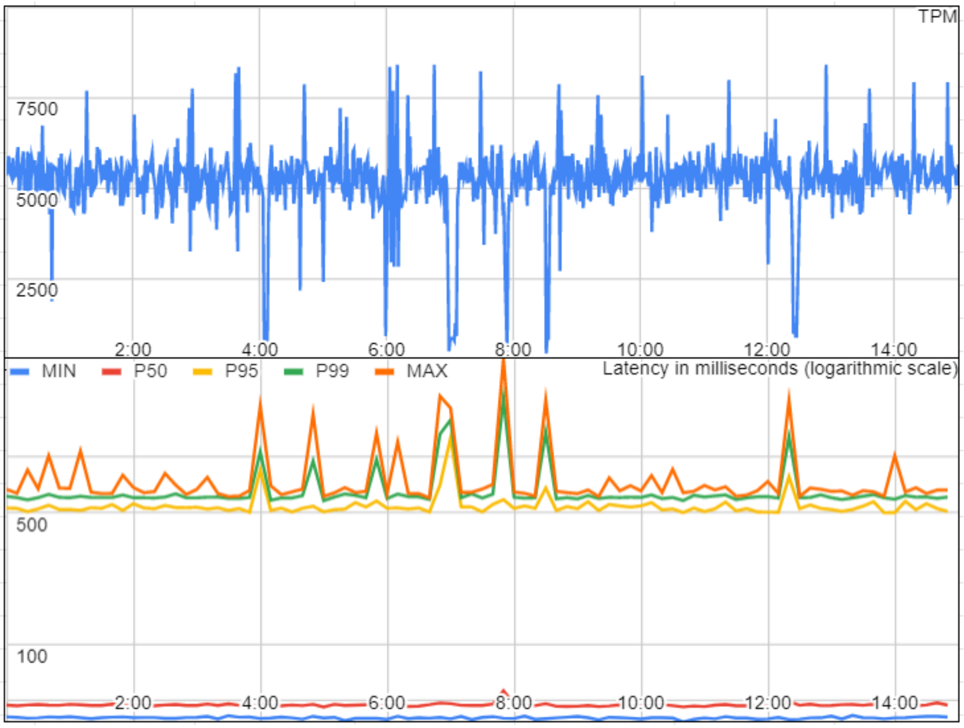


Figure B.2.1: Baseline

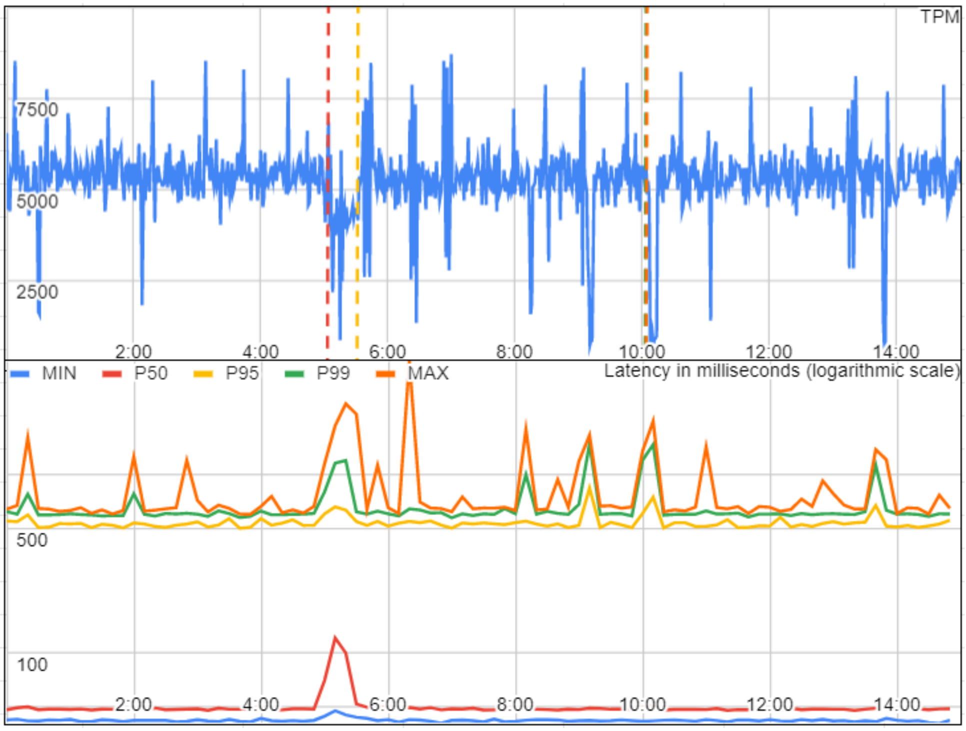


Figure B.2.2: Rename column with a batch size of 1.000

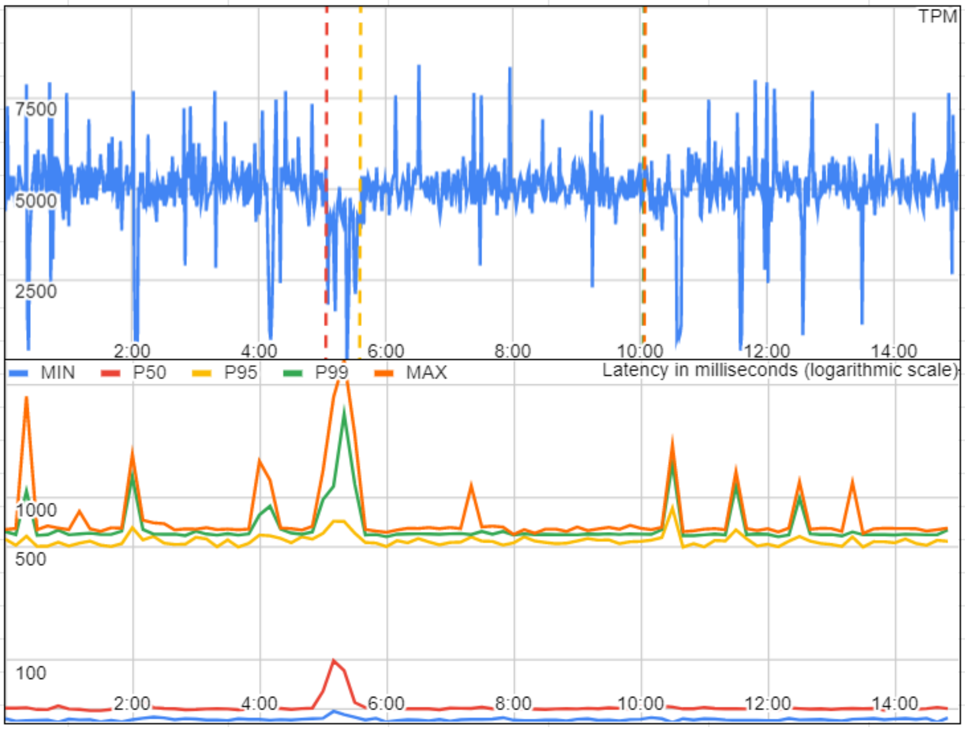


Figure B.2.3: Modify data type with a batch size of 1.000

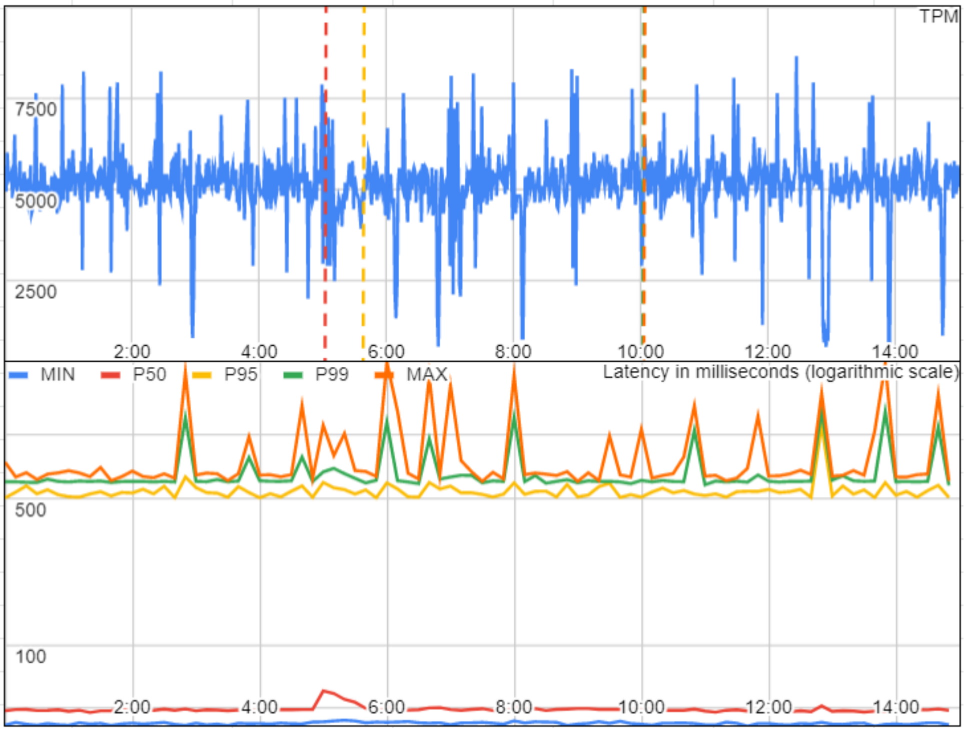


Figure B.2.4: Rename table using copy with a batch size of 1.000

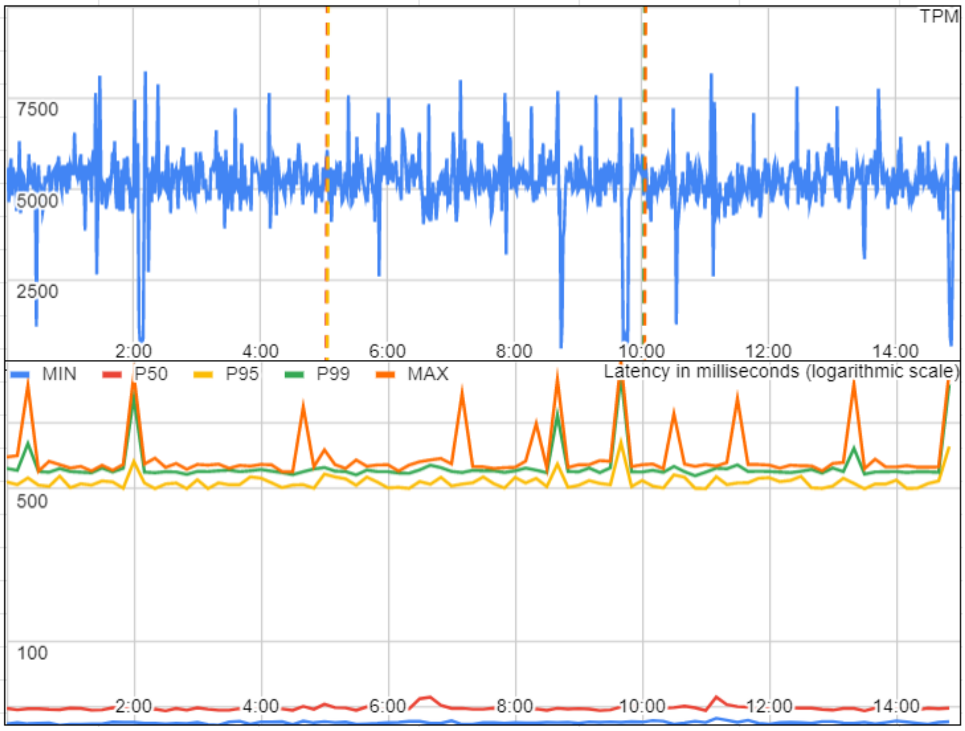


Figure B.2.5: Rename table using view with a batch size of 1.000

B.3 Contract from mixed-state

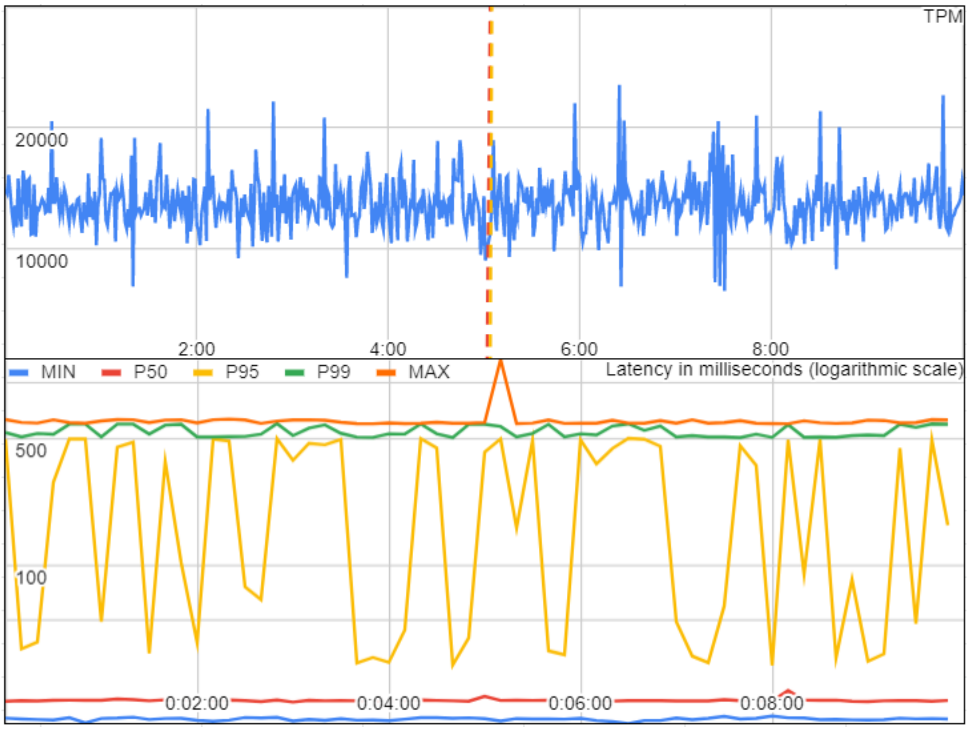


Figure B.3.1: Rename column

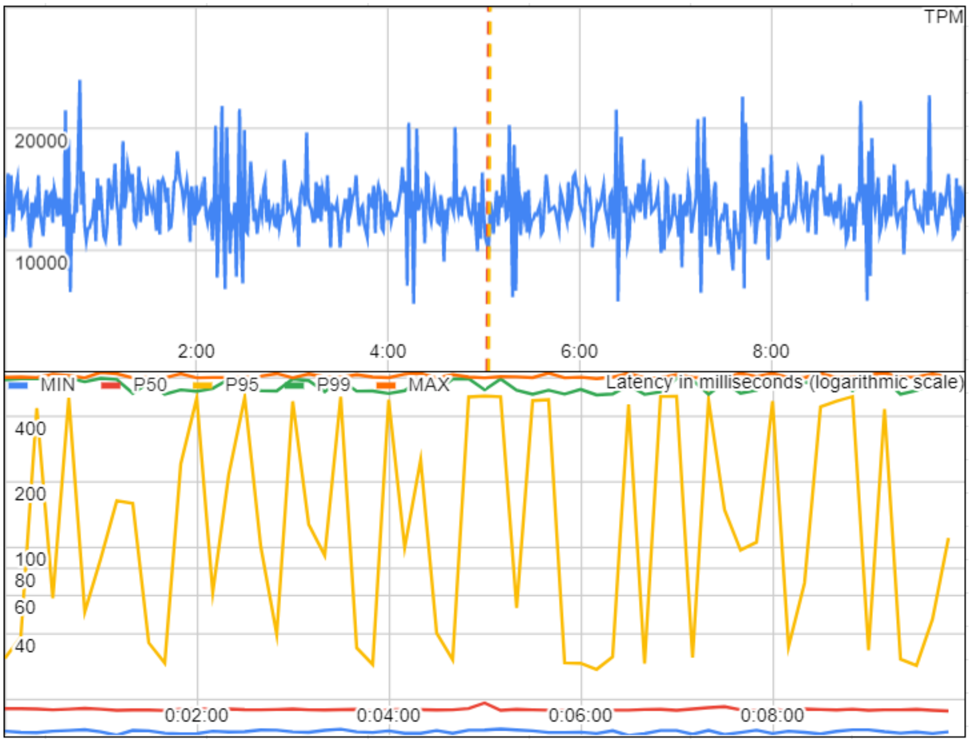


Figure B.3.2: Modify data type

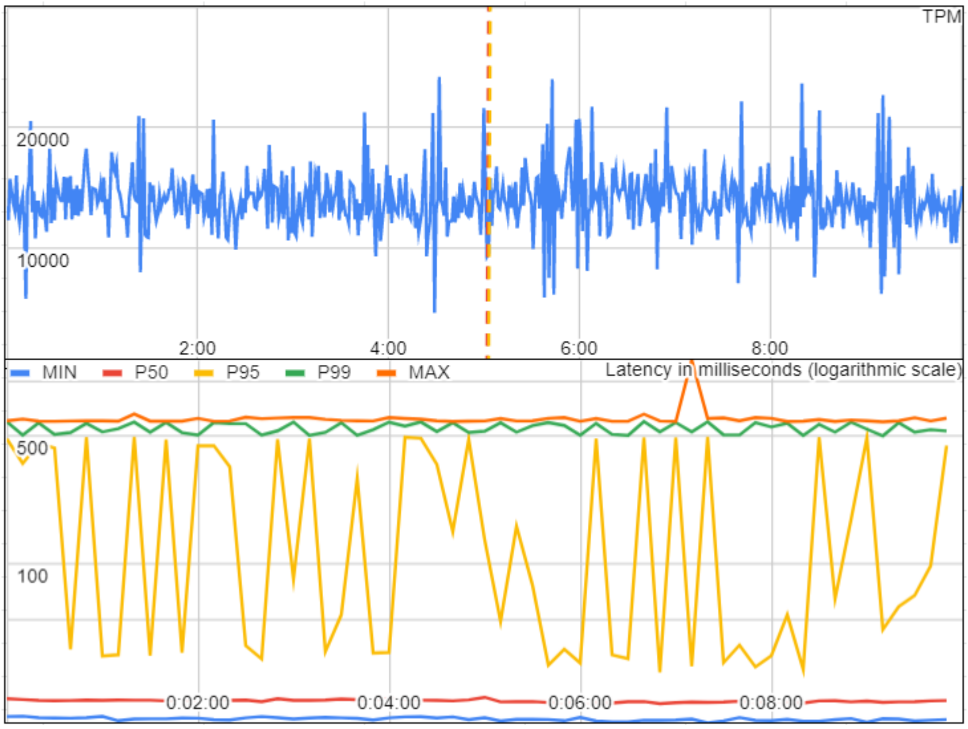


Figure B.3.3: Rename table using copy

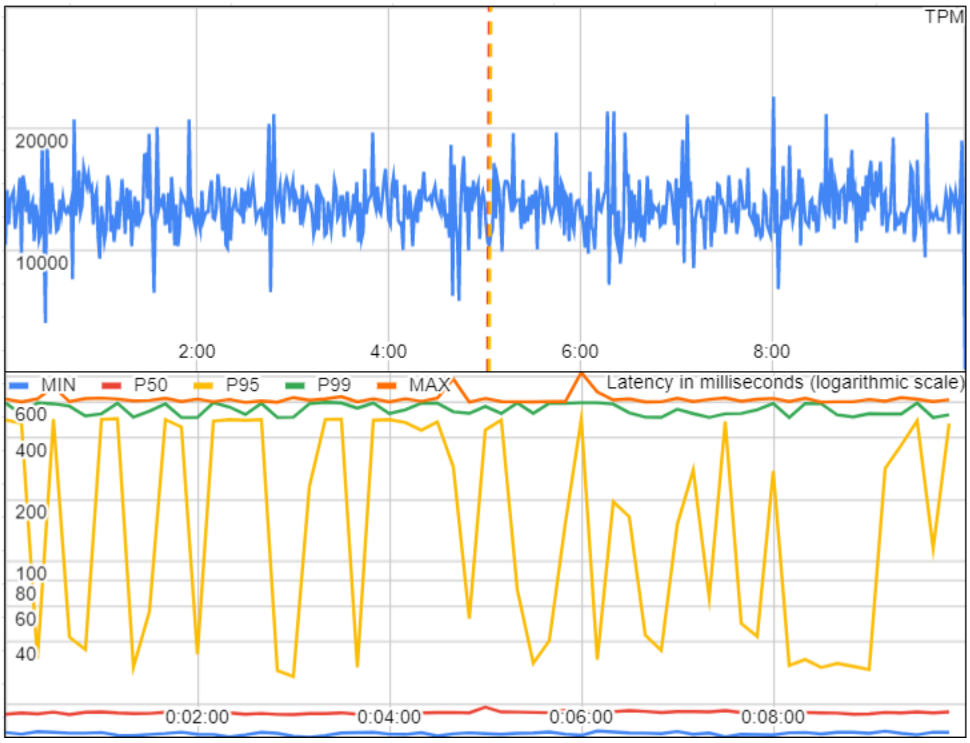


Figure B.3.4: Rename table using view

B.4 Batch migration change

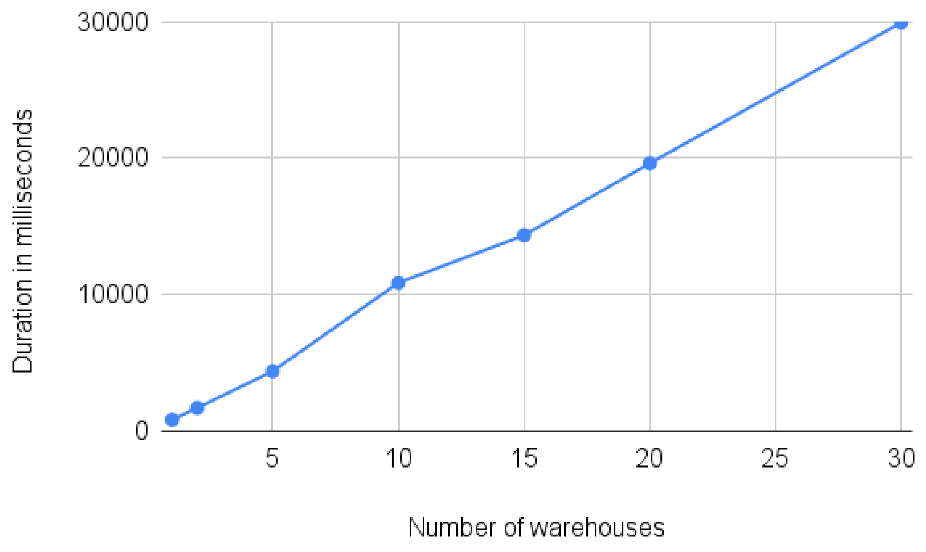


Figure B.4.1: Intra-table batch migration duration for increasing number of warehouses

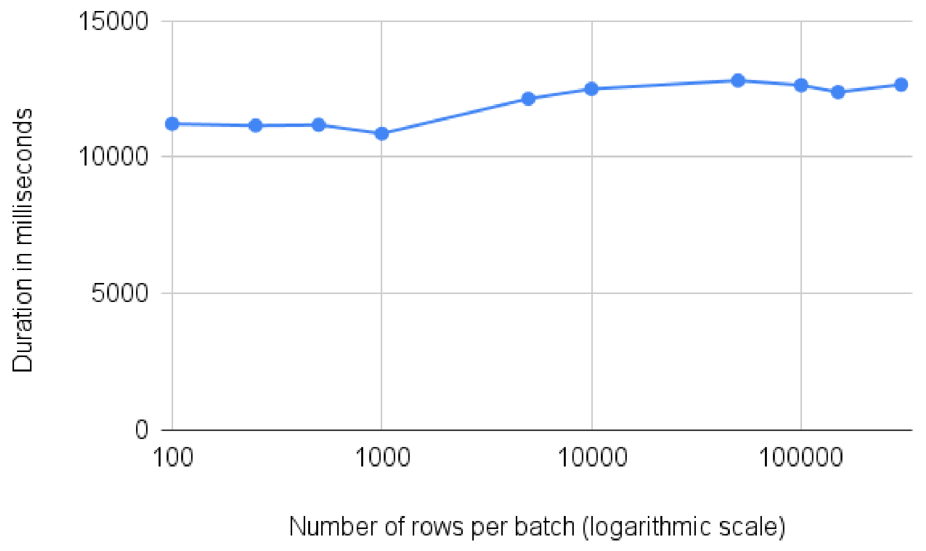


Figure B.4.2: Intra-table batch migration duration for increasing batch sizes

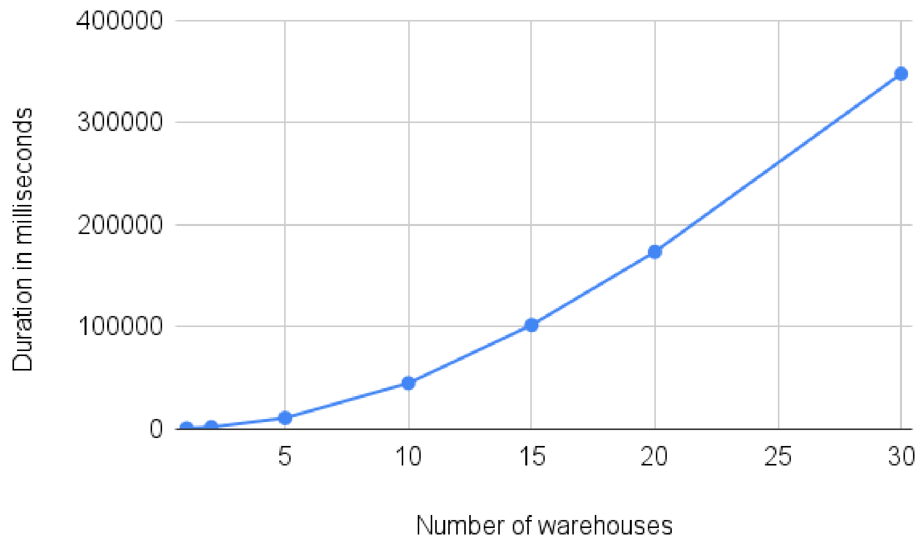


Figure B.4.3: Inter-table batch migration duration for increasing number of warehouses

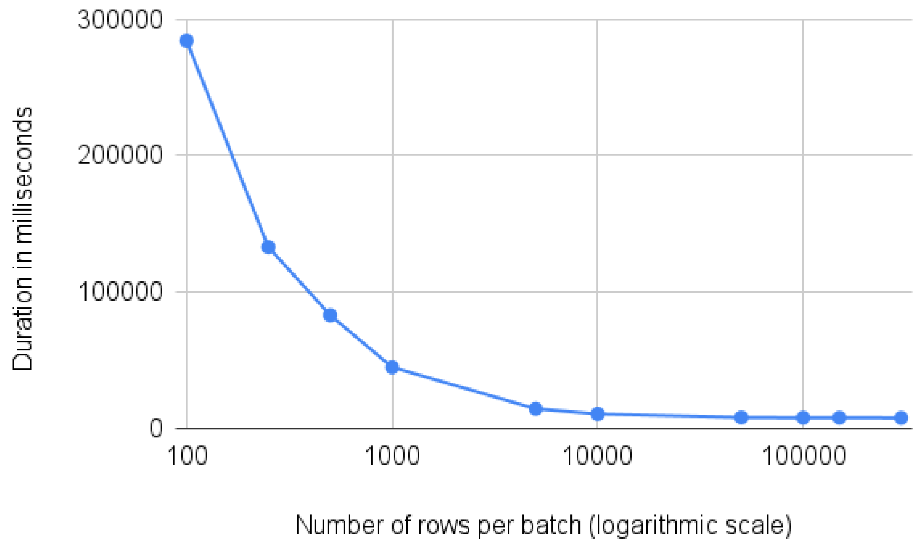


Figure B.4.4: Inter-table batch migration duration for increasing batch sizes