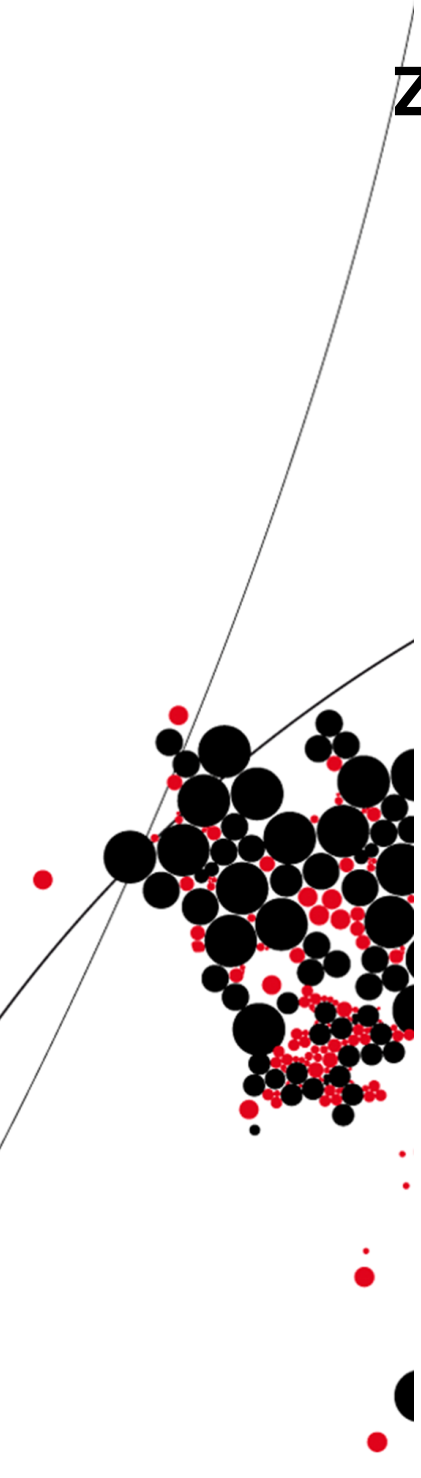
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Zero-downtime PostgreSQL database schema migrations in a continuous deployment environment at ING

Nick Geral Richter

Master of Science Thesis
Business Information Technology
Data Science & Business
October 2021



Supervisors:
dr.ir. Maurice van Keulen
dr. Abhishta Abhishta

ir. Jorryt-Jan Dijkstra (ING)

Abstract

Migrating a database schema to the next version without interrupting clients performing queries, aka downtime, is important for many reasons. A tool to perform such transformations is proposed which uses the database management system PostgreSQL, but findings are also applicable to other relational databases. The tool can be used as a step within a continuous deployment environment to provide automatic zero-downtime updates of applications when database schema changes are required.

As more services are made available around the globe by organizations, every minute or longer that the service is not reachable can cost organizations millions of euros. Most parts of an application can be updated without downtime due to redundancy measures, where multiple versions can co-exist. However, the database schema is not one of those things that can be updated with normal SQL queries while clients are still querying the database. A solution for this helps organizations increase the availability of their services and will provide clients with more satisfaction. Next to that, it enables a shorter time-to-market of bug fixes and features by allowing organizations to release multiple updates each day without availability loss.

This master thesis considers state-of-art tools from literature that promise database schema migrations without downtime. These tools are checked against a list of criteria that was retrieved from literature and furthermore ranked by experts. The best tool is chosen to be developed further and validated against various experiments. Next to that, a list of key challenges is made that currently limit the adoption of the tool(s), as well as the solutions to solve these problems.

Many existing tools are either not available online, the original developer had no intention of open-sourcing the tool, or had limited functionality in what it could do. One tool stood out from the rest, QuantumDB scored the highest on the list of criteria. QuantumDB can perform most schema transformations without blocking any existing clients from querying the database. Experiments showed that it could migrate the database schema to a new version without downtime. However, some challenges remained. Some of these challenges can be addressed with more time spend on further development while others require a different method to perform schema migrations without downtime.

To conclude, the highest-scoring tool for schema migration without downtime was selected based on criteria found in the literature. This tool is QuantumDB and during this research, it was improved further and any bugs that were found were fixed. Through validation, it is shown that QuantumDB can perform schema migration without downtime. There are still key challenges that organizations need to take into account before full implementation. To help with adopting the solution a management chapter was made that describes a business case and some use cases in which QuantumDB can be used. At last, the discussion chapter lists future work which future researchers can still perform to improve QuantumDB and give more concrete benefits for organizations to start adopting a zero-downtime schema migration solution.

Acknowledgment

At the beginning of the Research Topics part of the study program, I still had no idea what I wanted to do. I emailed a professor and she redirected me to some subjects on the open master thesis assignments page of the website. Here, one subject stood out for me which was 'data-intensive software development without downtime at ING', specifically 'using database capabilities to overcome downtime'. I already had some experience with databases but I wanted to learn more about them. The part where it mentioned ING also intrigued me as it meant I would also have the opportunity to see how it would be like in a large company.

My biggest thanks go out to Jorryt Dijkstra for the weekly meetings on Monday and Thursday where we talked about the issues I was having and how to continue. Without Jorryt I would not have started this journey but I would also not have completed it in the way it is now. Because Jorryt was already performing his PDEng thesis at the university, he already had extensive knowledge about the subject and could make me enthusiastic about doing research on this subject in the first place.

I would also like to thank my supervisors from the university, Maurice van Keulen and Abhishta Abhishta, for their advice. Many times I was trying different things and had no clear view of what I wanted but they steered me in the right direction. Establishing a meeting date was sometimes a bit difficult but always at the right times when I needed some steering.

During my research, I came across some articles by Michael de Jong. He summarized the problems in the field and had made a tool to try to solve these problems. I would like to thank him for his effort in making the software, QuantumDB, as clean and tidy as possible. I would not have been able to comprehend the flow of the software so I could make improvements. I would also like to thank him for the meetings we had and making time free to review my code submissions and for accepting the pull requests I made.

Additionally, at ING I was part of a team of both employees and intern students working on their master thesis. The standup meetings every Tuesday and Thursday helped me in following a schedule and trying to complete items before starting new ones. The two times we could physically go to the office were nice, with a great lunch. Thanks for the fun times, Luna and the others!

Lastly, I want to thank my friends who mixed things up from the mostly boring days of writing. The many Wednesday evenings in the city center will not be forgotten, although I wanted to forget the Thursday mornings as quickly as possible.

I cannot think of any others to thank here. If I have forgotten you, please do not feel like you did not contribute to this thesis in any way. I am grateful to anyone that contributed time and energy to helping me reach the end of this research. Now that the finish line has been reached, it is time to start participating in the job market!

Contents

Chapter 1

Introduction

1.1 Motivation

In today's interconnected world a lot of people are interacting with computer systems everywhere, be it at work, when browsing on social media or when checking the balance in their bank account. All these computer systems need to be maintained to ensure correct functioning. No application is ever complete, without errors and missing features. Quick resolution of these faults and implementation of missing features provide many customers with additional value. When that happens customers are willing to pay more or use the application longer which improves the profitability of the business.

Whenever a new application version is released for these systems, management wants every user to update to this latest version. Some systems, however, cannot be shut down to perform such an update without (financial) consequences. A lot of research has already been done on how to perform live or online updates to the software of these computer systems, but not a lot of research has been done to also update the underlying database schema on which these applications depend without downtime. Companies use different strategies to deal with this issue, they either deter updates to the database schema until it is really needed and then perform a large number of updates at the same time, have regular downtime windows in which a minimal amount of users use the system and accept the consequences of this downtime or adopt the Expand-Contract pattern as is discussed in [1]. None of these solutions are ideal as it puts a brake on the quick and flexible software development approach in continuous deployment.

To provide a much-used example, Wikimedia Foundation, known for Wikipedia and other sites, is continuously updating its wiki engine called MediaWiki. According to Curino et al. [2], in 2007 MediaWiki had seen 171 database schema versions. As of writing this thesis, this number is much larger. These updates provide additional features as well as fix mistakes made in previous versions. However, bringing Wikipedia and other sites down for maintenance would be disastrous for people or organizations that depend on it. Providing a temporary snapshot of the site would also not work as any changes committed during maintenance would not be in the database at the end of the update procedure.

While some research has been done on online updating of database schemas, no article or paper was found that also looked at the implementation of such tools/strategies/frameworks in a corporation. There should be various reasons why a solution is not widespread. Solving these problems can lead to an increase in business value when implemented. In short, decreasing the feedback loop between end-users and developers can increase customer retention and customer satisfaction.

This research is conducted at ING, the largest bank in the Netherlands. ING is not necessarily a traditional bank, they are investing heavily in providing the best digital experience possible, requiring a huge IT development landscape. Using ING and its employees to learn about potential key challenges and to see how a solution for zero-downtime schema migration would be implemented is therefore useful.

1.2 Research questions

The goal of this research is to design a tool that will provide zero-downtime schema migrations. For a tool to be designed, criteria and requirements need to be known. Before a new tool is build, existing tools have to be researched to see if they might already fit most, if not all, of the requirements. This would free up time and resources to progress further into the design cycle where limited time is available. Potential key challenges and limitations with the best scoring tool can be researched after which a decision can be made to develop a tool from scratch or continue development with the chosen tool to fix potential key challenges and bugs and to potentially have the tool comply with more criteria. The tool will have to be validated to ensure it works and performs as expected. This will raise confidence with executives at the decision-making level to start incorporating the tool into the development pipeline.

Since this research is not primarily focused on answering research questions but on designing a tool, the research questions that will be asked have to support the creation of such a tool. Some of these questions will be answered in Chapter 2 & 3 while others will be answered in later chapters.

The following research questions have been formulated to support the creation of a zero-downtime schema migration tool in this master thesis:

1. What are the criteria for a zero-downtime schema migration tool?
    1.1. Which criteria can be found in literature?
    1.2. Can a ranking be made between criteria?
2. Which tools are already available that aim to provide zero-downtime schema migration?
    2.1. Can a best tool be identified based on the criteria gathered?
3. What key challenges can be found?
    3.1. What technical challenges can be found that will hinder the creation of a tool?
    3.2. What challenges can be found that prevent implementation of the tool within an organization?
4. For which use cases can the tool be used?
    4.1. Are there specific use cases that the tool would be great for?
    4.2. What would need to change for the tool to become better and support more use cases in the future?

1.3 Methodology

This section discusses the methodology that is used during this master thesis. Since the problem in question is a design problem, i.e. a solution that has to be found for a problem, the Design Science Methodology by Roel J. Wieringa [3] is used. The methodology consists of a couple of stages as described in the book. This book provides guidelines to design information systems (and software engineering research) in a certain context and how to investigate the performance of the artifact in the specified context. The methodology is split into four stages, namely the following: implementation evaluation/problem investigation, treatment design, treatment validation, & treatment implementation. These are the steps for the engineering cycle. The last step was not performed due to time constraints limiting what could be done.

The first part of the methodology was performed during the pre-research stage called Research Topics. During this stage Chapters 2 & 3 were written to provide a complete picture of the current field

of research and already answer some research questions. During the master thesis part, the treatment design and validation have been performed and the remaining chapters have been written.

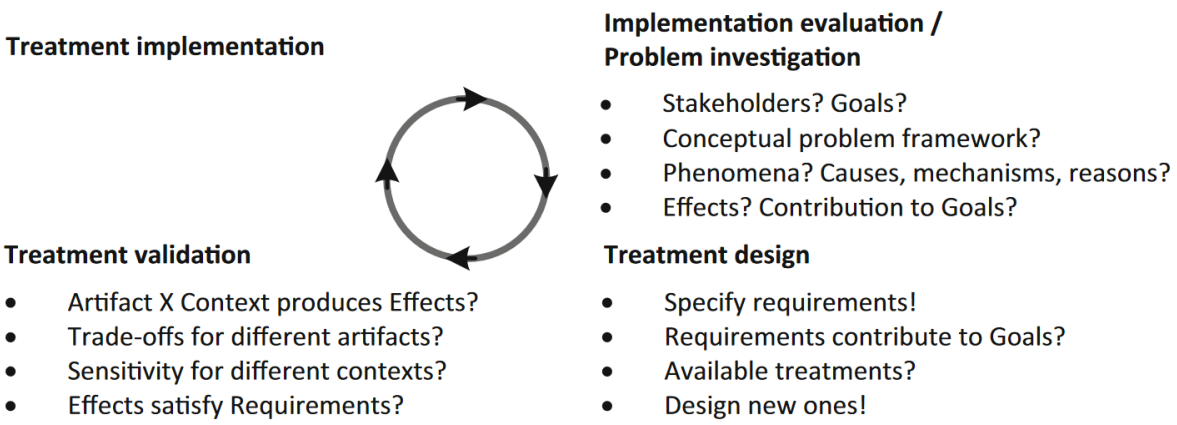


Figure 1.1: Engineering cycle as found in the book [3]

1.4 Report organization

This report is divided into several chapters of which this introduction chapter is the first one. After the introduction, some background into the field of databases and schema migration will be provided to refresh knowledge or give the reader a solid understanding of the concepts and nomenclature used in the field. To get a deeper understanding of the problem, a literature review of zero-downtime schema migration has been conducted in Chapter 3. After that chapter, a short investigation into the problem is given, after which the design of the tool is done. Next, the tool is validated to work in various scenarios using experiments. In Chapter 7, a discussion is held on the interpretation and implications of the data gathered, and limitations of the research and tool are given. Next, a business case and use case analysis is given for managers that want to start using a tool to perform zero-downtime schema migrations. A list of benefits is also given to provide more incentives to start the adoption. Finally, a conclusion where a summary is given, the contributions of this research are listed and in short, the answers to the research question are briefly summarized.

Chapter 2

Background

The background chapter will be used to fill in some knowledge about the subject of zero downtime database schema migrations. A lot of subjects are connected to the research questions and with this chapter, the aim is to supply a clear context in which the problem is positioned.

2.1 Databases

A database is a place that stores data in an ordered manner, typically in digital form on a computer system. Depending on what kind of database model has been used it has various rules about adding, modifying, retrieving, and deleting data from this place. Over the years many different database models have been designed, these models indicate how data has to be stored in which format and in what way the data can be requested. The first database model was called the hierarchical database model and was developed by IBM in the 1960s[1]. It uses a tree structure to link data to each other.

The hierarchical database model was innovative as it allowed companies to store, organize, link, and retrieve information but was inflexible in how it works. In 1969, Edgar F. Codd proposed a different database model that could solve this inflexibility. This model was called the relational database model [4]. The way this model was implemented is using tables where rows of data have specific columns, a tabular design. A row of data is also called an object as every object contains the same type of information. Certain columns can reference objects in other tables to create a link between each other. This way, many relationship forms can be achieved, whereas the hierarchical model could only achieve a one-to-many relationship down the tree structure.

2.2 Relational database schema

To specify what kind of data can be stored and retrieved by the relational database, a so-called schema is needed. This schema allows for consistency between stored pieces of data as it specifies what can and cannot be in a table with certain columns. A concrete example can be seen in Figure 2.1. This figure is made using a modeling language to give an overview of all the tables and columns and which tables are linked by which columns. This figure consists of 8 tables with the number of columns ranging from 5 to 20. The first column in the figure specifies the names of the column in the table. The second column specifies which type of data can be stored in this column, you can see the type int (integer), str (string), DateTime, and more. The third column specifies the constraints that are placed on the column. Primary keys are the columns that are uniquely identifying for that table and are used by other tables to

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

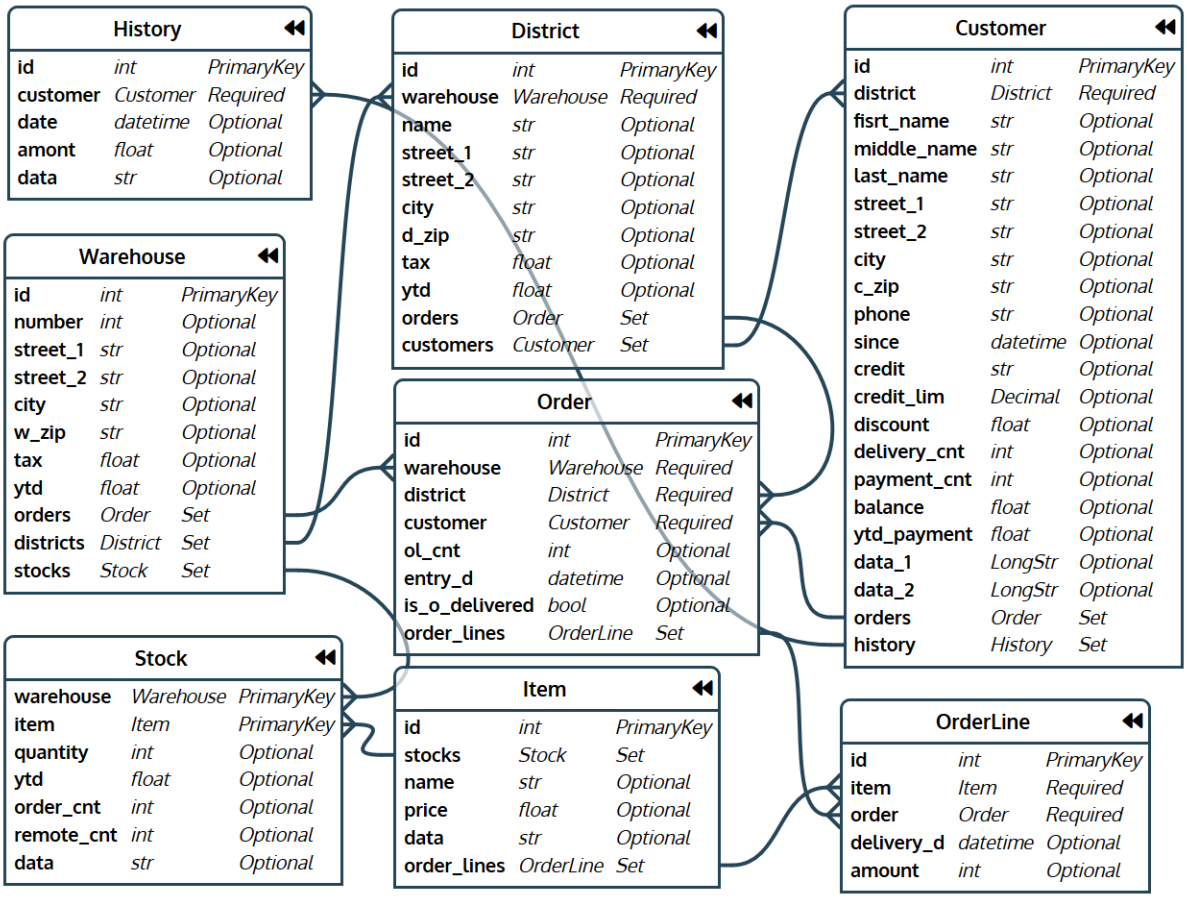


Figure 2.1: TPC-C benchmark schema
(found at: `https://editor.ponyorm.com/user/dominovtut/TPCC/designer`)

refer to a specific row of data. The 'id' column of the District table is such a primary key that is used by 2 other tables, namely the Customer table and Order table. A customer can only be associated with 1 district and an order can also only be associated with 1 district. To get all customers of a certain district, a query would have to be made searching for all customers where 'district' equals a certain id. This way no data would have to be stored multiple times, therefore reducing storage space.

Columns can also be optional, if they do not contain any data there will be no problem whereas if they are required and a new row of data does not have this information the database would not let you create this row of data. This can become a problem if at first, a table contains columns that are optional whereas later a database developer wants it to become required. To solve this, the database developer has to specify a default value for every row that did not contain this information.

In this figure, a set just means that another table is referring to this table, in a working database this column does not exist, only as a Foreign Key constraint. This Foreign Key constraint tells the database that only existing objects from that table can be referenced here. If that object does not exist, no row can be made. The same holds for deleting a row of data when another table references the to-be deleted row. The database does not allow it without also deleting the referencing row. An example in Figure 2.1 would be if a District is to be deleted, it is only allowed if there are no customers and orders associated with this district. This makes designing a good database difficult as it is not easy to implement some changes to this schema at a later date when the database is in production.

2.3 Other database models

In recent years more database models have been created and are in use by big corporations as well as small developers. Some of these database models were already used in the early days but are rising in popularity once more. Popular alternatives to the relational database are the so-called NoSQL databases (Not Only SQL or non-SQL) [5]. These databases do not work according to the tabular design but follow a more loose definition where fewer constraints are enforced. This helps corporations deal with the quickly changing landscape of their applications and their big data. These databases mostly allow for better horizontal scaling, also called scaling out. This scaling out means that you can just add clusters of computing power and everything works without concurrency problems. A relational database can only be scaled vertically, or scaled up. This mostly means substituting the current hardware with more powerful components. Jatana et al. [5] identified 10 popular database models of which the top 4 are discussed below. Mark Drake [6] also identified these 4 as the most popular NoSQL variants.

2.3.1 Key-value stores

A popular NoSQL database is the key-value store, this database provides a lot more flexibility to developers than a relational database. Data is stored as a value of any type in combination with a key, this key is unique in the whole database. The database does not care what type the value is, it may be an integer, string, or even a series of bytes. The database does not restrict developers from using keys in different formats, e.g. 'person.john.doe' vs '1234567890', or decide which type of value has to be used. There are however a couple of disadvantages associated with it. One of them is that there is no overarching architecture that can be used to provide guarantees to applications. Another disadvantage is that certain query features are not supported by the database and have to be implemented at the application level, these include the 'join' and 'group by' features. For organizations that want to build robust applications, key-value stores and some other NoSQL databases do not provide the features and guarantees that relational databases do. This is one of the reasons why this research is solely focused on relational databases.

2.3.2 Graph databases

In graph databases, information is stored as nodes and edges. Nodes are the entities or instances that represent an object and can be thought of as a row in a relational database. An edge represents the relationship between only two nodes. This representation can be directed or undirected depending on the meaning of the edge. A node can have more than one edge with another node, for example, the edge 'parent' and another edge 'family' can exist between two-person nodes. A node can also have multiple connected edges of the same type, in this case, the 'family' edge can be connected to every node that has the family relation. Graph databases are relational and have been created to address some limitations of the above-mentioned relation database model. They allow for less strict schema designs that in turn are more suitable for quickly changing data.

2.3.3 Document stores

A document store stores data in the form of 'documents', which can be seen as semi-structured. These documents can include any piece of data but have to be encapsulated in a standard format like XML or JSON depending on the implementation. To request these documents a unique key is needed like a path or URI. An advantage of this design is that data can be grouped inside such a document while it may be scattered throughout a database when a relational database is used. Document stores are

basically a subset of the key-value stores where the system actually cares in which format the value is stored.

2.3.4 Columnar databases (Column-oriented)

Relational databases can be designed to store data in two ways. The normal relational database stores data per row while some workloads perform better when data is stored per column. Requesting, adding, updating, or deleting a single row is much faster when data is stored per row but when a query requests the data where a certain column has a value between 1.000 and 10.000 the database has to go through every entry to return a list (if no indexes are used). A database that stores the data per column can quickly look up which rows satisfy the query and return this information.

2.4 Database Management System (DBMS)

A database management system is a software that facilitates the addition, modification, deletion, and retrieval of data in the database as well as the modification of the underlying database schema. A lot more than just facilitating those functions is happening behind the scenes and users need to know what guarantees they can expect from the system. In the sections below the most important ones are highlighted.

2.4.1 Structured Query Language (SQL)

As the name suggests, SQL is the language that was developed to let database developers construct the database and let applications interact with the data in the database. It mainly consists of 4 categories of statements, data query language (DQL), data definition language (DDL), data control language (DCL), and data manipulation language (DML).

DQL & DML

The first language, DQL, is only used to select data from the database. This is one of the most used statements next to the statements belonging to the DML. The DML contains the INSERT, UPDATE, and DELETE statements that manipulate the data in the database. Sometimes the SELECT statement is argued to be part of the DML when combined with some clauses such as WHERE and FROM that do manipulate the data in some way before returning the SELECTed data but it is strictly speaking still part of the DQL.

DCL

DCL is the odd one out of the 4, various DMBS implement these statements according to their own specifications. These statements are GRANT, REVOKE and DENY and are used to give users permissions for various statements on parts of the database. Oracle lets you use the statements of DCL in the DDL and MySQL calls it Database Administration Statements. PostgreSQL allows for transactional statements which allow for rollbacks, while SQLite does not implement any DCL statements and relies on file system permissions to handle who may modify the database.

DDL

The last language is the DDL, these statements let you alter the shape of the database schema. The statements include CREATE, ALTER, DROP and TRUNCATE and are used whenever there is a new need for how data is to be stored. These statements bring the database from one schema version to the next one and define how DQL and DML can select and manipulate the data. The focus of this research is mainly on DDL as these statements mostly block access to the database objects that it alters. However, DQL and DML are important too as these statements are the ones that cannot be blocked when the database is online.

There is no standard implementation of these statements and every DBMS has to implement them individually. Some DBMS decided to not implement some of these statements or alter the way they work. PostgreSQL has one of the best compliance to the specification[2] while other DBMS lack in quite some areas.

2.4.2 Transaction properties (ACID)

The ACID (Atomicity, Consistency, Isolation, Durability) properties are quite important for relational DBMS to provide guarantees. The first property, atomicity, enforces that all operations in a transaction either occur all at once or not at all. A characteristic of this property is that a transaction may not be able to see another transaction being partially executed. This means that at one point in time the transaction did not happen while at the next point in time it instantly happened. It can also happen that one of the operations in the transaction failed due to some error, in this case, all already performed operations have to be reversed. This is called a rollback. An example of why one would want atomicity is the transfer of balance from one bank account to another bank account. You either want the transfer to be correct or not happen at all. If only the operation completed where money was deducted and the operation where money was added to another account failed, an amount of money disappears which is not how it should work.

The second property called consistency calls for a consistent database in regards to the rules being enforced. A database schema might consist of rules that are not to be broken. If a column is set up to be an integer, the database has to ensure nothing else than an integer is being saved at that place. If a transaction wants to insert a piece of text the transaction gets aborted. The DBMS does not concern itself with rules that are set in the application such as a minimum or maximum value when it was not specifically declared when the column was made.

The third property is isolation and comes in different levels of strictness. Isolation determines how much other transactions can still see from a transaction that has not yet been committed. Ideally, this isolation should be at the highest level called serializable. At this level, no so-called read phenomena can occur. These phenomena are dirty reads, non-repeatable reads, and phantom reads, also no lost updates can occur at this level. However, this level of isolation comes at a price, namely less concurrency. If the application for which the database is being used is less strict, a database administrator might decide to lower the level of isolation to improve performance.

The last property of the 4 in ACID is durability, this means that any changes to the database need to be permanent whenever they are committed. In most cases, this just means that the data has been written to non-volatile storage like a hard drive or solid-state drive. Random-access memory (RAM) is an example of volatile memory and will lose all of its data whenever a system crashes and is therefore not durable to store for the long term.

Of these properties three of them are present or they are not. For the isolation property, there are different sub-levels and a database administrator might decide to lower the level to increase performance

[2] https://www.wikiwand.com/en/SQL_compliance

in exchange for some lesser guarantees. Most critical database systems want to use the highest level while hobby projects can use a lower level because a dirty read might not cause an issue.

2.4.3 CAP theorem & BASE

The CAP theorem by Eric Brewer [7] states that only two of the three following properties can be enforced at the same time: Consistency, Availability & Partition tolerance. What this basically means is that distributed systems, like NoSQL systems, cannot provide either consistency or availability at the same time whenever there is a network failure. The ACID systems described above choose consistency over availability while distributed systems almost always choose availability. For NoSQL systems this is called BASE, Basically Available, Soft state & Eventually consistent. It is not necessarily the counterpart of ACID because some systems still have some properties of ACID. The BASE properties state that the system is always available, meaning online and able to do work. The soft state property tells us that data can still change without input, as becomes clear from the last property. Eventual consistency makes sure that without additional input the system becomes consistent, however, whenever data gets written a short time exists before this data is redistributed to other partitions of the system. During this time the database is not consistent.

2.4.4 Concurrency control

Databases differ from other storage architectures, such as spreadsheets and the file system, in the way that multiple people can concurrently access and operate on pieces of information while not violating the integrity of the data. To achieve this concurrency, various control mechanisms have been used and proposed and can be important in determining if some transactions are blocking or non-blocking.

Concurrency methods can be divided into three categories. These are optimistic, pessimistic, and semi-optimistic. Pessimistic approaches always block operations if the possibility exists that a violation of the rules can occur. Optimistic approaches do not block operations but only check at the end if all rules still hold. Semi-optimistic uses a combination of both. All these approaches have different advantages in certain cases. If the assumption is made that almost no conflict will exist between operations, optimistic methods will perform better, while pessimistic methods will perform better if conflicting operations occur all the time.

Locking

Traditionally, databases used a locking protocol to guarantee concurrency. With this method, certain objects in the database could be locked. A transaction could be changing a table and requires that no other transaction is operating on the table in some way, this transaction would request to acquire a table lock for that table. Any subsequent transactions would be blocked until the lock has been lifted. Not providing table locks would not prove that useful as most transactions only want to read and write one row. To provide more granularity, locks can be requested in the following levels: database, table, page, row, and field. The field level lock is rarely implemented.

Next to this granularity, there are multiple types of locks, the most simple one being the binary lock. This lock just locks and unlocks the resource. A more sophisticated lock is the shared/exclusive lock. This lock, depending on what it wants to do, will either choose to acquire a shared lock if it wants to only read a resource or acquire an exclusive lock if it wants to modify the resource. With the shared lock other transactions that want to acquire a shared lock can also do that because they only read the resource. An exclusive lock is, as the name says, exclusive and no other transaction can acquire either the shared lock or another exclusive lock until the lock has been released.

A popular locking control method used is called two-phase locking and uses two phases of acquiring and releasing locks that are followed after one another to ensure no concurrent transaction can view or alter data that is about to be altered by a transaction. A transaction only starts if it acquired every lock and not sooner. This does not resolve deadlocks on its own, two transactions can still request locks that are already acquired by the other.

Timestamps

Another method to provide concurrency is the use of timestamps. This timestamp dictates the order of transactions and must be unique and monotone, meaning only increasing. Whenever a transaction wants to start it gets a timestamp, if this timestamp is newer than a transaction that is still running it will either wait until the older transaction is done or 'die' and restart.

Multiversion concurrency control (MVCC) is such a method that uses timestamps and is the dominant concurrency control method used nowadays[3]. Together with timestamps it also uses the multi-version approach to not overwrite existing data but create a new version. This way, the method can provide point-in-time consistency by serving a snapshot of the data. Read transactions see different versions of data depending on which isolation level is set. Higher isolation levels ensuring that newer read transactions see the version of data that corresponds to their timestamp.

Commitment ordering

Next to the two mentioned above, a couple of other models exist. One of them is commitment ordering (CO) and belongs to the optimistic approach. This approach assumes that the majority of transactions do not conflict and will not acquire locks or timestamps. This method will control or check if the chronological commit order is compatible with a so-called precedence graph, i.e. that no conflict is happening by committing the transaction.

Global serializability

When working with distributed systems different approaches have to be found to make sure the serializability of the whole distributed database is enforced. Distributed versions of the above-mentioned methods do not scale too well except for CO[4].

2.4.5 Procedural language (PL/SQL)

Sometimes an application might want to request some data, do some operations on it and request additional information based on the first piece of data. This will cause a lot of overhead sending the data between the database and application. To solve this problem, DBMS has implemented a procedural language into their system called PL/SQL. Procedural languages are also available in various other languages such as Python, Perl and TCL, and many more with plugins which is the case with PostgreSQL[5].

Triggers

As part of the procedural language, there is support for triggers. Triggers are functions that execute when an event happens. These events are 'insert', 'update', 'delete', or 'truncate' queries that arrive at the DBMS. By using triggers, databases can be further customized with specific functions such as also

[3] https://en.wikipedia.org/wiki/List_of_databases_using_MVCC
[4] https://en.wikipedia.org/wiki/Commitment_ordering#Distributed_serializability_and_CO
[5] https://www.postgresql.org/docs/current/xplang.html

updating another table or deleting some row based on an incoming insert. Triggers can be fired before or after the query happens but also instead of the actual query if they are defined on a view.

2.4.6 Database vendors

Many DBMS currently exist that are widely used for various purposes. There are some reasons to choose one DBMS over the other. In the following section, a couple of widely used and popular DBMS are listed together with their advantages and disadvantages. Two websites have been used to estimate popularity and widespread usage. Stack Overflow, a website where people can ask questions about mostly programming-related subjects, performed a questionaire[6] to see which DBMS were used. The second website, db-engines.com[7], ranks all DBMS based on a formula that includes mentions on popular websites, job listings, and people with the DMBS on their profile among other things. A difference can be noticed between the two rankings which can be attributed to official support answering most questions for enterprise editions while small developers that use free open-source systems have to rely on Stackoverflow for support. The 5 most popular relational DBMS: MySQL, PostgreSQL, Oracle DB, Microsoft SQL Server & IBM Db2 will be listed below. An overview of the various features can be found in Table 2.1

MySQL

MySQL is the most used database management system by developers, according to the survey by Stack Overflow. It is not a coincidence as it has a lot of advantages over other DBMS. MySQL is used in many popular web applications including Facebook, Twitter, WordPress, and Joomla[8]. It is commonly used in combination with other popular open-source programs such as Linux, Apache, and PHP. MySQL was acquired and is being developed by Oracle and is distributed in two versions, an open-source edition, and an enterprise edition.

PostgreSQL

PostgreSQL, or for short Postgres, is an open-source object-relational database management system (ORDBMS). It supports most of the latest technologies and techniques and is widely used by developers. According to Stack Overflow, it takes second place in most used DBMS while on db-engines.com it is listed on the fourth spot. PostgreSQL is not a product of a large corporation that wants to make a profit. It is being developed by the PostgreSQL Global Development Group and is thus more of a project of people enthusiastic about building a DBMS. PostgreSQL is not distributed as several editions which makes it ideal for troubleshooting problems as everybody uses the same system. There are many plugins or third-party apps that can work with PostgreSQL because of its open-source policy.

Oracle DB

A competitor to PostgreSQL is Oracle DB, Oracle has been developing its database management system since 1979. Oracle DB does not only support one database model, it supports several models next to the standard relational model, including many NoSQL models such as key-value stores, graph databases, and triple stores. It is also highly customizable with many features that can be included. It takes place 8 on the popularity ranking of Stack Overflow but it ranks first on db-engines.com and is used by many

[6] https://insights.stackoverflow.com/survey/2020#technology-databases-all-respondents4
[7] https://db-engines.com/en/ranking/relational+dbms
[8] https://www.mysql.com/customers/

Features	MySQL	PostgreSQL	Oracle DB	Microsoft SQL Server	IBM Db2
Open Source	✓	✓	x	x	x
Free	✓	✓	x	x	x
Large community	✓	✓	✓(depends)	✓	x
Provides NoSQL capabilities	✓	✓	✓	✓	✓
Official support	✓	x	✓	✓	✓
ACID compliant	✓	✓	✓	✓	✓
SQL compliant	x	✓	Mostly	Unknown (T-SQL)	Unknown
(Third party) Extensibility	Somewhat	✓	x	Unknown	x
Migration to another RDBMS	✓	✓	x	x	x
Easy setup	✓	x	x (licensing)	✓	x
Advanced security features	x	x	✓	Unknown	Unknown
Master-master replication	x	✓(third party)	✓	✓	Unknown
High availability	✓	✓	✓	✓	✓

Table 2.1: Features in the top 5 RDBMS

large corporations because of the enterprise features it offers. ING uses Oracle DB for some of its applications.

Microsoft SQL server

Microsoft SQL Server is another widely used DBMS in the enterprise world. It also offers a lot of features just like the ones above. There is a restricted free version available for use by small developers. Its main strength comes from Microsoft having a large customer base in other products such as Windows Server and Microsoft Azure.

IBM Db2

IBM started as one of the first vendors building DBMS and was widely used by corporations. However, the popularity of IBM has been dropping, and currently, not a lot of new projects that are started include IBM. Stack Overflow shows that only 2.9% of developers use IBM Db2, way below the other DBMS. db-engine.com still lists it on the fifth spot but it would not be weird if those are mostly legacy systems that still have to be supported. It is still included in this list as one of the top enterprise vendors next to Oracle and Microsoft.

Advantages & disadvantages

All these relational database management systems are made with different ideals and thus contain many advantages and disadvantages over each other. In Table 2.1 some of the most used arguments are listed on why a developer or organization might choose one over the other.

Why PostgreSQL for this research

PostgreSQL has been chosen as the best RDBMS for this research because it is open-source, free, and does not have restrictive licensing when using or modifying its source code. Another reason that PostgreSQL has been chosen is the large community that supports this RDBMS and the third party extensibility, but also its large compliance to the SQL specification. These features make it a great candidate to be used in a company without any dedication to paying a monthly or yearly fee for usage.

Database Version Control Tool	Database independent[9]	Automatic rollback	Transactional	Composability	Free	Open source
Visual studio SQL server data tools	No	No	No	No	Yes	No
Version SQL	No	No	No	No	Yes	No
Liquibase	Yes	Yes	Yes	Yes	Yes	Yes
FluentMigrator	Yes	Yes	Unknown	Yes	Yes	Yes
Sqitch	Yes	No	Yes	Yes	Yes	Yes
Apricot DB	Yes	No	Unknown	No	Yes	Yes
Flyway	Yes	No	Yes	Yes	Yes	Yes
DBGeni	No	No	Yes	Yes	Yes	No
yuniql	No	No	Unknown	Yes	Yes	Yes
Alembic	Yes	No	Yes	Yes	Yes	Yes

Table 2.2: Various version control tools with features

Other vendors might also change their licensing down the road, this will not be a huge deal if the solution is open source as the community will likely fork the project and support it. This has happened with MySQL in 2009, the fork is called MariaDB.

2.5 Schema Migration

2.5.1 Database version control tools

In a continuous development project the requirements change over time. A changing database schema is one of the results of that. The database management system already includes ways to change the structure of the database in the form of DDL statements. However this can get complicated if performing complex changes that include (foreign key) constraints. DBMS also do not provide a nice way to keep track of various versions of the schema.

Various tools exist to keep track of different versions and to provide a script to automatically update (or rollback) the schema. Most of these tools are programming language independent and provide a structured way to see which changes occurred in which version of the database. Table 2.2 shows different tools together with important features that it supports.

The composability feature in the table tells us if the tool can keep track of various versions as 1 atomic change. These are mostly files that specify the incremental versions of the schema. Transactional means if the update or rollback is performed in a transaction and can therefore be rolled back in case of an error. As for the automatic rollback, this tells us if a tool can automatically generate the exact opposite statements for a set of changes. Some tools let you specify what the undo procedure should be while two tools do that automatically. Some tools do not provide an undo procedure and just tell you to update the schema with the opposite statements.

2.5.2 Information loss

Any data that has been deleted cannot be recovered, if the deletion is done properly. But what if after the data is deleted a rollback is required because the change was not satisfactory. A method to achieve this is to soft delete the data by either hiding it or moving it to a temporary location where it will stay dormant for a certain amount of time before it is permanently deleted. However, there is still a problem with this approach with regard to schema changes. If two columns get merged from a table and the opportunity to roll back should be available, the original data in the two columns should still be available if no function exists that can reverse this merge. An example is if both columns contain integers which

[9] Supports at least PostgreSQL, SQL Server, MySQL & Oracle DB

gets summed up in one column. A simple one-way function is needed to go forward but to go backwards is not possible. 2 + 2 = 4, but 4 can be divided into infinite sums (or only 5 if counting the whole numbers $\mathbb{N}_0$). Some extra information needs to be stored to provide a rollback, which is in this case one of the original columns (the other can be calculated). In other examples you need all the original data.

But what if new data is created or data has been updated that did not exist before the schema change. The original data never existed and can thus not be stored. To still provide a rollback the application should either provide a default value for the data in the original format or internally work with the old data format. If eventually the change is so far in the past that the possibility for a rollback is not required anymore, a hard switch can be made where applications can switch to the new version and definitely delete the original data. In the mean time, applications can decide to use the new column only for displaying purposes and create or update information using the old format.

2.6 Software development

Databases are almost never used as a standalone service and are mostly coupled with various applications. This software does not pop up out of nowhere. Software engineers and programmers have to design and code the various parts that the software should be capable of doing. To facilitate this process and provide an organized manner or working, several methodologies have been developed to give structure to this process. Originally the waterfall method was the standard methodology that was being used. It came from the mechanical engineering field in which clear requirements were known and almost everything could be designed beforehand. However, this methodology causes a lot of friction with software engineering which is quite different. Requirements are almost never known beforehand and the scope will surely change during the development, partly due to relatively quick improvements in the software world. To solve this problem, the agile methodology was developed.

2.6.1 Agile

Agile characterises itself with small short improvement 'sprints' focusing on building a working prototype. This contrasts with the waterfall approach that first designs the solution, locks requirements and then starts building the solution. Agile is a popular buzzword and used by a lot of managers to indicate that their organization moves with the time. It provides several advantages when clients want software but don't know all the details yet. Agile is lenient in what can be done at what stage of the project. In more recent years Agile is being rolled out to large corporations with thousands of employees. These corporations provide applications to millions, if not billions, of users worldwide. These applications need to be maintained and improved but users are not happy with big sudden changes to their environment, many corporations now provide small incremental improvements. However, constantly updating the application can take a lot of time if some aspects are not automated.

2.6.2 Continuous integration & continuous delivery/deployment

With the advent of small incremental improvements came the need for automatic testing and deployment. Manually testing every release becomes tedious and running all checks before a new version comes live is time consuming. Continuous integration and continuous delivery/deployment is the new way to provide these small incremental updates to applications. Continuous integration is the practice of constantly merging small improvements into the main production branch. To make sure these improvements do not break important features developers have to write unit tests that automatically test functions and features of the application.

Continuous delivery or deployment is the practice of (automatically) deploying this branch to production at regular time intervals. Sten Pittet of Atlassian [8] describes a subtle difference between delivery and deployment. Whereas with continuous delivery the push to production is still manually done every x amount of time, say weekly. With deployment this is done automatically as soon as the automated tests return a success. Developers can already see the change they made a couple minutes after they push their improvements to the production branch. This does not come with only upsides as for every small feature or bug fix a new test has to be made or an existing test has to be altered.

The role of load balancers

A load balancer is a sort of entry point that sits before the servers that hosts the main application and will send users to different servers to balance the load on these servers. Via this way, whenever a new version of the application comes online, users can continue using the application on a server that has not updated yet. Load balancers will gradually send users to servers with the new version and servers with the old version will receive less requests. Whenever a server with an old version stops receiving connections it can be updated to the latest version and receive users once more. This type of update can be called a zero-downtime rolling update. For most front-end and back-end updates this works great, except when an update to the underlying database schema is required.

Problems with CI/CD in databases

Due to the way databases are designed which is already explained in an earlier section, relational databases like PostgreSQL can only be scaled up, not out. This means that only 1 instance of a database can be running at a time. This creates two problems when developers want to update the database without downtime. One of these problems is that updating the schema also means updating the data that depends on it because the database cannot be in an inconsistent state. If the data happens to contain millions or billions of objects, it can take quite a while to complete during which the database is unresponsive to new queries.

The second problem that plagues updating the database without downtime is that the applications also need to be updated to reflect the change in the schema. But this updating causes there to be at least two different versions for a short while. Both application versions expect a different database schema that governs the data. These two problems are why currently many developers postpone updating the database schema until it is not reasonable anymore to work with. However, this is not according to the principles of agile and continuous integration and continuous deployment.

Chapter 3

Literature Review

This chapter contains the results of the literature review that has been conducted. This chapter will look into the literature about (online) schema evolution and the state-of-the-art existing solutions that tackle zero-downtime schema migrations but will also discuss their flaws or shortcomings.

3.1 Exploratory literature review

At first, an exploratory literature review was conducted. Through the library of the University of Twente and Google Scholar, different articles on database schema migrations were found. Initially, some 15 articles were found that explained the problem well. Some articles were from researchers at the University of Twente [9] [10] [11]. Also, a paper by Michael de Jong [12] was instrumental in understanding the subject. References in these articles were checked and interesting titles were picked out. These include one from Ronström [13] about his method on online schema updates for a telecom database and a paper by Gary Sockut [14] about the online reorganization of databases.

3.2 Systematic literature review

After a first look into the subject, something more systematic is needed to ensure no important articles are missing in this literature review. Some keywords were brainstormed and put through Scopus. These keywords include the following: (”online” or ”*downtime” or ”*blocking”) and (”database” and/or ”schema”) and (”evolution” or ”reorgani?ation” or ”transformation” or ”migration” or ”update” or ”refactor”) and not (”bio*”). The exclusion of the bio* keywords was done because several biology articles contained keywords online, database and evolution. This search returned 116 articles of which the title and abstract were read. Of the 116 articles, 29 were relevant. A large part of the papers was about NoSQL and the Semantic Web, these were included in the relevant list if there was some mention of it being online and without downtime.

It became clear that authors named concepts differently. High availability and continuous development were terms that were used instead of online or zero downtime but also a couple of other synonyms for transformation were used, such as upgrade, reconfiguration, and changes.

3.3 Schema evolution

The evolution of database schemas has been widely studied as far back as 1979 by Sockut et al. [15], in that paper they talked about all kinds of reorganization that a database can have but also changes to the

schema. In the paper, they also mentioned the problem of having to block user access while maintaining the database, the problem that is currently also being researched. A paper by Hick et al. [16] developed a strategy to evolve the database schema in which all steps must be recorded in the conceptual level, logical level, and physical level as well. A study by Noy & Klein [17] shows that the evolution of schemas is not the same as the evolution of ontology (”schemas for knowledge bases” as they call it). They say that there need to be techniques to determine how compatible one version is with another even without traces.

In a short literature review paper by Roddick [18] in 1992, some papers have mentioned that looked at temporal databases and how to keep data that were created in a previous schema version into the next version.

Another possible solution to schema evolution is the schema-on-read strategy [19] that is used in many big data environments where raw data have to be stored. With this strategy, certain data are saved according to a schema in the application that requests the storage. When the applications read the data, it does not know in which format the data are stored, the application must first determine it. This could be useful if applications store the various versions of the database schema and their respective upgrade paths. In this way, applications can work with multiple versions of schemas at the same time. However, the database also has to support this, RDBMS such as PostgreSQL typically do not want to support such strategies as it will be hard to guarantee certain properties such as ACID.

A paper by Kaas et al. [20] in 2004 mentions that work had been done on schema evolution but did not consider the special characteristics of multidimensional schemas and queries. They mention that there has been some research into querying across multiple versions of star and snowflake schemas. The authors of the paper believe to be the first to present a study that investigates the evolution properties of star and snowflake schemas and the impact of existing queries. The paper concludes that star schemas are equal or better than snowflake schemas in all aspects, though they only investigated inserts and deletions in various levels of the schema. Something to note is that the paper handles OnLine Analytical Processing (OLAP) instead of OnLine Transactional Processing (OLTP) which can be implemented in different systems but the authors mention that the most common way it is implemented is via relational database management systems.

From the paper by Ronström [13], which will be discussed in a later section, a couple of papers can be found in the references that deal with schema evolution and how data have to be transferred to the new version. One paper developed a new language to accommodate the transfer of objects from an old schema to new schema objects [21]. Another paper [22] published a tool to gracefully evolve schemas and the data that it contained. The tool is called PRISM but searches on the internet reveal that the tool never became widely used. Something to note is that there exists a demo page, but it has been taken offline. This tool does not provide online schema evolution but focuses on automatic query rewriting to reduce the cost of evolving the schema. In a paper by the same authors, [23] they mention the following: ”The big players in the world of commercial DBMSs have been mainly focusing on reducing the downtime when the schema is updated.” They cite a paper by Oracle [24] on their efforts to reduce downtime with the DBMS_REDEFINITION package introduced in Oracle Database 10g version 2.

3.4 Online schema transformations

3.4.1 Definition

Many words have been used in the literature to describe roughly the same concepts. This research will use the following definition of online [25] according to The Free Dictionary: ’In production or operation, often as part of a supply chain.’ This means that the schema transformations happen while it is in

production or operation. Other words that are used in the literature are 'zero/without downtime' or 'non-blocking'. While these words refer to a slightly different concept, in this research the meaning of online will be meant.

Downtime is also mentioned a lot in literature. It is highly related to the definition of online that is described above. Downtime is the opposite of online and therefore has the following definition: 'Not in production or operation, often because of a malfunction.'

Another word that has many synonyms in literature is transformation. The other words found are the following: 'evolution', 'reorganization', 'migration', 'update', 'refactor', 'upgrade', 'reconfiguration' and 'changes'. In the context of this research the definition of change [26] according to The Free Dictionary will be used: 'To become different or undergo alteration.' In the context of this research, it is the alteration of the database schema. The combination of these two definitions is as follows: 'the alteration of the database schema while the system is still operating and able to serve clients.'

3.4.2 Research

Research into online schema transformations has been conducted in a couple of ways, some have looked at the strategies that must be used to perform online schema transformations [15]. Others have researched the performance decreases and which transactions are blocking when performing a migration [9] [10], while some have taken it upon themselves to create solutions [22] [12]. Also, Oracle acknowledges this problem and created a partial solution [24]. With the DBMS_REDEFINITION package introduced in Oracle Database 10g version 2, Oracle made it possible to do some transformations online such as column addition, renaming, and deletion.

One of the more known papers in the online schema transformation community is the one from Ronström [13] about doing schema transformations on a telecom database that must stay online no matter what. Ronström proposes a method to use triggers to keep two separate schemas in place and keep the data in both updated using triggers. The method consists of five phases, where the first two phases prepare the new schema and the last three validate and either undo or commit the change. Ronström distinguishes between soft schema changes and hard schema changes. With a soft schema change, new transactions can use the new schema, and old transactions use the old one concurrently. Concurrency problems are smaller according to Ronström. A hard schema change however waits until the old transactions stop, makes schema changes and the new transactions can start again. An example is if the old schema has two columns, namely hours worked and payment per hour, while a new schema contains only total payment. There is no way for a value in the total payment to go back to hours worked and payment per hour.

3.4.3 Simple & complex schema changes

Ronström also distinguishes between simple and complex schema changes. A simple schema change can be done in one transaction, while a complex one requires multiple transactions that otherwise would be a long-running, blocking, transaction. An important part of a database system is the ACID properties. This means that the transactions should either all complete or none of them. Also if the system fails, it should not be in an inconsistent state. Therefore, Ronström proposes to work with the SAGA pattern described in a paper by Garcia-Molina et al. [27], this pattern is widely used in the micro-services architecture to provide a consistent state between self-contained applications. SAGAs propose also storing an UNDO transaction in case any transaction in the sequence fails. By following this UNDO transaction, the complete system will be restored to a state before the first transaction in the sequence started therefore guaranteeing atomicity but also consistency by having a transaction that can be performed in case of a system failure where some things had been committed. Ronström uses a special SAGA table

to store these transactions, which is roughly the same as how Liquibase, Flyway, and other schema evolution tools operate. The paper subsequently discusses the various schema changes and how they can be implemented using this method.

3.4.4 (Non-)blocking SQL DDL statements

To provide zero downtime a query to the database must not be blocked or fail during the schema transformation. This means that the existing structure of the database cannot be changed in any way as that would cause failure in existing queries. Blocking transformations thus must be rewritten to a series of non-blocking transformations and not alter the existing structure. Some research has been conducted in this area by Wevers et al. [10] but found that different database management systems handled this differently and could not get some complex transactions to work online on some DBMS. Oracle seemed to have developed the DBMS_REDEFINITION package in 2005 but it comes with restrictions[1]. It also does not seem to be continuously updated as no mention can be found online of any updates.

3.4.5 Online migration strategies

Various online migration strategies can be identified, of which Rönstrom's method was already discussed. Other strategies that are identified are called the fuzzy copy method by Hvasshovd [28] [29] and the materialized view creation method by Løland and Hvasshovd [30]. The fuzzy copy method describes how to copy a table without blocking access to this table and still have all data up to date. This can be used together with other methods to perform online migration strategies.

The materialized views method is a bit more substantiated and offers a more complete strategy to perform online migrations. In this strategy, new schema versions are first created in a materialized view which can already be used by applications running the new version. This has also the advantage of first being able to test if things work and later committing the change.

3.4.6 Data migration

There are multiple data migration strategies discussed in the literature and recently researchers are focusing on NoSQL systems [31] [32]. These NoSQL techniques might be translatable to normal SQL DBMS but more research has to be conducted into that. The main types of data migration are eager data migration, lazy data migration, and hybrid data migration. For data to be eagerly migrated it means that as soon as a schema change is made all data will be transformed to the new format. This mostly means that the DBMS will either take a significant performance hit or block all transactions until the data has been transformed. This is the preferred method when having a planned maintenance window because it means that all data will already be in the correct format when bringing the service back up.

Lazy data migration is not as straightforward as one might think. It will only migrate data when it is requested. This seems fine until you think about what has to happen to data that it depends on, this data must also be migrated before the current data is migrated. And what if database objects are several schema versions behind? Several solutions for lazy data migration are proposed to solve this problem. Wevers et al. [11] propose queue operations as suspended computations in a tree structure. The solution is not directly usable in RDBMS as it uses key-value data stores, but it might be used as Rae et al. [33] build an RDBMS on top of a key-value data store with Google's F1 DBMS. Sheng et al. [34] also propose a lazy schema change method to perform online schema migrations with less performance degradation than eager migration. Neamtiu et al. [35] build an implementation that includes

[1] https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-tables.html#GUID-CB5589F0-B328-4620-8809-C53696972B4C

lazy migration in SQLite to perform ”online” schema migrations. However, this approach does not allow for two versions of the schema at the same time.

The combination of eager and lazy data migration is called hybrid data migration. Two strategies are proposed by Klettke et al. [31], one is called incremental migration, and the other is predictive migration. With incremental migration once every x schema versions an eager migration is performed to get every data object to the same version. This is only possible when data objects can be more than one version behind. The other strategy, predictive migration, might be better suited for applications where data can only be a maximum of one version behind. In this case, all data is lazily migrated except for a background process that migrates the data one by one. Depending on the algorithm being used it starts with the objects that it predicts to be accessed first. This method is especially useful if you want all data to be eventually on the same version while not wanting to take a performance hit at the moment when you migrate the schema.

3.4.7 Criteria

This section will consist of the criteria and requirements found in the literature for such an online schema migration solution. Five articles were identified that listed criteria in 4 categories: functional, performance, correctness, and tolerance of errors.

Performing online schema transformations is still a problem to be solved as is apparent from the work of Wevers et al. In their research they ask the database community to come up with solutions as they believe currently none exist that meet all their criteria for a good solution. They mention that there is a need for a solution as evident by the multitude of tools currently developed in the industry [9]. Wevers et al. have published multiple papers and have provided the community with a set of criteria that the solution should support [9]. These consist of two categories, namely functional and performance. These criteria include ideal behavior but also what could be acceptable if such behavior could not be reached.

A paper by Sockut et al. [14] also specifies requirements for the online reorganization of databases. They split the chapter with applications that require online reorganization and the required characteristics of strategies to perform online reorganization. The following are requirements for the latter split into four categories, namely functional, correctness, performance, and tolerance of errors. Correctness is defined as being correct with a specification, in their case following the SQL specification.

Three papers that implemented a solution also came up with their own criteria. These papers of Sheng et al. [34], Zhu et al., [36] and De Jong et al. [12] provided some requirements that their solution had to satisfy. These criteria have been summarized in Table 3.1 and the following sections will more clearly describe each criterion in these categories.

Each author divided these criteria into categories with different names and some criteria were listed under different categories. Criteria were reclassified according to the following taxonomy. The functional criteria state that the solution should be capable of certain tasks or actions to provide more functionality to the user. This is different from the correctness category as that one means that certain guarantees should be ensured internally. The performance category lists criteria that relate to performance requirements in the database system. Lastly, the tolerance of errors category lists criteria that deal with issues relating to both system and human errors.

Functional criteria

Wevers et al. specified five functional criteria according to their definition. Three of these criteria are better classified under the correctness category and will therefore not be discussed here but in a later section. The first criterion that will be discussed is the expressivity criterion, this criterion implicates that

Category	Criteria	Description
Functional	Expressivity	All transformations can be performed online
	Declarativity	The user should not have to deal with specific implementation details
	Composability	The user can perform many transformations in one go
	Concurrently active schemas	Multiple schema versions can be active at the same time supporting the same data
	Non-invasiveness	The solution should not require major work to applications to support it
	Aborting	The solution should not abort running transactions
Correctness	Transformation of data	All data should be available in the new schema version
	Transactional guarantees	The solution should satisfy the ACID properties
	Application migration	A client should be able to continue querying the database with its current schema version
	Referential integrity	Foreign key constraints have to be enforced at all times
	Schema isolation	Clients should not be able to see any other schema version than their own
	Consecutive migrations	The solution verifies that no other transactions are updating the schema
Performance	Performance degradation	Performance degradation should be within limits depending on the use case
	Space consumption	The solution should not significantly fill up available storage space
	Finite completion time	The solution should perform the transformation in a finite time, preferably as fast as possible
	Time to commit	Data should not take too much time to transform after clients are already using the new version
	Memory usage	The solution should not take up more memory than is allocated to the system
Tolerance of errors	Data recoverability	The solution should have the means to reverse the transformation
	New data reverse transformation	New data should have a transformation back to the previous version
	Transformation resilience	Transformations can be aborted without inconsistencies, preferably restarting from a checkpoint
	Successfully online	The new schema version should only come online when it is successful

Table 3.1: All criteria found in the literature

every schema transformation can be performed online. Wevers et al. mention that this would be the ideal case, however, it could be sufficient to allow for a subset of transformations in practice.

Zhu et al. also coined this criterion. According to them, the solution has to be able to handle both simple as well as complex scenarios. Examples of such complex scenarios are given as multi-table joins and mergers. However, also in this case some complex scenarios might be achieved by allowing for only a subset of transactions to achieve the same result.

The second criterion that falls under the functional category is declarativity. What they mean by that is that the user should not have to deal with the implementation details of the transformation. If users have to manually declare actions to perform certain operations online there will be a high chance that mistakes are made and the integrity of the database can become compromised. Wevers et al. mention that the SQL data definition language can be considered declarative.

A criterion from De Jong et al. describes the same functionality as the declarativity criterion above. They mention that software engineers should have the ability to perform transformations in one go. They should not have to deal with developing and deploying intermediate versions. They call this criterion 'Schema Changesets' as Liquibase provides these declarative changesets.

A second criterion from De Jong et al. that falls under the functional category is the non-blocking schema changes. This criterion is the same as the expressivity criterion from Wevers et al. as it says that schema changes should be online and not blocking any queries issued by database clients.

The third criterion of De Jong et al. is concurrently active schemas. De Jong et al. mention that this criterion avoids putting restrictions on the method of deployment when upgrading web services depending on a database schema version. At least two schema versions should be able to be active at the same time because applications might not be able to perform a big flip upgrade.

Non-invasiveness is the fourth criterion from De Jong et al. in the functional category. They state that the solution should require as little change to the integration with the application as possible. This criterion will ensure that databases with a lot of depending applications will not require immense refactoring to allow for the solution to work. Preferably, any solution will totally transparent to the clients accessing the database.

Wevers et al. also coined some performance criteria of which some can be rallied under functional

criteria. One of those is similar to the non-blocking schema changes of De Jong et al. However, Wevers et al. do allow some blocking for short periods of time depending on the application, but not more than a couple of seconds each time.

A second performance criterion that actually falls under the functional criteria is what Wevers et al. call aborts. The solution should not abort already running transactions. They propose that snapshot isolation and translation into the new schema is a way to solve this issue. As well as with the previous one Wevers et al. do propose that some short-running transactions can be aborted if they do not suffer from starvation. Starvation in a computer system refers to the problem where a concurrent task is repeatedly denied access to a resource to the point where the task will never finish.

Correctness criteria

As was already mentioned, the correctness criteria can be confused with the functional criteria but provide a different meaning of ensuring certain guarantees. Three of the five functional criteria determined by Wevers et al. can be placed under this category. These criteria are so-called hard criteria in that if they are not satisfied a tool can not be called a solution as it does not support essential operations or provide essential guarantees.

The first criterion in this category is called the transformation of data by Wevers et al. This criterion specifies that all existing data should be available in the new schema. This obviously does not count for columns that were dropped or other constraints that were lifted in the new version but all other data should be available in their original format or in the transformed format which was specified by the schema change.

The second criterion is the transactional guarantees already specified for OLTP transactions. These transactional guarantees are to ensure the database integrity and correctness of the database programs. This guarantee can be fulfilled if the schema transformations satisfy the ACID properties: Atomicity, Consistency, Isolation & Durability.

The third criterion is called application migration. This implies that existing clients can continue to operate during and after the schema transformation. If this is not the case the solution cannot be called online and therefore not be a solution at all. Wevers et al. mentioned this criterion but so did Sheng et al. and Zhu et al. However they only specified that during the installation of the schema old clients still have to be able to retrieve and update and not necessarily after.

As for correctness criteria from De Jong et al., one of them is referential integrity. Foreign key constraints have to be enforced at all times to ensure that there is no way any action can leave the data in an inconsistent state. This applies to both normal use and during the evolution process.

Another criterion from De Jong et al. is schema isolation. Clients should not see any other schema version apart from their own version. This is to ensure that clients do not get data they did not expect or update data into the wrong schema version causing all kinds of problems.

Sheng et al. go one step further by saying that transactions should not be able to see any other schema version. However, this is a consequence of how their solution is implemented. Transactions might have to request data from other schema versions to provide the client with their version of the data. In that case, the above-mentioned criterion by De Jong et al. might be better as to not limit the solution too much.

Sheng et al. further brought up a new criterion not listed by the other authors. This one can be called consecutive migrations and states that the solution should check if any existing transactions are already performing DDL statements and if that is the case not perform the current transaction until after the other one committed. Because the database has to continue to serve clients no lock can be requested on the tables. With this in mind, a DDL transaction cannot lock a table and therefore not prevent a new DDL transaction to also change the schema.

Performance criteria

Three performance criteria were distilled from the paper of Sockut et al. about online reorganization. These criteria are vaguely described but give insights into the performance that has to be expected from the solution. The first criteria mentioned is that the performance degradation experienced by the user should tolerable. Sockut et al. define performance degradation as the average increased time it takes for a user transaction to complete while reorganizing the database. To achieve this criterion either more system resources should be allocated before reorganization is started, the reorganization should be happening at times when users do not request much from the system or the reorganization should be run with a low priority to not impact user transactions.

The second criterion mentioned in Sockut et al. is that the space consumption of a reorganization should be tolerable. Depending on how much reorganization is going on, the methods that are used, or the size of the database, not enough space can be allocated. At today's prices per terabyte, storage is rather cheap and will be getting even cheaper still. This does not mean that no limitations on storage space exist. Sockut et al. does not mention what type of space is meant, limitations in the amount of RAM that the system has can also greatly impact performance.

The third criterion in the paper of Socket et al. is that the process of reorganization should eventually finish. This seems like a given that that should be the case but if user transactions appear quicker than that the reorganization can handle it will never finish. This will be the case if the reorganization is given the lowest priority and user transactions almost take up 100% of the resources of the system.

Not only Sockut et al gave performance criteria but Wevers et al. also mentioned six. These criteria are divided into two categories, the impact of schema transformation on concurrent transactions and performance criteria for online schema transformations. Two of the former were already discussed in the functional category and will therefore not be discussed here. Wevers et al. also do not provide a clear cutoff point for when a criterion is satisfied and when it is not, this entirely depends on the application of the solution. The first criterion is similar to a criterion already mentioned above and it is called slowdown. This just means that the decrease in throughput and increase in latency of transactions should be acceptable to a certain degree.

The rest of the performance criteria of Wevers et al. apply to the performance of the online schema transformations themselves. The first criterion they observed is that the solution should have a relatively low time to commit. What they mean by that one is the time that it takes the data to be transformed to the new schema (or old schema for that matter) after the online transformation has already been completed. This is especially important for the lazy migration approach that will be discussed later as it will reduce the performance of the whole system drastically if this value is too high.

Just like the space criterion of Sockut et al. in which memory is not mentioned, Wevers et al. does mention a memory criterion. They say that ideally a transformation should be performed in-place and not construct copies of data. However, depending on the application and available hardware resources, a solution can request additional memory or use memory swapping to complete its transformation.

Tolerance of errors criteria

The tolerance of errors criteria is only coined in the article by Sockut et al., however, some criteria from the other papers also fall under this category. These criteria deal with errors and how the solution should handle these. Sockut et al. mention three criteria of which the first is data recoverability. Data that is being transformed to a new version should always have a reverse associated with it so that if the new version is not satisfactory it can be transformed back into its original state. Sockut et al. state that reorganization by copying data can serve as the basis for this criterion. Zhu et al. also coined this criterion that data should have a reverse if anything goes wrong.

The second criterion that Sockut et al. present in their paper is that any user action that has become redundant due to online reorganization has to be logged in case of a rollback. If the new version of a database schema does not store a variable that existed in the previous version it must still be saved or logged to ensure the correctness of a possible rollback when a problem occurs. Not doing so might leave the database in an inconsistent state or cause constraints to be violated. This criterion is almost identical to the previous one in that data should have a reverse transformation but this criterion goes a step beyond in that new data should also have a reverse even though it never existed in the original version.

Sockut et al. also mention a third criterion to tolerate errors which is that online reorganization must be restartable if an error occurs. Not doing so can cause your database to be in an inconsistent state. Sockut et al. also mention the term checkpointing as a possible strategy to reduce redoing a large amount of work when an error occurs. One might argue that this criterion can fall under the correctness category but due to it being related to helping prevent errors it is moved to this category.

Not only Sockut et al. specified tolerance of errors criteria but also De Jong et al. specified one. They call it resilience, the database should always remain in a consistent state. If anything fails the solution should be able to roll back and return to the original state without affecting the database clients. This is similar to how Sockut et al. described it.

Wevers et al. also specified a criterion that deals with resilience. They call it abort and recovery, and it deals with what has to happen when issues come forward. They mention that it is acceptable that transformations are aborted before they are started when concurrency issues are detected but should not be aborted during the transformation. Ideally, the solution can recover and continue with the schema transformation after a failure occurs but because system failure is rare, safely aborting and rolling back might be acceptable. New schema transformations should only be rejected if another schema transformation is still uncommitted and will cause a conflict.

A criterion by Sheng et al. might sound obvious but is important nonetheless. It says that a new schema version should only come online whenever it successfully commits. Sheng et al. do not mention how a solution must achieve this. Upon failure, the solution should likely remove any intermediate steps so no one can interact with those in any way.

3.5 Benchmarks

To quantify potential performance loss or gain in case of gained hours not spend on downtime for that matter, Wevers et al. specified a benchmark framework for online non-blocking schema transformations [10]. In this paper, they again urge the database research community to address the open issue of non-blocking schema transformation. This benchmark can be useful in determining if a solution fits certain criteria already mentioned in previous sections. They do provide characteristics of an ideal solution but no real solution.

A paper by Möller et al. [37] provides an overview of online schema evolution benchmarks of which the above one from Wevers et al. is included. Möller et al. found 5 benchmarks in literature dating from 2008 to 2018, they are the following in chronological order: Phanta Rei [22], STBenchmark [38], BigBench [39], Twente [10] & Unibench [40]. In their overview, they mention that 3 out of 5 benchmarks do not directly deal with online scheme evolution but share many similar concepts that can be used as a benchmark. They found that all benchmarks want to model real-world applications while only 1 uses actual data from a real-world system, the one from MediaWiki. They also found that all benchmarks use queries and statements typically found in their specific application domain to measure the performance. Some used complex queries while others use analytical queries. One benchmark has the performance metric to see how many queries were still successful after evolution. Most benchmarks also generated

Category	Requirement
Data model	The benchmark should consider data variety, this includes relational data as well as semi- and unstructured data
Data generation	The benchmark has to generate data to prove the correctness of complex queries
Data generation	The benchmark should provide a parametrizable, yet deterministic data generator
Queries	The benchmark should distinguish between SQL statements (initial queries) and SMOs statements (queries to move to a new schema version)
Queries	The benchmark should use queries or statements that are typical in the specific application domain
Metrics	The benchmark should measure performance. If data is migrated eagerly, it refers to the efficient immediate transformation. If data is lazily migrated, it refers to the on-demand transformation time.

Table 3.2: Benchmark requirements

their own test data, except for the benchmark that referenced the MediaWiki dataset. Möller et al. propose that benchmarks provide a parametrizable, yet deterministic data generator to provide flexibility and reproducibility of benchmark results.

Table 3.2 list the requirements Möller et al. identified for a standardized schema evolution benchmark.

3.6 State of the art solutions

Some authors have taken it upon them to come up with solutions, an overview of these solutions can be found in Table 3.3. Some provide similar approaches while others present other innovative solutions. When looking at the criteria listed in one of the previous sections, it can be concluded that the DBMS has to support 2 concurrent schema versions that still support the same underlying data.

3.6.1 QuantumDB

One solution by Michiel de Jong called QuantumDB [12] uses a proxy, a middleware application, that sits between the database driver and the application. It translates requests depending on the schema version the application expects. To support 2 schema versions the underlying data gets copied to the table that is in the next schema version and gets synced by using triggers. QuantumDB will then translate the table name that the application requests to the table name that is present in the database and is therefore transparent to the application itself. This does provide overhead by having 2 copies of the data but how much performance degradation there is, has to be tested.

3.6.2 Ratchet

Another tool called Ratchet by Yu Zhu [36] uses a similar setup but instead of just copying all data to a second table uses materialized views. The advantage over the previous solution is that data does not needlessly have to be copied as the underlying data is essentially the same. However, for this solution to work they had to modify the database server a bit. Another difference with the aforementioned solution is that it does not reuse the database driver but instead uses a custom-made RPC proxy that is capable of performing the changes and queries. This makes it that all applications need to be changed to

incorporate this new way of talking to the database. Due to Ratchet not being downloadable or online, the performance degradation has to be taken from the paper itself.

3.6.3 SqlTable & DataTable in Terrier

A different approach to the above two tools is proposed by Yangjun Sheng [34] by using what he calls SqlTable & DataTable. His method modified the storage engine of an open-source database to separate how databases store and present tables. This way a new presentation can be made in SqlTable while the previous view stays the same. This solution is also not downloadable nor found online. Due to the use of the DBMS Terrier, no request has been made for the source code as it is probably implemented completely different than how PostgreSQL works. Nevertheless, important findings can be extracted from this paper.

3.6.4 InVerDa

InVerDa (Integrated Versioning of Databases) by Herrmann et al. [41] is arguably the most advanced attempt at an online migration tool. It uses the same concept of views as Ratchet in combination with their own Database Evolution Language, which is explained further in a section later in this chapter. It can automatically set up bidirectional triggers to keep new data synced with the old data by use of auxiliary data tables. These additional tables and views can be materialized again to start anew when a schema version is deemed stable. On their about page[2], more information is displayed as well as a demo application. However, this service is down. A request for the source code has been made, but in a reply, it was mentioned that the source code has been lost.

3.6.5 DECA

DECA (Database Evolution for Cloud Applications) is described as implementing InVerDa in cloud applications and has been developed by Mohapatra et al. [42]. The main author of InVerDa is also a co-author of this paper. It proposes an architecture to integrate DECA into existing tools. The paper mentions that it does not in any way modify the DBMS but runs as an independent component. They mention that they define three triggers for each view, this causes a large overhead when row-wise triggers are used. Not all DBMS have implemented statement-wise triggers, PostgreSQL has implemented it. The same story as above applies to this tool, it cannot be found online and the professor that oversaw this paper has said that the source code cannot be found.

3.6.6 PRISM/PRISM++/PRIMA

PRISM (Panta Rhei Information & Schema Manager - 'Panta Rhei' (Everything is in flux)) is the oldest tool that attempted to provide online schema migration support. Curino et al [22] [23] [43] [44] and Moon et al. [45] developed this tool with 5 needs that according to them had to be addressed. The first being a language to express complex schema changes. The second is an evaluation of the effects that such changes are about to have on the database itself. The third need is the optimized translation of old queries to the new schema version. The fourth one being the automatic migration of data to the new schema. And the last one is full documentation on change history. PRISM also presents schema modification operators which will be discussed in the next chapter. Just like the other solutions, this one was also not downloadable, a demo site was made but is offline. No request was made to acquire this tool as it is rather old (2010). It looks, however, quite promising with what it can do.

[2] https://wwwdb.inf.tu-dresden.de/research-projects/inverda/

Online migration tool	Downloadable	Open source	Last update	Author	Comments
QuantumDB	Yes	Yes	Sep/18	De Jong	Proxy
Ratchet	No	No	Oct/17	Zhu, Yu	Proxy
SqlTable & DataTable in Terrier	No	No	2019	Sheng, Yangjun	Storage engine overhaul
InVerDa	No	No	Sep/17	Herrmann	Proxy
DECA	No	No	2018	Mohapatra/Herrmann	InVerDa applied in cloud applications
PRISM	No	No	Nov/10	Curino/Moon	Automatic rewriting of queries
pt-online-schema-change	Yes	Yes	Jan/21	Percona toolkit	Copy table + atomic rename table
Google F1 (Spanner)	No	No	-	Ian Rae	Cloud
Oracle package	Yes	No	-	Oracle	Some tools
Oracle Edition based	Yes	No	-	Oracle	Multi version views
Kvolve	No	No	Apr/16	Karla Saur	NoSQL key-value store
gh-ost	Yes	Yes	2020	Github	Copy table + atomic rename table
OSC	Yes	Yes	2021	Facebook	Copy table + atomic rename table

Table 3.3: Various online migration tools

3.6.7 Other tools

The above tools all account for the serious attempts at solving the issue of online schema migration. Multiple other strategies and packages have been made to help developers change the schema. pt-online-schema-change from MySQL is one of them, it will work on a copy of the table and at the end atomically switch the name of the table.

Also, gh-ost[3] by Github and OSC[4] by Facebook use the above strategy of copying tables and at the end atomically switching table names. To use this strategy, applications have to support 2 schema versions. When the switch happens the applications can continue working. These are not complete solutions as there is still some downtime between switching table names and the solution is also not as transparent as the other solutions.

The package by Oracle called DBMS_REDEFINITION uses materialized views and has the possibility to roll back and abort. However, this approach is all but transparent to the developer with a lot of added complexity. Also, the Edition package by Oracle uses views to create different schema editions but this solution is also complex to use and not transparent.

Google (Rae et al. [33]) also came up with a solution to schema evolution for their distributed relational database. This database is a little bit different as it is a relational database management system built on top of a key-value store. The advantage of this design is that the system can be massively distributed while still following a relational schema. This, however, did come with some problems they had to solve because of many servers not simultaneously migrating to the new schema version. The protocol they described allowed them to have 2 schema versions concurrently without any data corruption. In their solution 2 intermediate states are proposed called ”delete-only” and ”write-only”, this allows old or new schema versions to not break constraints. The solution at Google has limitations in that only 2 schema versions can be used simultaneously. Rae et al. also mention that it does not support common DDL operations because reorganization overhead can be amortized if multiple changes are batches together.

It seems only QuantumDB can be found online and is open-source. The other solutions either provided a demo that has been brought down or did not publish any code or executable. The inner workings of these solutions can be deducted from the papers.

SMO	(Bi)DEL	ICMO
CREATE TABLE	CREATE TABLE	ADD PRIMARY KEY
DROP TABLE	DROP TABLE	ADD FOREIGN KEY
RENAME TABLE	RENAME TABLE	ADD VALUE CONSTRAINT
ADD COLUMN	ADD COLUMN	DROP PRIMARY KEY
DROP COLUMN	DROP COLUMN	DROP FOREIGN KEY
RENAME COLUMN	RENAME COLUMN	DROP VALUE CONSTRAINT
DISTRIBUTE TABLE	DECOMPOSE TABLE	
MERGE TABLE	MERGE TABLE	
COPY TABLE	(OUTER) JOIN TABLE	
COPY COLUMN	SPLIT TABLE	
MOVE COLUMN	CREATE SCHEMA VERSION	
	DROP SCHEMA VERSION	

Table 3.4: SMO & BiDEL commands side by side

3.6.8 Declarativity & Composability (SMOs & DELs)

To make the life of developers easier some solutions propose the use of Schema Modification Operators (SMOs) [22] [45]. These SMO's can be found in Table 3.4. These consist of commands that are commonly used when changing the database schema and are an abstraction from the underlying SQL code that is generated by the solution. In this way, developers do not have to deal with specific implementation details and the online schema migration solution becomes more transparent.

In a subsequent paper by Curino et al., [43] more SMOs were added to the list. These are the Integrity Constraints Modification Operators and specify the constraints that certain fields in a table can have.

InVerDa and DECA both use such SMOs but based on another language they call Database Evolution Language (DELs). Multiple versions exist of this language, CoDEL [46], InDEL [41] and BiDEL [47], of which BiDEL supports more features and guarantees more properties. Just as with the above-mentioned availability, no implementation is currently available to be downloaded or used. A request has been made to the author for the code.

QuantumDB uses 'ChangeSets' from Liquibase as the language to abstract away the complex inner workings of the solution. The SMOs that Liquibase offers are given in Table 3.5. These are not all yet supported by QuantumDB as evident on the site[5] and the issues on Github[6]. However, these changesets can be used by Liquibase, and due to Liquibase being open source, plugins can be written to extend functionality.

[3] https://github.com/github/gh-ost

[4] https://github.com/facebookincubator/OnlineSchemaChange

[5] https://quantumdb.io/docs/master/

[6] https://github.com/quantumdb/quantumdb/issues

Entity	Create/Add	Drop	Change
Table	createTable	dropTable	setTableRemarks renameTable
Column	addColumn	dropColumn	renameColumn modifyDataType setColumnRemarks addAutoIncrement
Index	createIndex	dropIndex	
View	createView	dropView	renameView
Procedure	createProcedure	dropProcedure	
Sequence	createSequence	dropSequence	renameSequence alterSequence

Table 3.5: Liquibase ChangeSet commands

Chapter 4

Problem investigation

This chapter discusses the problem investigation of the methodology that was used during the master thesis. This part discusses the problem investigation as was done in the Research Topics document. Some of the issues that are currently experienced in the industry will come forward.

As specified by the design methodology book of Wieringa [3], there is a template to construct a design problem. A better understanding can be gained by filling in this template about what it is that has to be created. It consists of the context in which the problem is situated, the artifact to be designed, the requirements to help solve the problem, and the goals of stakeholders. An example of such a filled-in template that would be applicable is:

Improve *time to market of schema updates*
by *designing a tool*
that satisfies *the zero-downtime requirement*
in order to *reduce financial costs associated with bringing down applications for maintenance*.

This is not the only example that can be thought of and not all parts might be known before starting the project. In the above example, only the goals and problem context of some stakeholders are highlighted while for some other stakeholders they might differ. In the next section, observation is done into who the stakeholders are.

4.1 Stakeholders

Stakeholders for these design problems can be numerous. Some of these stakeholders are close to the problem, while others are farther removed from it. The book calls this 'awareness level', the lower the awareness level the less the person either knows, cares, or has resources to handle the problem. Some stakeholders might not even know that they are stakeholders and some have no influence over the final product, but they are stakeholders nonetheless.

In Table 4.1, a list of potential stakeholders has been identified from various sources such as books [48] [49] but also online web pages with job listings. ING has its own job naming convention which can be found in brackets. This list is by no means exhaustive and might not describe every company or organization, but most stakeholders are present.

Awareness level	Stakeholders	Desires	Commitment
High	Database administrator (Ops Engineer - Database & Middleware)	24/7 functional and correct running database with redundancy	Time
	Database developer (Dev Engineer - Database & Middleware)	Working with an understandable database structure	Time
	Database architect (Domain Architect - Risk, Security & reliability)	Designing and implementing database environments on various levels	Time
	Database security officer (Security Engineer - Security & IT Risk)	Creating an impenetrable database system that is not susceptible to attacks from anywhere	Time
	Software developer (Dev Engineer - Coding)	Being able to make applications as if the database is in a consistent manner	Time
	Project manager (Chapter Lead)	Being able to push updates as soon as they are available and tested	Time
	Portfolio manager (Chapter Lead)	Managing an assortment of projects and redistributing assets depending on performance	Time
	Quality assurance (Dev Engineer - Testing)	Making certain that code is ready for production and if not that updates are performed on time	Time
Medium	CEO	Having a solid product/service that customers can depend on	Time
	CFO	Generating profit while reducing costs associated with maintaining products/services	Money & time
	COO	Having an efficient team that can get the work done in the scheduled time frame	Time
Low	End-user	Being able to always use a working product/service that provides value	Money
	Competitor	"Stealing" profit/market share from other competitors	No resources
	Human resources	Healthy and happy employees that do their work correctly and on time	No resources
	Marketing department	Marketing great product features that attract customers that are willing to pay a higher price	No resources

Table 4.1: Overview of potential stakeholders with their desires and commitment

4.1.1 Desires & goals

Goals are desires where stakeholders allocated resources. Some stakeholders are not aware or do not have the resources to spend and therefore have no goal they can obtain. Resources are in the form of a commitment of money and/or time and these desires and commitment can be found in Table 4.1. Since this design problem initially only happens within a company, only one stakeholder allocates money, the CFO. The other stakeholders are employees that commit their time into implementing the solution and working with it. The end-user is also a stakeholder, if this user is a client, they can commit money because they want to always be able to use the service. No immediate conflicts, other than the desire of the competitor, have been identified between stakeholders.

4.2 Conceptual framework

To design a new artifact, the conceptual framework in which the problem lies has to be known. The definition of a conceptual framework is rather abstract and according to the book is a set of concepts called constructs. Table 8.1 in the book gives an overview of what can be meant by it. The background and literature review chapters give a good understanding of the concepts in which this research is placed. Any further clarification or definition of concepts that have not been discussed in either the background or literature review chapters will be explained whenever it is relevant.

4.3 Problem description

To design a good solution to a problem, first, the cause, mechanisms, and reasons have to be deducted why a solution has to be made to solve certain problems. A lot has already been explained in Chapters 2 & 3. Below is an overview of the reasons and phenomena.

With the inception of Continuous Integration & Continuous Deployment and a 24/7 global economy, a resting period for companies does not exist anymore. Systems and services need to be available at all times. It can be night on one side of the world but on the other side, people are using your systems and services. To just flip a switch and update an application or service for people that are still in the process of performing their task is not ideal. At best they can continue performing their task but at worst their task is aborted and they lose precious data and work. It is therefore necessary to only update the application whenever they have finished their task, but in today's world, there is always someone starting a new process. To commemorate this process, the rolling update procedure is used to update people to the newest version whenever they are not using the service for a while. This method, together with the 24/7 economy, does come with the implication that multiple versions have to be running at the same

time, at least for a short while. But there is a problem with this approach, the underlying database cannot natively handle two or more schema states at the same time. This causes some necessary database schema upgrades to be delayed until the cost-benefit trade-off between downtime and further delaying upgrades is not worth it anymore.

Criteria have been defined in literature but have focused most of their attention on the technicalities of such a solution. While that is important, they have not taken it into a business perspective. Important questions to get an answer to are the cost of downtime and how much the delaying of updates would cost. In the previous chapters, it is already described how organizations currently deal with this problem. Existing solutions have also been proposed but have not been implemented yet. Another important question is why organizations have not implemented these solutions yet and what defers them. This would help in designing a solution that will provide value by tackling important hurdles along the way.

4.4 Effects

By designing and implementing a treatment to the above-described problem, many of the desires of stakeholders can be improved or achieved. Developers and administrators will not have to delay certain updates until a maintenance window is planned and do not have to work at night on the weekend when fewer people are using the service. In most cases, a rollback is not immediately possible due to breaking changes and the need for downtime. A solution will also solve this by handling multiple schema versions with the same underlying data.

Costs associated with bringing a service or product down will be eliminated or significantly reduced. Additionally, customers might value the service or product higher which might increase revenue and profit margin. Organizations might also provide more guarantees to their customers by providing service level agreements when their service does not need to be brought down for maintenance. More of these effects can be found in Chapter 8 where a business case analysis is performed and use cases are given for which a tool would fit well.

Chapter 5

Treatment design

The problem has been investigated and it is now time to start designing the treatment. In this chapter, the design of the treatment on how that process went is written down.

5.1 Determining requirements

Since reinventing the wheel is not smart, using an existing solution and modifying it to accommodate all the requirements is better, especially when time is limited. Both in scientific literature and on the internet, multiple solutions have been identified that claim to solve zero-downtime schema updates. Additionally, multiple criteria were identified that solutions should comply with. To create a ranking among the solutions, all solutions have been checked to see if they fully comply with the criteria, partially comply, or do not comply. The results can be found in Tables 5.1, 5.2, 5.3, 5.4, 5.5. Results were split between criteria categories for better readability. Because some criteria can be deemed more important than others, a weight had to be determined for each criterion. To do this, a questionnaire had been made that experts had to fill in to see what the most important criteria were. Scores for the criteria were given on a range between one and five, where five indicates that the criterion is important and one indicates that the criterion is unnecessary.

Unfortunately, not many responses to the questionnaire were recorded. Most likely due to employees of ING not having the time or knowledge to properly fill in the questionnaire. It has been decided that these weights will still be used despite the low turn-out. The questionnaire did not include the two additional criteria. These criteria are, however, of such high importance that they will receive the maximum weight of five.

Points were awarded to the solutions based on how they ranked for each criterion. Compliance with a criterion was awarded one point times the weight for that criterion, non-compliance was awarded minus one point times the weight. For some solutions, it could not be determined if they complied with some criteria or the solution partly complied but not fully, for these criteria zero points were awarded. The weights from the questionnaire can be found at the bottom of the tables, these were rounded to the nearest integer.

5.2 Chosen tool

The scores of the solutions are found in Table 5.6. As can be seen, QuantumDB comes out on top. Other solutions score quite lower. This can be partly attributed to the availability and support criteria. Another reason for the lower score is that some solutions are not open source and compliance could

Solution	Expressivity	Declarativity	Composability	Concurrently active schemas	Non-invasiveness	Aborting
QuantumDB	No, complex transformations not supported	Yes	Yes	Yes	Yes	Yes
Ratchet	No, column constraints not supported	Yes	Yes	Yes, but only 2	Yes	Yes
SqlTable & Datatable in Terrier	Unsure, but probably yes	Yes	No	Yes	Yes	Yes
InVerDa	No, column constraints not supported	Yes	Yes	Yes	Yes	Yes
DECA	No, column constraints not supported	Yes	Yes	Yes, but only 2	Yes	Yes
PRISM(++)	Yes	Yes	Yes	Yes	Yes	Unsure
pt-online-schema-change	No	Yes	No	No	No	Yes
Google F1 (Spanner)	Unsure	Yes	Yes	Yes, but only 2	Unsure	Yes
Oracle package	No	No	No	No	Unsure	Unsure
Oracle Edition based	No	No	No	Yes	Unsure	Unsure
gh-ost	No	Yes	No	No	No	Yes
OSC	No	Yes	No	No	No	Yes
Weight	4	4	2	4	4	3

Table 5.1: Functional criteria

Solution	Transformation of data	Transactional guarantees	Application migration	Referential integrity	Schema isolation	Consecutive migrations
QuantumDB	Yes	Yes	Yes	Yes	Depends	No
Ratchet	Yes	Yes	Yes	Unsure	Depends	No
SqlTable & Datatable in Terrier	Yes	Yes	Yes	Yes	Yes	Unsure
InVerDa	Yes	Yes	Yes	Yes	Depends	Unsure
DECA	Yes	Yes	Yes	Yes	Depends	Unsure
PRISM(++)	Yes	Yes	Yes	Yes	Depends	Unsure
pt-online-schema-change	Depends	Yes	No/Depends	No	No	Unsure
Google F1 (Spanner)	Yes	Yes	Yes	Yes	Yes	Unsure
Oracle package	Yes	Unsure	Unsure	Unsure	Unsure	Unsure
Oracle Edition based	Yes	Unsure	Yes	Unsure	Unsure	Unsure
gh-ost	Depends	Yes	No/Depends	No	No	Unsure
OSC	Depends	Yes	No/Depends	No	No	Unsure
Weight	3	5	5	3	5	3

Table 5.2: Correctness criteria

Solution	Performance degradation	Space consumption	Finite completion time	Time to commit	Memory usage
QuantumDB	Depends	Depends	Yes	Yes	Unsure
Ratchet	Depends	Depends	Yes	Yes	Unsure
SqlTable & Datatable in Terrier	Depends	Yes	Yes	Depends	Unsure
InVerDa	Depends	Depends	Yes	Yes	Unsure
DECA	Small	Depends	Yes	Yes	Unsure
PRISM(++)	Depends	Yes	Yes	Depends	Unsure
pt-online-schema-change	Depends	Depends	Yes	Yes	Unsure
Google F1 (Spanner)	Yes	Yes	Yes	Yes	Unsure
Oracle package	Depends	Unsure	Yes	Yes	Unsure
Oracle Edition based	Depends	Depends	Yes	Yes	Unsure
gh-ost	Depends	Depends	Yes	Yes	Unsure
OSC	Depends	Depends	Yes	Yes	Unsure
Weight	3	3	5	4	3

Table 5.3: Performance criteria

Solution	Data recoverability	New data reverse transformation	Transformation resilience	Successfully online
QuantumDB	Yes	Yes	Yes	Yes
Ratchet	Yes	Yes	Unsure	Unsure
SqlTable & Datatable in Terrier	Yes	Unsure	Unsure	Unsure
InVerDa	Yes	Yes	Yes	Yes
DECA	Yes	Yes	Yes	Yes
PRISM(++)	Mixed	Yes	Probably no	Unsure
pt-online-schema-change	No	No	Yes	Yes
Google F1 (Spanner)	Unsure	Yes	Unsure	Unsure
Oracle package	Yes	Unsure	Yes	Unsure
Oracle Edition based	Yes	Yes	Yes	Unsure
gh-ost	No	No	Yes	Yes
OSC	No	No	Yes	Yes
Weight	4	3	5	5

Table 5.4: Tolerance of errors criteria

Solution	Availability	Support
QuantumDB	Yes, open source	Yes, developer still reachable
Ratchet	No	No
SqlTable & Datatable in Terrier	No	No
InVerDa	No	No
DECA	No	No
PRISM(++)	No	No
pt-online-schema-change	Yes, open source	Yes, community or paid
Google F1 (Spanner)	No	No
Oracle package	Yes, closed source	Yes, paid
Oracle Edition based	Yes, closed source	Yes, paid
gh-ost	Yes, open source	Yes, developers
OSC	Yes, open source	Limited
Weight	5	5

Table 5.5: Additional criteria

Solution	Total
QuantumDB	72
DECA	38
Oracle Edition based	36
InVerDa	35
Google F1 (Spanner)	32
PRISM(++)	28
SqlTable & DataTable in Terrier	26
Oracle package	20
Ratchet	19
pt-online-schema-change	17
gh-ost	17
OSC	17

Table 5.6: Solution score

not be determined from the articles in which they were described. DECA and InVerDa look like great alternative solutions if not for the fact that the original developers have no intention to disclose the source code of the solutions. Tools such as pt-online-schema-change, gh-ost, and OSC that promise online schema migration rank the lowest. This is caused by their extremely limited scope of what they can do, therefore not scoring high on many criteria.

5.3 Testing & fixing the chosen solution

To see in what state the solution is and if it can be used for real projects, it had to be tested first. As was already said in Chapter 3, there were no available benchmarks to test database schema migrations. The benchmark that Wevers et al. [50] specified is just a specification with no implementation. Due to time constraints, it was decided that it was not possible to implement such a benchmark, partly because of the huge scope that the benchmark covers. What was used to test the solution were some

transformations from the benchmark because of the ease of making a new TPCC-like database structure with HammerDB.

Initially, testing was done on manually made sets of changes (changesets), found in Appendix A that adapt the TPCC-like schema made with HammerDB version 4.2. Multiple errors stood in the way of successfully using the solution. Luckily, QuantumDB is open source and could be forked and modified. After some troubleshooting, the solution was able to run, albeit not without bugs. Before it could be fixed, the complete source code had to be read through and understood. Thanks to the original developers' effort to make the code-base as readable as possible, it was relatively easy to understand the flow of logic of QuantumDB. Multiple weeks were spent fixing java exceptions, bugs, and logic mistakes with the help of the original developer.

QuantumDB already had a multitude of unit tests using JUnit[1]. Some new tests were made and some existing ones were modified to incorporate new programming logic. Additional column types were introduced that PostgreSQL uses such as Numeric, Bytea (byte array), Serial (auto-incrementing), and more. Additionally, the logic to request the type of a column was not correctly implemented, causing a migrated table to not always have the exact same characteristics as the original table. Testing with a load on the database revealed another bug that had to be fixed. When a table is copied and kept in sync using triggers, QuantumDB made indexes non-concurrently. This caused the database to be locked for a short while. This was, however, easily fixed by adding the 'concurrently' keyword, introduced in PostgreSQL version 8.2, to the index creation SQL statement. Many more fixes were implemented but not discussed here as that would be quite boring and unnecessary, all commits and changes can be found in the closed pull request section on Github[2]. In total, 5.809 additional lines were added to the codebase and 1.737 lines were removed in 42 commits.

5.4 Panel discussion

Getting a clear overview of why a solution was never developed further and implemented is key to improving QuantumDB to be used in the future. To get multiple opinions from experts in the database field, a panel discussion was organized to gather opinions from various points of view. Five employees, of which one in a one-on-one meeting and the other four in a group meeting, from within ING were asked to participate based on their experience with the topic and possible influence on getting the tool to be used within ING. Two of these employees had read the paper by De Jong et al. about QuantumDB when it was released, the other employees got a quick overview and a demo of QuantumDB. After this demo, they were asked what key challenges they foresaw to QuantumDB getting implemented at ING to support zero-downtime database schema updates. The following are the identified key challenges that were discussed during these panel discussions.

5.5 Identified key challenges

5.5.1 The tool is still in alpha

A big challenge that was immediately recognized was that QuantumDB and the other solutions are still in alpha. Being in alpha means that the software is still subject to a lot of bugs and has not been tested extensively. To get out of this state and into beta, more features and use cases need to be extensively tested. A problem with this approach is that organizations do not want to start using or testing the tool

[1] https://junit.org/

[2] https://github.com/quantumdb/quantumdb/pulls

before it has reached at least beta, and can therefore be trusted not to majorly break the database. Organizations like ING want to be relatively sure of the software and tools they use as one misbehaving link in the chain can cause catastrophic damages. This will not be the case with tools that do not have an important function, but QuantumDB actually changes the structure of all underlying data of an application, making it a really important piece of software where errors and bugs are not welcome. This issue is part of a bigger problem, namely trust. If users and organizations do not trust QuantumDB, it will not be used.

5.5.2 The tool must be simple to use

A general remark that all panel members shared was that the tool should be simple to use and not require more than approximately 15 to 20 minutes of training on how to use it. If a tool or piece of software is widely used and already incorporated into a release pipeline, the amount of time it could take to learn to use the tool can be higher as the benefits of using the tool are already apparent. This is not the case when getting started with using the tool, if it takes too long to learn or start using it again after a while, it will not be used.

5.5.3 The mechanics of the tool must be relatively simple to understand

This is another issue related to the trust problem. Due to the importance of databases and how quickly it can go wrong if faulty SQL statements are supplied, the mechanics of the tool should be relatively simple to understand. The panel members agreed that if a database administrator or software developer that has to update the database schema does not know what the tool does or what the tool tries to accomplish, they will not trust the tool. Having a black-box solution that 'magically' transforms the database from one state to the next without being clear on what it does, is not something the person responsible for the database wants.

5.5.4 Most edge cases must be supported

One of the panel members said that if less than 95% of the edge cases are supported it will not be used within ING. If organizations start using the tool and they come to the conclusion that what they want to achieve is not possible they will have to build workarounds or stop using the tool. A difficulty for the development of the tool is to know in what environments it is going to be used. An example that was brought up, is a database that is being populated every hour with massive amounts of data, and subsequently, most of it is deleted by one delete statement. The question is if the tool does not degrade the performance so much that it will not be usable. Also, databases that use rarely-used features are a problem as implementation and testing of those features takes a lot of time.

5.5.5 Pipeline integration must be supported

One tip that was given for easier adoption of the tool is to support pipeline integration. If developers can quickly set up the tool within the deployment pipeline, it will not be such a hassle to start and keep using the tool. Any manual work will be limited to the things QuantumDB cannot do automatically. ING is currently using Azure DevOps which provides the deployment pipeline. Azure DevOps is really flexible and can accommodate almost any step that can be thought of.

5.5.6 Benefits are badly quantifiable

A view that did not come from the panel members but was agreed upon is that the benefits of a solution are not really known. This is most likely due to the panel members having a technical background and are generally not concerned with the price tag or potential benefits of technology on a larger scale. If it can be shown that a solution will either save or earn more money, executive managers at ING might be willing to invest in implementing a solution.

5.6 Resolutions to the identified key challenges

To solve the key challenges listed above, a plan of approach was made. Some solutions were implemented while others would take too much time. The solutions that would take too much time are found in the future work chapter at the end of this thesis.

The first challenge of the tool being still in alpha can be solved by releasing a new version of QuantumDB and call it beta. However, this will not solve the myriad of bugs or errors that possibly still plague the tool. Luckily during testing of QuantumDB, a lot of bugs and errors were found and fixed, some more tests were written, and no further major problems were found. To know in what stage of the software life cycle QuantumDB currently is in is best known by the original developer. To support the transition to beta, tests and validation need to show that everything works as expected. Unit tests were already included and passed successfully. The next step is to validate QuantumDB with real use cases. In the validation section of the methodology, this will be further specified.

The second identified key challenge is not specific to QuantumDB itself but in general. Users can interact with QuantumDB via the terminal with several commands. These commands include: 'init', 'changelog', 'fork', 'drop', 'status', 'query' & the newly implemented 'cleanup' (more on this one later). Additionally, these commands also include some parameters that can be set. All of these commands with parameters can be found in the documentation[3] where a clear description is given on what it does and what parameters can be set. Care has to be given that no feature creep happens and that all commands speak for themselves. Some features do not necessarily need a separate command, instead, a parameter or flag to an existing command would be better.

To make the solution easier to use, a new feature was implemented such that changesets are not selected by a random hash anymore but by the changeset id that the user can specify. This will make using QuantumDB easier as it is known beforehand which changeset can be selected with which name. Another positive aspect is that it will be easier to support in a deployment pipeline as there will be no randomly generated hash that first has to be extracted before a migration can happen. To make QuantumDB more usable, a graphical user interface can be implemented next to the command line interface. However, this might only help small developers as large organizations want to make it part of the deployment pipeline. Overall, QuantumDB is rather easy to start and keep using, however, research on process complexity should happen that compares current best practices and QuantumDB side-to-side to determine if QuantumDB is actually easier.

As for the third identified key challenge, a complex problem is mostly tackled with a complex process. QuantumDB does not necessarily work in a complex way but if that process is not shown, also known as a black box, it might as well be. Luckily, the process can be made more insightful and transparent. To accommodate that, a new feature was implemented that prints the generated SQL code to the console or a file. Before migration, the user can inspect the generated code and evaluate if everything works as it should. It is not sufficient to execute these printed SQL statements yourself as some metadata SQL statements are left out to make things easier to read. QuantumDB additionally migrates data in

[3] `https://quantumdb.io/docs/master/#cli`

batches of 2.000 rows, the SQL statements to achieve that are not printed to the console as it first has to request the number of rows in the database and iteratively migrate the next 2.000 rows. A different implementation is needed to support manually executing the code. Additionally, supporting lesser-used functions and features of PostgreSQL might make some processes more complex. Care has to be used when implementing these features that normal usage is not being over complicated.

The remark that edge cases have to be mostly supported is a valid opinion. However, it does not mean that QuantumDB has to support almost all features of PostgreSQL as most features are not used nowadays. It can be seen in Appendix B Table B.1 that within ING projects only basic operations are used. Operations such as createTrigger, createFunction, or createProcedure are not used, which indicates that databases are used to only store data and do not execute any database-side logic. Such databases are so-called ”dumb”, supporting those features is not on top of the priority list if they are not (widely) used. Support for all column types can be implemented, right now a sub-set is implemented with the most used types. Types such as CIDR (IP addresses) or geometric shapes are PostgreSQL specific and are not seen in any ING projects.

Some features to support pipeline integration are already implemented, such as the changeset id being the one to use with commands instead of a randomly generated hash. Additional support for pipeline integration can be implemented by using determined hashes instead of randomly generated ones for table and other database structure names such that it will always be the same in both development as production environments as this helps in knowing what names are going to be generated beforehand.

The last identified key challenge is not related to a solution to solve zero downtime database schema migrations but is related to the business side. In Chapter 8, a business case will be given to show that investments of time and money into a solution are worth it. The business case is made in the hope that organizations see the positive aspect of a solution to zero-downtime database schema migration and want to invest in it.

5.7 Additionally implemented features

As was already shortly mentioned above, a couple of new features were implemented to solve some of the identified key challenges. These features should improve the usability of QuantumDB and cause organizations to more easily start using it. One of these features is the ’dry-run’ parameter for the ’fork’, ’drop’, and ’cleanup’ commands. This feature will let database administrators first see what SQL statements get generated before trusting QuantumDB to perform the migration. It is not possible to manually execute these SQL statements, as some statements to migrate data and insert metadata are left out. In the future, new migration logic can be made to allow for manual execution by database administrators. This will increase the level of trust, as you can exactly see what logic gets executed on the DBMS.

The second implemented feature is the ’cleanup’ command. This command will rename all tables back to their original name, after which a client can access the database normally, without the query rewriter. This command is important to let organizations try using QuantumDB while having a way to remove QuantumDB if it turns out to not add the expected benefit to the project. This command will not completely remove QuantumDB from the moment it is called. It will first enter into mixed-state where the last version is the one with all tables in their correct form. The database administrator can drop the oldest version after which clients can access the database normally. Dropping the ’quantumdb’ schema, where all metadata is saved, will completely remove all traces of QuantumDB from the database.

Chapter 6

Treatment validation

During the design stage various scenarios with manually made changesets were tested and QuantumDB seemed to work as expected. The next stage in the design cycle as described by Roel J. Wieringa [3], the time for validation has arrived. In this chapter, QuantumDB is validated on various levels to ensure a correctly working software product. Stages include testing under load to ensure the zero-downtime requirement, consistency testing on a real ING project, and (java) unit testing. Any problems that were found during validation testing were fixed except where they could not reasonably be solved. These problems have been listed in the last section of this chapter. Validation experiments were re-run when modifications were made to the code, even when they should not have impacted anything in the experiment.

6.1 Load testing

One of the key features of QuantumDB is that it can perform migrations while database clients still interact with the database. Proper validation of this feature is a must. Unfortunately, no standard benchmarks are (yet) available to test QuantumDB against. Getting a precise value for the performance drop that QuantumDB causes is also not possible as all server equipment, database structures, and client loads are different. This experiment will validate that QuantumDB does not block client transactions and will get a rough figure on the performance drop.

To test migrating during load, HammerDB version 4.2 was used. HammerDB is an implementation of the TPC-C specification[1]. It is not the most ideal benchmarking tool out there, but it is rather simple to use. A couple of things to note with the specific implementation of HammerDB is that it does not create and comply with foreign key constraints. When these foreign key constraints are made, roughly 75% of the transactions do a rollback due to the load function generating invalid data points. Another problem is that one table does not contain a primary key constraint. Why these points are problematic, will be explained at the end of this chapter.

To solve some more problems with how HammerDB constructs the schema, all of the nine tables were modified to include an extra auto-incrementing column as the primary key column due to the migration function not performing as expected when a composite primary key was used. This also solved the lack of a primary key constraint of one of the tables. HammerDB ran fine with these changes and negligible performance loss was expected from these changes.

HammerDB can be configured for multiple workloads to be used on home or enterprise systems. To generate load, virtual users can be created to query the database. To test various sizes of data in

[1] http://www.tpc.org/tpcc/

databases, HammerDB has an option to create a database with a configurable number of warehouses. One warehouse equals a fixed number of rows in each table associated with that warehouse. The number of rows per warehouse can be found in Table 6.1. To reduce the chance of virtual users requesting or working on the same data, a minimum of 4:1 ratio of users to warehouses is recommended[2]. Since these virtual users take up computer resources, there is a limit to how many a computer can have running. HammerDB has built-in functionality to also use other computers as secondaries to increase the number of virtual users querying the database. It was decided that it was not needed as the server on which the database runs is not that powerful and will not saturate the resources of a personal computer with virtual users.

The server on which the PostgreSQL instance ran was a virtual machine running Debian 10.9 with 2 virtual CPUs, each having a dedicated physical thread. The server had 8 GB of RAM allocated. The version of PostgreSQL was 11.12, which was the default packaged PostgreSQL version with Debian at the time of writing this thesis. During development op QuantumDB, PostgreSQL version 13.3 was used, but no differences were identified that could cause a problem.

The server on which the load testing scenario was tested was a Virtual Private Server with 2 cores, 8 GB ram, and 100 GB NVMe SSD storage. Various configurations of virtual users were created on a remote desktop to simulate multiple connections to the database and to simulate transactions happening. These transactions were performed using stored procedures to lessen the load on the client's computer. It does, however, increase resource usage on the server. If a database is only used as a data store instead of actually performing calculations on incoming data, more transactions can be performed in any allotted time.

In Table 6.3 & 6.2, the various configurations can be found with which QuantumDB was tested. The changesets that were used in the migration can be found in Appendix C. Unfortunately, HammerDB cannot be run on the new version of the schema. To make it work, additional features like the migration of stored procedures should be implemented in QuantumDB. To keep all tests the same, HammerDB was started and allowed to stabilize, after which the migration started. The transactions per minute and new orders per minute (NOPM) were recorded during the process, as well as the times at which point the migration started and finished. The Python script to request the TPM and NOPM can also be found in Appendix C.

QuantumDB uses 1 connection for the migration process. The migration function also executes sequentially, this means that PostgreSQL at most uses 1 core of the host machine for the migration. A simple calculation can be made that QuantumDB reduces the performance of the database server by roughly 1 / # of cores. However, this is not exactly true as QuantumDB has some wait time and other checks to perform on the client computer per batch of data. QuantumDB also blocks certain rows during migrating which will reduce the performance of any other clients currently working with that data. An experiment was designed to test the relative performance increase of increasing the number of CPU cores from 2 to 4, but it was not the bottleneck of the migration process as CPU utilization reached a maximum of 75% throughout the experiments. The bottleneck was the 8GB of RAM which was fully utilized during the migration process. Increasing this was not possible without increasing the budget for the VPS which already became too expensive.

The NOPM values for the migrations are shown in Figure 6.1, 6.2 & 6.3. Due to HammerDB creating a lot of new data, also the number of rows in the database after the migration was recorded. These values can be found at the bottom of Table 6.1. For the 1 warehouse with 1 virtual user configuration, the end amount of rows in the largest table, order_line, was 444.962. For the 10 warehouses and 2 virtual user configuration, the number of rows was 3.276.431. For the 100 warehouses with 10 virtual user configuration, the number of rows was 36.545.726.

[2] https://www.hammerdb.com/docs/ch03s07.html

	1 Warehouse	10 Warehouses	100 Warehouses
Customer	30.000	300.000	3.000.000
District	10	100	1000
History	30.000	300.000	3.000.000
Item	100.000	100.000	100.000
New_order	9.000	90.000	900.000
Order_line	300.000	3.000.000	30.000.000
Orders	30.000	300.000	3.000.000
Stock	100.000	1.000.000	10.000.000
Warehouse	1	10	100
Total	599.011	5.090.110	50.001.100
Order_line (after)	444.962	3.276.431	36.545.726
Virtual users	1	2	10

Table 6.1: Amount of rows generated for various configurations of warehouses in HammerDB 4.2

	1 warehouse (2.000 batch) modified	10 warehouses (2.000 batch) modified	100 warehouses (2.000 batch) modified	100 warehouses (100.000 batch) modified	100 warehouses (100.000 batch) unmodified
Customer	1,64 (0,03)	15,19 (0,25)	158,05 (2,63)	68,19 (1,14)	309 (5)
District	0,01 (0)	0,01 (0)	0,03 (0)	0,03 (0)	0,04 (0)
History	1,26 (0,02)	12,43 (0,21)	124,16 (2,07)	39,85 (0,66)	–
Item	3,9 (0,07)	4,08 (0,07)	3,88 (0,06)	1,13 (0,02)	1 (0)
New_order	0,32 (0,01)	3,51 (0,06)	36,81 (0,61)	8,7 (0,15)	12,2 (0)
Order_line	12,38 (0,21)	126,96 (2,12)	1267,68 (21,13)	398,47 (6,64)	8642,2 (144)
Orders	1,24 (0,02)	12,32 (0,21)	123,47 (2,06)	37,24 (0,62)	57,4 (1)
Stock	4,75 (0,08)	49,33 (0,82)	481,17 (8,02)	207,6 (3,46)	619,1 (10)
Warehouse	0,01 (0)	0,01 (0)	0,01 (0)	0,02 (0)	0,01 (0)
Total	25,51 (0,43)	223,84 (3,73)	2195,26 (36,59)	761,22 (12,69)	9640,95 (161)

Table 6.2: QuantumDB migration time without load in seconds (minutes)

The graphs show that at all times the transactions per minute never dropped to zero for the scenario where QuantumDB is used to migrate to a new schema version. Tables 6.2 & 6.3 show that it takes quite a lot longer to migrate data to a new version when there is more data. This is expected but is worrying for companies when QuantumDB is to be used on data sets containing orders of magnitude more data than is used in this experiment. Increasing the batch size is also not really doable as more data is blocked for longer. This even caused deadlocks in the experiment with a batch size of 100.000 with 100 warehouses. It has not been proven that it cannot happen when the batch size is set to the default of 2.000, though it will be highly unlikely as it only blocks certain rows for a couple of milliseconds and has not happened during testing.

The green line in the graphs are the NOPM when the migration is performed using normal SQL where no thought is giving into making it non-blocking. It can be seen that as soon as the code is executed, the line drops to 0. In the 1 warehouse configuration it did not reach 0 as it took less than 1 second to complete and for the NOPM to go back up again, however it did block the database access for some time. In the 10 warehouse configuration the block time was a bit longer due to the larger volume of data contained in the tables, this is also true for the 100 warehouse configuration. The block times were respectively 7 seconds and 39 seconds. This means that the database would have been unresponsive for at least those seconds when the migration is happening. This is unacceptable when a service level agreement is made promising 99,99% up time.

As for the performance drop that is shown at the end of the graphs, not much can be seen of it. Calculating from the end of the migrations, a 14%, 12%, and 10% decrease in NOPM can be calculated for respectively the 1, 10, and 100 warehouse configurations. This is reasonable as it is not the point to keep 2 versions active at all times. However, if it is needed, it can be done without too much performance

	1 warehouse (2.000 batch) modified	10 warehouses (2.000 batch) modified	100 warehouses (2.000 batch) modified	100 warehouses (100.000 batch) modified
Customer	1,47 (0,02)	17,12 (0,29)	206,78 (3,45)	
District	0,01 (0)	0,01 (0)	0,09 (0)	
History	1,49 (0,02)	13,84 (0,23)	153,54 (2,56)	
Item	3,98 (0,07)	3,84 (0,06)	4,8 (0,08)	
New_order	0,34 (0,01)	3,54 (0,06)	40,61 (0,68)	Deadlock
Order_line	15,77 (0,26)	140,07 (2,33)	1751,45 (29,19)	
Orders	1,61 (0,03)	13,49 (0,22)	154,34 (2,57)	
Stock	4,77 (0,08)	49,28 (0,82)	702,16 (11,7)	
Warehouse	0,01 (0)	0,01 (0)	0,02 (0)	
Total	29,45 (0,49)	241,2 (4,02)	3013,79 (50,23)	

Table 6.3: QuantumDB migration time with load in seconds (minutes)

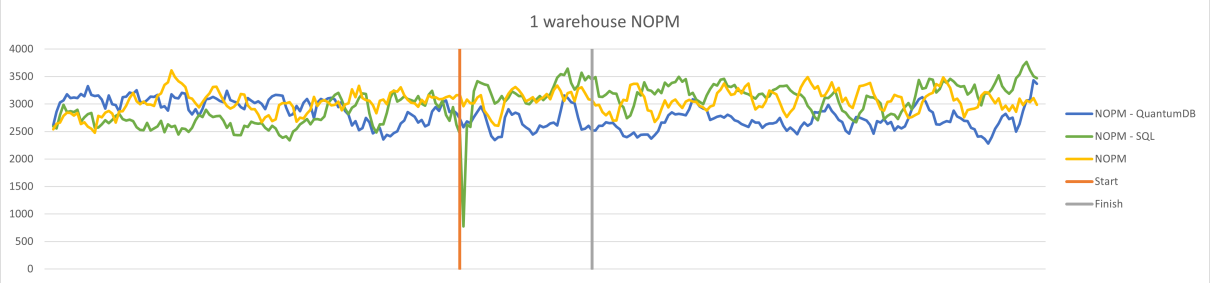


Figure 6.1: New orders per minute with migration using QuantumDB (blue), migration using normal SQL (green) and without any migration (yellow) with 1 warehouse and 1 virtual user

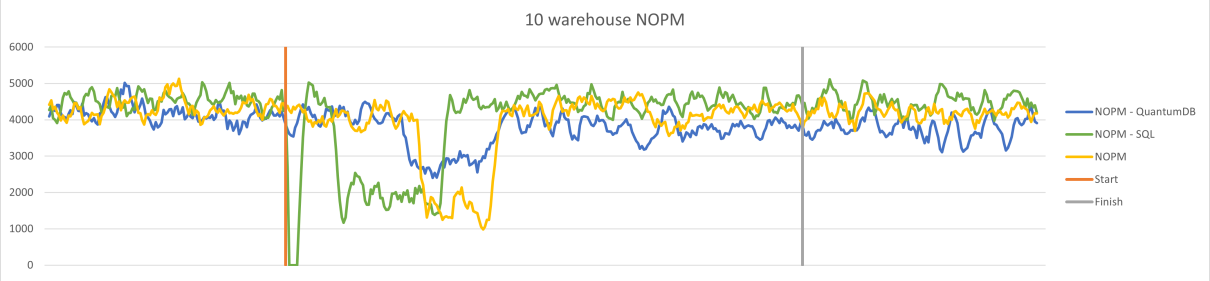


Figure 6.2: New orders per minute with migration using QuantumDB (blue), migration using normal SQL (green) and without any migration (yellow) with 10 warehouses and 2 virtual user

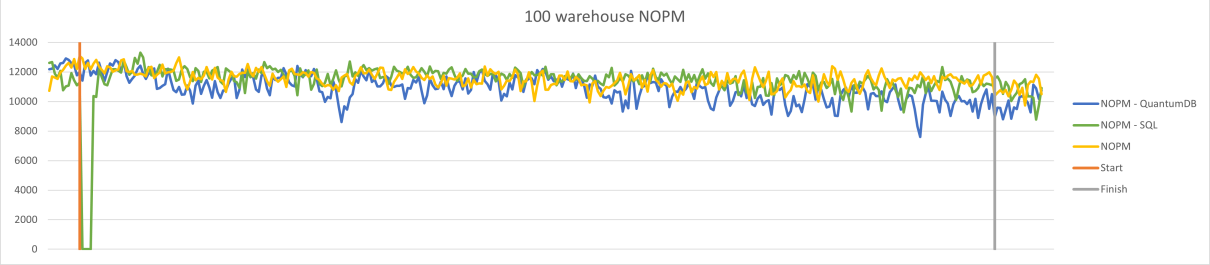


Figure 6.3: New orders per minute with migration using QuantumDB (blue), migration using normal SQL (green) and without any migration (yellow) with 100 warehouses and 10 virtual user

difference. A third active version was not tested as the synchronization between the oldest version and the newest version is not yet fully tested and does not work as expected. Additionally, the graphs show the performance decrease when clients utilize the resources to the maximum. When there is no high load on the server, a slightly lesser performance decrease can be expected.

It is also shown that all rows of the ghost tables contain the same data (except for the changed structure parts of course). Due to the triggers still happening inside a transaction, a client connecting to one version of a schema cannot see data any earlier than another client connecting to another version. This is a requirement as DBMS, like PostgreSQL, follow the ACID principles which state that all operations are atomic and consistent.

The performance drops found in Graph 6.2 are not easily explained as it happened in all runs that were done. It might have something to do with the 5:1 ratio of warehouses to virtual users instead of the 10:1 with the other configuration. It might also be the point at which the RAM was saturated and it had to switch to a different garbage collection process, therefore, limiting the NOPM for roughly half a minute. However, it did not happen at the 100 warehouses experiment which might indicate a bug in HammerDB itself.

6.2 Consistency testing

Load testing through HammerDB has shown that clients connected to the original schema version can continue querying the database and can continue to access the data contained in the database. Via queries from the query rewrite, it has been shown that all data is also available in the new version. To validate if QuantumDB can stay consistent with schema operations, a changelog from an ING project has been chosen to be redone. Unfortunately, this changelog cannot be made public and will not be found in the appendix.

To ensure that the schema updates were completed, dummy data was inserted at every step, as well as some update and delete statements were executed. It was then checked if all data acted according to what the new schema should have done. This process was a time-consuming part as the existing Liquibase changesets had to be changed as the syntax of QuantumDB is a little bit different from that of Liquibase. Additionally, these changes were made for an Oracle database, while QuantumDB, right now, only works with PostgreSQL. These changes translate to the renaming of column types and different ways of implementing auto-incremented columns.

In the end, a manual inspection was performed to ensure the database is the same as what the changesets specified. A couple of bugs were fixed along the way but in the end, the schema looked exactly like it should. One of the problems was the creation of a new primary key index every time a migration for a table happened. This was fixed by modifying which indexes to copy and which to let the database make automatically.

A script has been created to be run in the future to test a real project on future versions of QuantumDB, at the end, it will check if the data it returns is the same as what has been returned right now. If any of the migrations that QuantumDB performed change in a future date, a warning will pop up telling the user that functionality has changed and that something might be different. The operations in this changeset are listed in Appendix C, it does not include all operations possible and does not test all available variations of input variables. It is however a real project, which is a better validation than many synthetically made changesets.

During the migration, the code coverage tool JoCoCo was used to determine the code coverage that was achieved during this experiment. The results can be seen in Appendix A, Table A.1. The results show that for most classes reasonable coverage was achieved. As with the other experiments the Driver package is not covered which bring the average down. The cli.xml and core.schema package also shows

is low code coverage, which is expected as the changeset did not use all operations and therefore did not execute those parts of the codebase.

6.3 Unit testing

Due to the ambition of the original developer of QuantumDB to make a well-tested, clearly understandable software application, unit tests were made to test the separate components of QuantumDB. Unit tests are made next to the source code of the project and are intended to test small pieces of the software to ensure their functionality is as what it is intended to do. To ensure no large portion of code is missed in the unit tests, code coverage by JaCoCo is used to determine the areas where the unit tests do not reach. The results can be seen in Appendix A, Table A.3.

Unit tests are not the holy grail of validating code as developers cannot envision all states in which the program is to be executed. An example would be an if-statement with multiple components where only one part is checked. Code coverage would say that the line of code is executed and a developer might think that most cases are covered, while in truth more possibilities could be tested.

Unit tests are great for making sure future versions of the software do not unknowingly change the intended functionality of a function and therefore the result of the program. However, developers can make a mistake in expecting what a certain input should return for that function. What can also happen is that for the tested input a certain output is returned that is correct while a different input for that function will return an incorrect output.

6.3.1 Components of QuantumDB

QuantumDB consists of multiple components that all perform their own task. QuantumDB consists of the following packages: core, cli, postgresql, driver, and the query-rewriter package. Currently, only PostgreSQL is supported but more packages can be made to support the other relational database management systems, although, they might need a vendor-specific implementation to achieve the zero-downtime requirement as it currently uses after triggers. The driver package does not contain any unit tests and has not been tested during the experiments. The cli package originally did not contain any unit tests but these were created to also test the functionality of the cli package. During the design stage, multiple bugs were found in the cli package that could have been prevented if unit tests were made earlier.

Core package

The core package contains the main logic and ontology classes that represent the database structure and provide the zero-downtime schema migration used to power QuantumDB. The tests in this package therefore also test if this functionality is correct and will provide correct error messages when an action is not valid. Two examples of such tests are checking if a table can be made with a NULL name and an empty string as a name, in both cases the program should return an illegal argument exception because this is not possible according to the SQL specification. This package contains a total of 233 unit tests, which all pass. Manual observation shows that not all methods and lines are covered. Specifically, code to migrate views, indexes, sequences, and identities are not covered. This could become a problem when QuantumDB is going to be used for such operations.

Cli package

The cli package contains the classes which let users perform migrations using commands on the command line interface like how Liquibase works. This package also contains the necessary classes to process the XML code in the changesets. It originally did not contain any tests, but it was needed as multiple bugs were found during testing. Unit tests were made for all operations and included multiple wrong inputs to see if QuantumDB acts accordingly. Due to failing tests, some functionality was implemented that will warn users when the wrong syntax is provided in the XML. Previously, QuantumDB would just continue working like nothing was wrong and return an SQL error later when it tried to perform a transformation but did not include the malformed syntax. By implementing this functionality, a malformed syntax is found earlier and the user will be warned before attempting to perform wrong transformations. A total of 22 tests were made for all transformations currently supported and some tests for malformed syntax.

PostgreSQL package

Together with the core package, the PostgreSQL package is the most important one as it tests the specific implementation of PostgreSQL. The unit tests in this package also test their functionality on a real database. To set up, it needs a database development environment where databases can be created. Most of the tests use the same test scenario of a video store back-end where multiple tests are performed. Due to the implementation of additional column types, new tests had to be made to ensure tables were made using the correct types and that data would be in the same format when it would be synced across multiple tables.

Existing unit tests were only testing small changesets at a time. No multiple changeset migrations were tested, as were dropping multiple changesets at once. Unit tests for testing mixed-state of more than 2 versions are not implemented.

Query rewriter package

The query rewriter package includes a couple of unit tests, but not many. This is, in part, because of the limited functionality that it provides. However, for proper usage of QuantumDB, this package needs to be fully tested, as a single mistake can cause unexpected malfunctions in production.

6.4 Known limitations

Databases can be used for a lot of tasks, it is therefore not doable to support and test all edge cases. While performing the validation experiments, a list has been made of what QuantumDB does not currently support. In theory, all features and edge cases can be supported, but the time required to make it work will not make sense for most of them. As was already seen in the previous section, within ING not many, if at all, advanced features of database management systems are used. Many of ING's applications use the database to only store data conform to a schema with constraints. These applications handle any logic on the application level and do not use the database level functions, procedures, triggers, or any other specialized features.

QuantumDB does not currently support the migration of functions, stored procedures, and triggers. Migration with 'instead of' and 'after' triggers do not work at all, the 'before' triggers do work on the initial schema version they were defined on and will propagate their changes to the new schema version, however, the trigger will not be copied and any clients connecting to the new schema version will not

execute the 'before' trigger. Manually making a new 'before' trigger can be done and should work, but this is not tested.

Functions and stored procedures can be supported, but QuantumDB will need a similar mechanism as implemented with the query rewriter, substituting any mentions to tables, and including, altering or removing modified columns. An additional inconvenience is that changing any logic will be difficult as the previous version has its own functions to change the data according to its own function definition. It will be much easier to have all logic within the application instead of the database where the blue-green updating scheme can be done much easier.

Unfortunately, ING did not have any additional projects that did not require massive time investments to translate between SQL or Liquibase and QuantumDB syntax. Most of these changesets use Liquibase as the executor of SQL statements and do not use the Liquibase syntax. QuantumDB cannot work with such statements to provide zero-downtime migrations. A couple of open source projects have been found using the advanced search feature of GitHub, searching for Liquibase changelogs. However, these changelogs were either too small, in YAML syntax (instead of the supported XML), also used Liquibase to run SQL queries, or used Liquibase in other ways that QuantumDB cannot be used for. This will cause a problem when developers want to redo a project in QuantumDB as most existing syntax is not exactly compatible. However, this is not a problem when developers create new changesets and keep the differences and limitations in mind. Efforts can be made to make QuantumDB more compatible with existing syntax and changelogs used in projects but that would require considerable development time.

6.5 (inherent) Problems

During the validation stage, multiple problems were identified that can not be solved with some additional programming. These problems happen due to the mechanisms QuantumDB operates with. QuantumDB does not warn the user when trying to perform a migration when one of the below-mentioned problems is present. This would be a nice feature to have as it could potentially make the migration process malfunction and cause the existing application to behave differently.

6.5.1 Table without primary key

QuantumDB migrates data from one table into the next using a migration function. It does this in batches with a default of 2000 rows where it picks the next 2000 rows starting at the primary key of the last migrated row in ascending order. It will continue until all data is migrated. This migration function does not work if there is no primary key. This can not be easily solved as some solutions cause other malfunctions. One of those solutions is to not migrate data in batches of 2000 rows each time. However, this will be the same as blocking the whole table for as long as it takes to migrate all data. Another possible solution is to use a unique index, however, this will mostly only exist on tables with a primary key. A problem is also that it should not allow NULL values, as the migration function can get into an infinite loop if there are more than 2000 rows with the same 'unique' key.

If functions or stored procedures in PostgreSQL can begin and end transactions within another transaction a different migration function can be made that will loop over all data in a single call of the migration function but other side effects start happening at that point. To conclude, zero-downtime migration without a unique not null index is not feasible.

6.5.2 Constraint addition

Certain technicalities look like a problem until a closer look is given to them. One of these is what happens when a foreign key, unique or other constraint is added in the next version of a schema. Suddenly, data that is migrated and synced, has to comply with this new constraint. And what happens when it does not comply? The database rolls the whole transaction back. The new constraint is now also in force on the old version of the schema. It can be said that this would be a problem, as you are changing how the schema works after it has been defined. But a closer look at the schema update reveals that when such a constraint normally gets added to the schema, the original data must already be able to comply with this requirement, otherwise the schema transformation will not succeed and is rolled back.

It will only be a problem if database administrators expect that the original schema will not change and have not tested if data will comply with the newly introduced constraints. Also, if an application was responsible for keeping the constraints in check, it might not do it within one transaction. It might follow the NoSQL consistency variant, eventual consistency. In this case, when the application gets shut down for maintenance, the database would look like it was always consistent even though it was not during operation. A solution for this problem might be to assign a function to the column in case of non-compliance to the new constraint on how to transform it to a compliant one. However, in this case, both versions would not get the same data or the original version would not insert or update the specified data. A warning within QuantumDB to prevent users from accidentally causing unexpected system behavior would be something nice to have but is not implemented right now.

6.5.3 Column ordering

QuantumDB does not depend on fate in assuming PostgreSQL always returns tables with the same column ordering. While it has yet to happen that a column was in another position during validation, it is not guaranteed by PostgreSQL. It is not so much a problem as a general awareness that developers have to be conscious about when interacting with the database, that they always explicitly specify which column they mean instead of relying on the not guaranteed ordering of columns. Oracle DB does guarantee the column order in their databases.

An exception to the above statement is the column ordering of indexes. When developers use the create or drop command for indexes, it is important to keep in mind the differences when placing one column in front of the other. To remove an index with QuantumDB, a developer has to specifically write the index with the right columns down. A slightly better way would be to remove indexes by name, but due to index names being universally unique across the whole database this would not work when the table is migrated more than once, as QuantumDB would make another name.

Chapter 7

Discussion

During the design and validation steps, data was gathered in various forms. To get information out of this data it has to be transformed in a certain form. Luckily, when collecting this data, care was given into already transforming the data into a correct form for usage. Using this information, new knowledge can be gained by applying algorithms and visualizations and closely looking at possible patterns. One step further in the DIKW hierarchy (Data, Information, Knowledge, Wisdom) is applying this knowledge to future cases to make the best possible decisions, which is called wisdom.

In this chapter, results from the previous chapters are discussed and interpreted. The implications that those results have are also discussed as well as the limitations of the research that was done. At the end of the chapter, future work and recommendations for further research are given in areas that have not been investigated due to time constraints.

7.1 Interpretations & implications

What can be interpreted from the information is that QuantumDB works for the specified operations and that it will migrate to a new schema version without downtime in the sense that clients can keep on querying the database without receiving an error or timing out. During the migration process, batches of 2.000 rows are migrated at a time. During testing, this took roughly 40 milliseconds per batch on the hardware that was used. QuantumDB, however, only reads the rows and should therefore not be blocking any other transactions on those rows of data. An experiment with a batch size of 100.000 rows resulted in deadlocks, possibly due to write locks on the new table that data is migrated to, but more research needs to happen to determine the exact cause. A batch size of 100.000 took roughly 2.500 milliseconds during the experiment, which might have been too long for a read lock.

As was mentioned in Chapter 6, the performance drop calculated was between 10% and 14%. These performance drop numbers were noticed for all experiments with 1, 10 and 100 warehouses. This indicates that a performance drop while migrating happens separate from the amount of queries the database receives. This means that it can even be used on systems that do not have much headroom available. Latency and throughput however, will drop by roughly 10% to 14%. It is noteworthy to mention that performance drop decreases as more users query the database and the database contains more data. Although this is only seen for one experiment, results might vary with other database layouts and changelogs.

Due to the unmodified schema performing way worse, QuantumDB cannot properly work with composite primary key constraints. This means that existing databases that use such constraints have to be adapted. Doing this is not difficult but the schema will look a bit different than before. Database administrators have to drop the primary key constraint because only one may be active at a time. Next,

they have to add an extra primary key column with auto-increment on. In PostgreSQL, this is as simple as adding a column using the serial type. At last, they have to add a unique not null index on the original composite primary key constraint columns. Doing this ensures that QuantumDB can work properly while still having the primary key constraints on the original columns, albeit in a different syntax.

It can be seen that migrations will take longer the more data is in the database. This would make migrating a schema in a huge database multiple times a day not doable. However, this would probably not be the case. At the start of the development of a software product, many updates can be made each day. At this time, not a lot of data would probably be in the database. Migrating to a new schema version would therefore not take long. At the end of the development process, fewer updates come out which are probably bundled, because developers do not want to bother end-users by updating multiple times each day. Longer migrations are therefore justified.

The above also implicates that QuantumDB is less useful when used in projects that have a large database, probably from data in earlier applications, that also want to release several times each day. It can be done if schema updates are not part of many updates or if developers plan schema updates or group them. Even though QuantumDB promises to let developers perform updates whenever they want to push a new version, the time-to-market of these updates drops considerably in this case.

7.2 Limitations & future work

It would have been nice to provide a solution to all challenges described in this master thesis, but all research has to end somewhere as it cannot go on indefinitely. In this section, all of these limitations, questions, and challenges are discussed that are still open for future research. Some of these listed might depend on other research in the list that has to be done first.

7.2.1 Limitations

QuantumDB was only tested using the default configuration of PostgreSQL, this includes the default value of the isolation level. This level defaults to Read Committed, which is the second level in the SQL standard, but the first in PostgreSQL as Read Uncommitted performs the same due to the MVCC design. Results might change when the isolation level is increased, in particular, the performance drop might be more severe and a batch of 2.000 might even cause deadlocks due to stricter rules.

Performance drop numbers were collected using only one experiment, so different workloads, and schema layouts can probably expect different performance drops, the database developers and administrators using QuantumDB should first test it with their specific instance before relying on these numbers to be true. The performance drop numbers were calculated when all tables in the schema got an update. Most updates only perform updates to one or a couple of tables, performance drop numbers can in that case be lower.

7.2.2 More testing & start implementing into the pipeline

More additional work can be done on zero-downtime schema migrations and QuantumDB. The next logical step is to actually implement QuantumDB into a release pipeline, but before that could happen more testing is required to solve the problem of the tool still being in alpha. To solve this, future research could look at testing more changesets as well as testing changesets with additional complexity. Future research could also look into running two separate database instances, one with and one without QuantumDB, next to each other to see potential differences that occur.

7.2.3 Implementing additional features

Future research could also look at implementing additional features that could be migrated. Right now, triggers, functions, and procedures are not supported as they require a different approach to migrate without downtime and without (much) human intervention. Migrating these features is not as easy as just rewriting the different tables and columns inside the code as calculations can become quite different when columns get changed, added, or deleted. Before this research actually gets started, however, it first has to be investigated how many database instances still use these features. One of the experts in the panel discussion mentioned that databases were getting used more as dumb data stores and that developers decreasingly used these additional databases features. It would be a waste of time to implement these features if they would not be used.

7.2.4 Mixed-state with more than 2 active versions

Another feature that was not tested was the ability to achieve mixed-state with more than 2 active versions. Future research could look at fixing and start testing this functionality. A different way it could be used is to achieve A/B testing, where two different versions spawn from a baseline version to see which one performs the best, keeping in mind that data between all versions stays the same.

7.2.5 Driver testing

One of the features of QuantumDB, the driver that rewrites queries, was not tested during this research. In part because of the lack of real applications. The driver package is based on the Java Database Connectivity (JDBC) driver that handles the connection between the application and the database. The codebase consists of some specific implementation that lets applications connect to a specific version and rewrites the query, but most other code is just the same as the normal JDBC driver.

7.2.6 Additional DBMS support

Right now, QuantumDB only supports PostgreSQL. This can be extended to also include DBMS like Oracle DB, Microsoft SQL Server, and MySQL. These will require their own specific implementation as these DBMS act a little differently. However, most code can be copied and the specific parts can be changed to accommodate the differences. One important part that might require special attention is the integrated function pg_trigger_depth() that QuantumDB uses for the migration in PostgreSQL. Other DBMS might not support such a feature, which will prove troublesome as a different way to migrate data has to be found.

7.2.7 Usability testing (complexity)

A solution would not be a solution if it is used in the wrong way with more errors as a consequence. QuantumDB works for the tested use cases, that is shown, but it has not been shown that it is simpler or less error-prone than when developers do not use QuantumDB. Future research can look at the steps in the procedure that is used now and compare it with the procedure when QuantumDB is being used. Researchers can look at the time spend dealing with the migration and the errors made during migration. They can also look at the perceived difficulty by developers and the time taken to learn to use QuantumDB. During this research, additional features can be implemented that would make it easier to learn and use QuantumDB in the future.

Chapter 8

Management (business case & use cases)

This chapter will discuss the business case as well as the use cases of a zero-downtime schema migration tool. QuantumDB will be mainly used as the reference in the use cases analysis as it was developed further and validated in this master thesis. Executive management of companies can read this chapter to get an overview of the benefits and costs of implementing a tool, as well as for which use cases the tool would fit best and which it would probably not work. Since QuantumDB has not yet been implemented within an organization, the first to do so will probably encounter additional problems and limitations, but would have an advantage over competitors not implementing a zero-downtime solution.

8.1 Problem context

Updating database schemas without downtime is one of the only procedures that still does not occur at organizations when updating applications. The front-end, back-end, and other intermediate software tools can mostly all be updated without bringing down the system, as multiple versions can co-exist to some extend. Database schemas are an exception in that the underlying data that is governed by a format (the schema) has to be the same across all versions that are used. NoSQL databases do already provide some ways to update a schema without downtime, if such a schema even exists in that database. However, updating relational SQL database schemas without downtime is still relevant due to the ACID principles governing the data. NoSQL databases lack some or most of these principles making it less useful to use them to store data when applications need strict requirements.

The most important part of a solution is that organizations do not need to schedule maintenance windows each time the database schema needs to be changed. Having to deny clients access to your service, be it intentional or unintentional, can be detrimental to the perceived satisfaction [51] [52] of clients with your service or product. If updates happens once a month or less often for a couple of minutes, most clients would not even notice and satisfaction would drop an negligible amount. However, if schema updates happens multiple times a day or for a lot longer, say 2+ hours, clients would become upset that the service is not available when they want to use it.

Adopting a zero-downtime database schema migration solution would solve the above sketched problem and would even introduce more benefits. These are listed below, together with the cost of implementing a solution. At the end of this chapter, potential use cases are given for which QuantumDB, or another solution, would work the best.

8.2 Benefits

Benefits of a solution can be anything, a 1% increase in profit margin, an increase in happiness of employees, or compliance with some regulation. Some of these benefits are easier to quantify into a concrete amount of money than others. A 1% increase in profits is rather easy but how do you quantify the happiness of employees? Chapter 8 of the book Applied Cost-Benefit Analysis by Brent R. J. [53] goes in depth on how to quantify these intangible and unquantifiable benefits and costs. It mainly consists of finding a physical unit to measure the benefit or cost metric. For example, noise pollution can be measured in decibels. After determining this, a monetary value has to be assigned to this unit. This can be difficult because most of the time no standard value can easily be found in the literature or online. Research has to be done to find an estimate. Having a single metric to compare costs versus benefits helps in decision-making. If two metrics are used, one of the values need to be converted into the other for a better comparison. This value is almost always an amount of money.

If the amount of money the solution brings in outweighs the amount of money it costs, the solution can be said to deliver value. But not every company implements every solution. Most of the time, companies have a whole list of potential solutions that they can implement, so which ones do they choose? That question depends on multiple factors but one of them is that they choose the solutions with the best cost-benefit ratio [54]. If, for example, a solution only costs one person one day of work but brings in €10.000, you would be a fool to not do it (except when other more profitable projects are present). Whereas if it would take one person one month to bring in the same amount, the uncertainty in potential profit and ratio between cost-benefit would make it less attractive.

No company is the same and so the trade-off between benefit and cost is also different for every company. Therefore, no clear amount of money can be given that a business will earn with a solution for zero-downtime schema migrations. A startup might not benefit much from the solution but it would also not costs that much to implement and keep using, as they probably do not have a large infrastructure, much data, and many customers yet. Whereas a large corporation might benefit extremely well because small improvements to a process might bring in a large amount of money.

Depending on the application of the database, various benefits can be achieved. If the database is used for internal applications used by employees, different benefits are achieved than if the database is used for externally facing applications that clients/customers are using. This is the first distinction that has to be made when trying to calculate the benefits of a solution. Implementing a tool for databases from internal applications has fewer benefits compared to databases from externally facing applications. A scenario can also exist where both internally facing applications make use of the database as well as externally facing applications, in this case, benefits from both categories can be combined.

This chapter will focus on the costs and benefits of implementing a zero-downtime database schema migration solution and give examples of what to include in the cost-benefit analysis that companies should make to decide to implement a solution.

8.2.1 Internal

Benefits achieved from implementing a zero-downtime schema migration tool in internally facing databases include various aspects depending on certain circumstances. One of these circumstances is the number of different shifts in which employees work. If only one shift exists where employees use the application and therefore the database, benefits are not much. It is only the savings of the additional salary that is paid (overtime) to the employees performing the migration and additional opening time costs for the office when performing a migration versus when no tool is used.

If two shifts of eight hours exist, the calculations do not need to be changed as there is still a window where no one is working with the database. However, if shifts exist around the clock without any time

in between, then the calculation gets a bit more complicated. To minimize the time in which employees cannot work with the application, these maintenance windows would have to be scheduled when the least amount of employees will work with the application. If companies do not plan the migrations during that time they are unnecessarily losing more productive working time that could have been saved if the migration had been moved to a quieter time. The benefit of implementing a zero-downtime schema migration tool is the elimination of lost working time.

Also depending on the nature of the application, the benefits of a solution can differ. Zero-downtime migration for a crucial application that employees constantly use to perform their tasks has a higher benefit due to the elimination of lost working hours. Applications that employees sometimes use whenever they have time can be upgraded with downtime for far fewer costs as it matters less that they possibly have to wait for a while.

To summarize, the benefits of a zero-downtime solution for internally facing applications can range from a small monetary amount to a substantial one. The benefits of a solution are maximized when the application is used 24/7 for critical business processes. The benefits include the otherwise lost revenue during that downtime but also the cost saved by unproductive employees. The benefits of a solution become increasingly smaller when the application is not used 24/7 or is not critical to business processes. In that case, only the additional employee overtime costs and extended office opening hours have to be calculated as saved costs.

8.2.2 External

The benefits of externally facing applications that use a database are more pronounced as more factors have to be taken into account. One of the benefits is that the service can continue generating revenue during upgrades where, normally, revenue would be missed. The extra revenue is not the same as the benefits, as most people might just wait for the service or product to be online again before spending their money. It has to be measured what percentage of customers do not delay their spending until the service is up again.

Two questions to answer are: 'How much revenue is missed out on during times when there is a maintenance window?' and 'How much extra employee costs are made when there is a maintenance window?' Multiply these costs by the number of maintenance windows there are in a year to get the benefit of not having maintenance windows where the database is updated.

8.2.3 Time-to-market of features & bug fixes

Additional benefits also come in the form of decreased time-to-market [55] of features & bug fixes. If features or bug fixes are normally delayed for one month before coming online, a zero-downtime solution can accelerate releases and therefore achieve more value at a quicker rate. To calculate the benefits is not that easy. Knowledge is needed about the value of these new features and bug fixes and the average time-to-market when developing them. To begin an estimation, categories of errors can be thought of from insignificant to critical and for every category, a cost can be assigned. The benefit of a zero-downtime solution is the decreased time that the errors exist in production. The benefit is the decreased time in percentage until the release of fixes multiplied by the number of fixes and again multiplied by the cost of the bugs/errors.

Estimating the benefit of new features can be done by logging the usage of new features and calculating increased sales, usage, and/or client satisfaction. An increased usage or client satisfaction means that clients are less inclined to switch to competitors which prolongs the time that they use the service or product [56]. Bringing these features sooner to your clients can help with client retention as competitors might not have their new features online (yet). This is called competitor advantage and might

be really important for some industries where first-movers can have significant advantages over their competitors [57].

One might say that the calculation above is not really a fair calculation as some new features and most bug fixes do not require a database schema update, and that is true. However, the calculation becomes more difficult the more aspects are included. By implementing a zero-downtime solution all feature and bug fix updates can be performed as soon as they are ready which is the advantage of CI/CD. It will therefore reduce the average time-to-market of all features and bug fixes as developers do not have to think about if their update should be scheduled during a maintenance window.

8.3 Costs

Costs of implementing a zero-downtime schema migration tool are of course the salary paid to employees that work on it. Initial costs might be higher as the tool would have to be validated to work, otherwise, unexpected problems might occur. The release pipeline would have to be changed, depending on how it is configured before the implementation, either a lot needs to change or little.

Employees would need to be trained in using the tool. Due to the effort in making QuantumDB easy to use, it is not expected that a lot of time is required to learn QuantumDB. Due to the similar syntax of Liquibase, if developers and database administrators already use Liquibase, only some additional training is required for learning the specific utility of QuantumDB.

An unquantifiable cost when implementing a zero-downtime schema migration solution is the missed opportunity of not being able to go after other projects, because some employees are busy. However, this is the case with any project that is decided to be done, so these costs can be discarded.

Lastly, every solution will not continue working indefinitely, maintenance will need to happen to ensure the tool is working on future versions of database management systems. These costs can be higher if development teams switch to newer software versions more quickly, instead of waiting for the next major release. If maintenance stops for some reason, the tool cannot be used after a certain while due to various phenomenon such as bit-rot [58]. Care has to be given into making sure that the tool is to be swapped out on time for a new tool or development teams have to use a different methodology to achieve the same results. This process will also cost money as is described by the total cost of ownership (TCO) model [59]. In short it describes how much you think it will cost to also dispose of the solution at the end of its lifetime.

8.4 Use cases

QuantumDB is shown to perform well under specific circumstances, while other configurations do not seem to work or outright will not work, these can be found in Section 7.2. In case a manager of an existing project wants to adopt QuantumDB, an analysis will have to be done to ensure QuantumDB can work with the existing database layout. One of them is that no functions, stored procedures, and triggers exist on tables. These features might be implemented in the future, but as of writing this thesis, QuantumDB does not support these. This is not necessarily a hard restriction as these functions, stored procedures, or triggers can exist, but no transformations can happen on these tables as expected functionality would break. Secondly, as can be read in Chapter 6, tables may not contain a composite primary key or have no primary key constraint at all. Performance drastically drops when using a composite primary key, likely due to unoptimized indexes. In case of no primary key, a migration on that table might not even finish in finite time or will cause undefined behavior. The simple fix is to perform a transformation on that table that introduces a new auto-incrementing column as the primary key.

Also, the amount of data contained in the database might cause issues as more data means longer migration times. It has been shown that the performance drop of running queries does not considerably suffer while a migration is running, but if a project wants to release a new version every couple of hours, it might not be possible due to the time it takes to migrate all the data. Managers for existing projects need to make clear what their goals are when they want to implement QuantumDB, releasing multiple times each day with a database containing hundreds of millions of rows might not be doable. Research needs to be done on how long it takes for a certain amount of rows to migrate on specific hardware. Managers should also keep in mind that when old schema versions need to be dropped no users should be using that version anymore.

Projects that have not started yet have an easier time adopting QuantumDB as certain design principles can be changed before development starts. QuantumDB works best if databases are just used as simple 'data stores', instead of intelligent databases that have to perform operations on data. Therefore, when using QuantumDB, applications need to take over the calculations that would have been done at database level. This would not be difficult but would require a bit of planning to perform calculations at the application level instead of the database level. This would also free up system resources at the database level where scaling possibilities are more limited, whereas at the application level that is not the case.

In short, development projects that still have to start or are in the early stages of development can definitely make use of QuantumDB and the benefits. Some limitations have to be worked around but will not necessarily limit capabilities. Existing projects will likely need to consider various aspects and determine if QuantumDB is a good tool to use. Complex functions within the database need to be shifted to the application level, short release times while huge amounts of data are present in the database have to be carefully planned, and composite primary keys have to be transformed to unique, not null indexes while a new column is made that is an auto-incrementing primary key.

Chapter 9

Conclusion

This chapter first lists a summary and the contributions to science of this master thesis. After that section, the answers to the research questions are summarized. The threats to validity, key challenges, and future work were discussed in Chapter 7.

9.1 Summary & contributions

This research resulted in various contributions to science. Firstly, a literature review was conducted to summarize most of the research in the field of zero-downtime schema migrations. Secondly, criteria for a zero-downtime schema migration solution were derived from various literature sources. Thirdly, an overview was given of the solutions that are currently available to be used. During the design cycle of the methodology, stakeholders were identified together with the goals that they want to achieve.

During the treatment design stage, a more specific criteria list with a ranking was made using a questionnaire sent to experts. In this questionnaire, experts could indicate which criteria were more important and which were not. A ranking of the available solutions was made to show which solutions comply the most with the criteria. The solution with the best compliance to the criteria was tested and improved. These improvements were partly based on the opinions of experts on identified key challenges why a solution is not yet used within organizations. Some solutions were implemented as well to mitigate the identified key challenges.

Next, the solution was validated to work on three different experiments to ensure a correctly working solution. During validation, some other limitations of the solution were found and ways to solve these were listed. Some of these limitations are fundamental problems that are yet to be solved by future research, these are listed in Section 6.5.

A business case and use case analysis was made for companies to adopt a zero-downtime schema migration solution. This business case lists the potential benefits and costs and pictures an improvement in the speed of updates in the development pipeline when using a zero-downtime tool.

Next to all that, QuantumDB, the tool that scored the highest on the criteria, was improved and the changes to the code were pushed to the official GitHub repository[1]. A total of 5.809 additional lines and removal of 1.737 lines in 42 commits were pushed. The complete commit history can be viewed in the issues and pull request tab with additional comments from the original developer. Most issues have been solved while some are still open and will have to be solved in the future.

At last, future researchers could continue this master thesis research by looking at the list of future work and limitations that was made in Section 7.2. Potential areas to explore further are the implementation of additional features and functionality of QuantumDB, additional testing and validation of

[1] `https://github.com/quantumdb/quantumdb`

QuantumDB to ensure proper operation, and additional research into why organizations will not invest in the implementation of a zero-downtime schema migration solution. Additionally, implementing a zero-downtime schema migration solution in an organization and investigating the key challenges that come up is also a research direction that can be followed.

9.2 Answers to the research questions

To recap the research questions from the first chapter:

1. What are the criteria for a zero-downtime schema migration tool?
    1.1. Which criteria can be found in literature?
    1.2. Can a ranking be made between criteria?
2. Which tools are already available that aim to provide zero-downtime schema migration?
    2.1. Can a best tool be identified based on the criteria gathered?
3. What key challenges can be found?
    3.1. What technical challenges can be found that will hinder the creation of a tool?
    3.2. What challenges can be found that prevent implementation of the tool within an organization?
4. For which use cases can the tool be used?
    4.1. Are there specific use cases that the tool would be great for?
    4.2. What would need to change for the tool to become better and support more use cases in the future?

All of the answers below are already listed in previous chapters but are summarized here in a condensed form.

9.2.1 Research question 1

Research question 1 was asked to get a clear picture of the requirements and to see if a ranking of importance could be made. Criteria were extracted from various articles that specifically specified them for zero-downtime schema migration tools. More articles were found that specified criteria but for tools that would perform a bit differently, these were not included. Via a questionnaire, experts could rank each criterion with their view of the importance of the criterion. They could also introduce new criteria, but that did not happen. The criteria can be found in Table 3.1 where they are categorized and a short description is given. In total, 21 criteria were defined over 4 categories. Tables 5.1, 5.2, 5.3, 5.4, and 5.5 show the results of the ranking at the bottom of each table. Only the composability requirement got a score of 2, meaning that experts rated compliance to this criterion as unnecessary but nice to have. Other requirements were all in the range from 'nice to have the compliance' to 'necessary for compliance'.

9.2.2 Research question 2

It would have been unwise to start from scratch when a perfectly fine tool already exists. Therefore, research has been done into which tools exist and a list of tools that promise to provide zero-downtime schema migrations can be found in Table 3.3. These tools all came with their own limitations, as some could not even be found except for the article or master thesis that was written about them. Correspondence with most authors also proved unfruitful as they would not send and would not provide support for the tool they had worked on in the past. Table 5.6 has all solutions ranked based on the compliance of the tool on the criteria multiplied by the weight given by the experts in the questionnaire. QuantumDB came out on top with the highest score, therefore indicating that it would be the best tool to continue with. QuantumDB did not comply with the 'expressivity' and 'consecutive migrations' criteria, but features could be implemented in the future to comply with these criteria.

9.2.3 Research question 3

There are always challenges remaining, if you have not found any, you probably have not searched hard enough. QuantumDB also had them in the beginning and still has some at the end. Upon first trying to test QuantumDB, it would not even run. Multiple bugs and errors were preventing the tool from operating properly. Luckily, with QuantumDB being open-source, these bugs could be fixed. After having a working demo, a panel discussion with experts was held and they were asked what challenges they foresaw with having to implement QuantumDB. These challenges are found in Section 5.5 and resolutions to these challenges are addressed in Section 5.6. QuantumDB also came with technical limitations that were found during validation testing. These limitations and the potential resolutions for these limitations can be found in Section 6.4 & 6.5.

9.2.4 Research question 4

In Chapter 8, a business case and use case analysis was made. Executive managers can read this chapter to get an idea of the benefits of adopting a zero-downtime migration solution into their release pipeline. In this chapter can be read that the time-to-market of bug fixes and features would decrease considerably as well as the costs for downtime if a solution would be implemented. Due to QuantumDB having some limitations, some use cases would not provide many benefits as workarounds would have to be made for various aspects.

QuantumDB would work considerably well with projects that are just starting and want to release multiple times each day. Having to block access to your clients when they are still starting to adopt the new service is detrimental to the adoption rate. Where QuantumDB also shines is in applications which are generating revenue around the clock. Having to bring the service down for maintenance a couple hours each month will hurt sales, even if carefully planned in the least used periods.

Bibliography

[1] J. Humble and D. Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation," *Continuous delivery*, p. 497, 2010.

[2] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema evolution in wikipedia - Toward a web Information system benchmark," *ICEIS 2008 - Proceedings of the 10th International Conference on Enterprise Information Systems*, vol. DISI, pp. 323–332, 2008.

[3] R. J. Wieringa, *Design science methodology: For information systems and software engineering*, 2014.

[4] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, jun 1970.

[5] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, "A Survey and Comparison of Relational and Non-Relational Database," Tech. Rep. 6, 2012.

[6] M. Drake, "A Comparison of NoSQL Database Management Systems and Models — DigitalOcean," 2019.

[7] E. A. Brewer, "Towards robust distributed systems," no. January 2000, p. 7, 2000.

[8] S. Pittet, "Continuous integration vs. continuous delivery vs. continuous deployment."

[9] L. Wevers, M. Hofstra, M. Tammens, M. Huisman, and M. Van Keulen, "Towards Online and Transactional Relational Schema Transformations," Tech. Rep., nov 2014.

[10] L. Wevers, M. Hofstra, M. Tammens, M. Huisman, and M. van Keulen, "Analysis of the blocking behaviour of schema transformations in relational database systems," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9282. Springer Verlag, 2015, pp. 169–183.

[11] L. Wevers, M. Huisman, and M. Van Keulen, "Lazy evaluation for concurrent OLTP and bulk transactions," in *ACM International Conference Proceeding Series*, vol. 11-13-July. Association for Computing Machinery, jul 2016, pp. 115–124.

[12] M. De Jong, A. Van Deursen, and A. Cleve, "Zero-downtime SQL database schema evolution for continuous deployment," *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017*, no. Section II, pp. 143–152, 2017.

[13] M. Ronstrom, "On-line schema update for a Telecom Database," *Proceedings - International Conference on Data Engineering*, pp. 329–338, 2000.

[14] G. H. Sockut and B. R. Iyer, “Online reorganization of databases,” *ACM Computing Surveys*, vol. 41, no. 3, 2009.

[15] G. H. Sockut and R. P. Goldberg, “Database Reorganization—Principles and Practice,” *ACM Computing Surveys (CSUR)*, vol. 11, no. 4, pp. 371–395, dec 1979.

[16] J. M. Hick and J. L. Hainaut, “Strategy for database application evolution: The DB-MAIN approach,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2813, pp. 291–306, oct 2003.

[17] N. F. Noy and M. Klein, “Ontology Evolution: Not the Same as Schema Evolution,” *Knowledge and Information Systems*, vol. 6, no. 4, pp. 428–440, jul 2004.

[18] J. F. Roddick, “Schema Evolution in Database Systems: An Annotated Bibliography,” *ACM SIGMOD Record*, vol. 21, no. 4, pp. 35–40, 1992.

[19] T. Deutsch, “Why is Schema on Read So Useful? — IBM Big Data & Analytics Hub,” 2013.

[20] C. E. Kaas, T. B. Pedersen, and B. D. Rasmussen, “Schema evolution for stars and snowflakes,” *ICEIS 2004 - Proceedings of the Sixth International Conference on Enterprise Information Systems*, pp. 425–433, 2004.

[21] B. S. Lerner and A. N. Habermann, “Beyond Schema Evolution to Database Reorganization,” *ACM SIGPLAN Notices*, vol. 25, no. 10, pp. 67–76, jan 1990.

[22] C. A. Curino, H. J. Moon, and C. Zaniolo, “Graceful database schema evolution: The PRISM workbench,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 761–772, aug 2008.

[23] C. Curino, H. J. Moon, and C. 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, New York, USA: ACM Press, 2009, p. 1.

[24] A. Oracle and W. Paper, “Oracle Database 10g Release 2 Online Data Reorganization & Redefinition Oracle Database 10g Online Data Reorganization & Redefinition,” Tech. Rep., 2005.

[25] “Online - definition of online by The Free Dictionary.”

[26] “Change - definition of change by The Free Dictionary.”

[27] H. Garcia-Molina and K. Salem, “Sagas,” *ACM SIGMOD Record*, vol. 16, no. 3, pp. 249–259, dec 1987.

[28] S. O. Hvasshovd, S. O. Hvasshovd, T. Sæter, Ø. Torbjørnsen, P. Moe, and O. Risnes, “A continously available and highly scalable transaction server: Design experience from the HypRa project,” *IN PROCEEDINGS OF THE 4TH INTERNATIONAL WORKSHOP ON HIGH PERFORMANCE TRANSACTION SYSTEMS*, 1991.

[29] J. Løland and S. O. Hvasshovd, “Online, non-blocking relational schema changes,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3896 LNCS. Springer Verlag, 2006, pp. 405–422.

[30] ——, “Non-blocking materialized view creation and transformation of schemas,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4152 LNCS. Springer Verlag, 2006, pp. 96–107.

[31] M. Klettke, U. Storl, M. Shenavai, and S. Scherzinger, “NoSQL schema evolution and big data migration at scale,” in *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*. Institute of Electrical and Electronics Engineers Inc., 2016, pp. 2764–2774.

[32] S. Scherzinger, U. Störl, and M. Klettke, “A datalog-based protocol for lazy data migration in agile NoSQL application development,” *DBPL 2015 - Proceedings of the 15th Symposium on Database Programming Languages*, pp. 41–44, 2015.

[33] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek, “Online, Asynchronous schema change in F1,” *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1045–1056, aug 2013.

[34] Y. Sheng, “Non-blocking Lazy Schema Changes in Multi-Version Database Management Systems,” Tech. Rep., 2019.

[35] I. Neamtiu, J. Bardin, M. R. Uddin, D.-Y. Lin, and P. Bhattacharya, “Improving cloud availability with on-the-fly schema updates,” Tech. Rep., 2013.

[36] Y. Zhu and C. Sciences, “Towards Automated Online Schema Evolution,” Tech. Rep., 2017.

[37] M. L. Möller, S. Scherzinger, M. Klettke, and U. Störl, “Why It Is Time for Yet Another Schema Evolution Benchmark: Visionary Paper,” in *Lecture Notes in Business Information Processing*, vol. 386 LNBIP. Springer Science and Business Media Deutschland GmbH, jun 2020, pp. 113–125.

[38] B. Alexe, W. C. Tan, and Y. Velegrakis, “STBenchmark: Towards a benchmark for mapping systems,” Tech. Rep. 1, 2008.

[39] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. A. Jacobsen, *BigBench: Towards an industry standard benchmark for big data analytics*, 2013.

[40] C. Zhang, J. Lu, P. Xu, and Y. Chen, “UniBench: A benchmark for multi-model database management systems,” Tech. Rep., 2019.

[41] K. Herrmann, H. Voigt, T. Seyschab, and W. Lehner, “InVerDa - Co-existing schema versions made foolproof,” *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, pp. 1362–1365, 2016.

[42] A. Mohapatra, K. Herrmann, H. Voigt, S. Lüders, T. Tsokov, and W. Lehner, “Seamless Database Evolution for Cloud Applications,” in *Proceedings of the 7th International Conference on Data Science, Technology and Applications*. SCITEPRESS - Science and Technology Publications, 2018, pp. 197–207.

[43] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, “Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++,” *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 117–128, nov 2010.

[44] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, “Automating the database schema evolution process,” *VLDB Journal*, vol. 22, no. 1, pp. 73–98, dec 2013.

[45] H. J. Moon, C. A. Curino, A. Deutsch, C. Y. Hou, and C. Zaniolo, “Managing and querying transactiontime databases under schema evolution,” Tech. Rep. 1, 2008.

[46] K. Herrmann, H. Voigt, A. Behrend, and W. Lehner, “CoDEL – A relationally complete language for database evolution,” Tech. Rep., 2015.

[47] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner, “Living in parallel realities - Co-existing schema versions with a bidirectional database evolution language,” *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. Part F1277, pp. 1101–1116, 2017.

[48] C. Coronel and S. Morris, *Database Systems: Design, Implementation, and Management*.

[49] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*, 2007.

[50] L. Wevers, M. Hofstra, M. Tammens, M. Huisman, and M. Van Keulen, “A benchmark for online non-blocking schema transformations,” Tech. Rep., 2015.

[51] O. Faltejsková, L. Dvořáková, and B. Hotovcová, “Net promoter score integration into the enterprise performance measurement and management system – A way to performance methods development,” *E a M: Ekonomie a Management*, vol. 19, no. 1, pp. 93–107, 2016.

[52] S. Fedushko, T. Ustyianovych, Y. Syerov, and T. Peracek, “User-engagement score and SLIs/SLOs/SLAs measurements correlation of e-business projects through big data analysis,” *Applied Sciences (Switzerland)*, vol. 10, no. 24, pp. 1–16, dec 2020.

[53] Brent R. J., *APPLIED COST–BENEFIT ANALYSIS*. Edward Elgar Publishing Limited (2006), 2006.

[54] J. Miller, “A proven project portfolio management process,” in *Proceedings of the Project Management Institute Annual Seminars & Symposium*, 2002, pp. 347–352.

[55] J. T. Vesey, “Time-to-market: Put speed in product development,” *Industrial Marketing Management*, vol. 21, no. 2, pp. 151–158, may 1992.

[56] G. N’Goala, “Customer switching resistance (CSR): The effects of perceived equity, trust and relationship commitment,” *International Journal of Service Industry Management*, vol. 18, no. 5, pp. 510–533, 2007.

[57] M. B. Lieberman and D. B. Montgomery, “First-mover advantages,” *Strategic Management Journal*, vol. 9, no. S1, pp. 41–58, jun 1988.

[58] V. G. Cerf, “Avoiding ”bit rot”: Long-term preservation of digital information,” *Proceedings of the IEEE*, vol. 99, no. 6, pp. 915–916, 2011.

[59] A. Zahran and G. Liberopoulos, “Total cost of ownership: a key concept in strategic cost management decisions Cite this paper Related papers Flow Cont rol of Failure Prone Manufact uring Syst ems: Cont roller Design T heory and Applicat i. . . .”

Appendix A

Code coverage

Package	Class (%)	Method (%)	Line (%)
All classes	79,2% (133/168)	35,6% (694/1949)	52,7% (3513/6667)
io.quantumdb.cli	100% (2/2)	55,6% (5/9)	68,8% (22/32)
io.quantumdb.cli.commands	100% (10/10)	85,7% (48/56)	47,8% (214/448)
io.quantumdb.cli.utils	75% (3/4)	50% (8/16)	66,7% (34/51)
io.quantumdb.cli.xml	66,7% (16/24)	44,6% (54/121)	56,9% (250/439)
io.quantumdb.core.backends	50% (1/2)	70,8% (17/24)	69,8% (37/53)
io.quantumdb.core.backends.planner	100% (11/11)	59,6% (53/89)	49,3% (181/367)
io.quantumdb.core.backends.postgresql.migrator	100% (1/1)	60% (3/5)	30,8% (8/26)
io.quantumdb.core.migration	100% (4/4)	77,3% (17/22)	83,6% (112/134)
io.quantumdb.core.migration.operations	100% (16/16)	80,6% (29/36)	63,1% (135/214)
io.quantumdb.core.planner	90% (18/20)	67,8% (103/152)	58,1% (936/1610)
io.quantumdb.core.schema.definitions	56,2% (9/16)	38,9% (88/226)	36,9% (231/626)
io.quantumdb.core.schema.operations	56,5% (13/23)	44,1% (63/143)	42,9% (120/280)
io.quantumdb.core.utils	66,7% (2/3)	62,5% (5/8)	71% (22/31)
io.quantumdb.core.versioning	95,5% (21/22)	77,2% (176/228)	81,6% (1082/1326)
io.quantumdb.driver	50% (4/8)	2% (16/805)	7,3% (70/961)
io.quantumdb.query.rewriter	100% (2/2)	100% (9/9)	85,5% (59/69)

Table A.1: Code coverage for consistency testing

Package	Class (%)	Method (%)	Line (%)
All classes	74,4% (125/168)	33,9% (662/1952)	52,6% (3505/6667)
io.quantumdb.cli	100% (2/2)	77,8% (7/9)	90,6% (29/32)
io.quantumdb.cli.commands	100% (10/10)	91,1% (51/56)	74,3% (333/448)
io.quantumdb.cli.utils	75% (3/4)	62,5% (10/16)	76,5% (39/51)
io.quantumdb.cli.xml	50% (12/24)	33,1% (41/124)	43,5% (191/439)
io.quantumdb.core.backends	50% (1/2)	70,8% (17/24)	69,8% (37/53)
io.quantumdb.core.backends.planner	100% (11/11)	59,6% (53/89)	49,3% (181/367)
io.quantumdb.core.backends.postgresql.migrator	100% (1/1)	60% (3/5)	30,8% (8/26)
io.quantumdb.core.migration	100% (4/4)	72,7% (16/22)	80,6% (108/134)
io.quantumdb.core.migration.operations	100% (16/16)	72,2% (26/36)	57,9% (124/214)
io.quantumdb.core.planner	90% (18/20)	69,1% (105/152)	57,5% (926/1610)
io.quantumdb.core.schema.definitions	56,2% (9/16)	38,5% (87/226)	38,5% (241/626)
io.quantumdb.core.schema.operations	39,1% (9/23)	30,1% (43/143)	30,4% (85/280)
io.quantumdb.core.utils	66,7% (2/3)	62,5% (5/8)	71% (22/31)
io.quantumdb.core.versioning	95,5% (21/22)	77,2% (176/228)	80,3% (1065/1326)
io.quantumdb.driver	50% (4/8)	1,6% (13/805)	6,9% (66/961)
io.quantumdb.query.rewriter	100% (2/2)	100% (9/9)	72,5% (50/69)

Table A.2: Code coverage for load testing

Package	Class (%)	Method (%)	Line (%)
All classes	83,7% (190/227)	49,6% (1183/2384)	75,1% (9312/12398)
io.quantumdb.driver	0% (0/8)	0% (0/804)	0% (0/1271)
io.quantumdb.core.backends.integration.videostores	85,7% (12/14)	85,9% (61/71)	82,7% (1194/1443)
io.quantumdb.cli.commands	0% (0/10)	0% (0/62)	0% (0/459)
io.quantumdb.core.schema.definitions	90% (18/20)	61,3% (203/331)	72,9% (725/994)
io.quantumdb.core.versioning	93,8% (15/16)	77,6% (170/219)	86,7% (1169/1348)
io.quantumdb.core.planner	100% (19/19)	92,9% (195/210)	92,6% (1752/1893)
io.quantumdb.cli.xml	100% (25/25)	87,5% (56/64)	84% (336/400)
io.quantumdb.core.schema.operations	97,1% (33/34)	76,1% (121/159)	77,1% (373/484)
io.quantumdb.core.migration.operations	80,8% (21/26)	85,5% (65/76)	90,6% (511/564)
io.quantumdb.core.backends.planner	91,7% (11/12)	89,1% (122/137)	96,9% (1744/1799)
io.quantumdb.cli	0% (0/2)	0% (0/11)	0% (0/37)
io.quantumdb.cli.utils	0% (0/3)	0% (0/11)	0% (0/51)
io.quantumdb.core.state	100% (1/1)	60,6% (20/33)	72,2% (104/144)
io.quantumdb.core.backends	75% (3/4)	81,4% (35/43)	85,2% (195/229)
io.quantumdb.core.migration	100% (5/5)	75,7% (28/37)	88,2% (165/187)
io.quantumdb.core.backends.postgresql.migrator	100% (1/1)	57,1% (4/7)	32,1% (9/28)
io.quantumdb.query.rewriter	100% (3/3)	81% (17/21)	86% (104/121)
io.quantumdb.core.utils	83,3% (5/6)	92,3% (24/26)	89,6% (95/106)
io.quantumdb.core.backends.integration.types	100% (3/3)	100% (16/16)	99,1% (460/464)
io.quantumdb.cli.xml.operations	100% (14/14)	100% (42/42)	100% (342/342)
io.quantumdb.core.backends.integration.multistate	100% (1/1)	100% (4/4)	100% (34/34)

Table A.3: Code coverage for unit testing

Appendix B

ING application operations

	aax	pam*	irs
addColumn	6	79	10
addForeignKey	0	38	0
dropForeignKey	0	3	0
addNotNullConstraint	0	0	4
dropNotNullConstraint	0	10	0
addUniqueConstraint	0	32	0
dropUniqueConstraint	0	7	0
addPrimaryKey	0	34	3
createIndex	8	15	6
createSequence	4	37	0
createTable	2	37	5
dropColumn	4	3	1
dropIndex	0	2	2
dropSequence	2	0	0
dropTable	1	4	1
modifyDataType	0	2	1
renameColumn	3	0	5
renameTable	0	0	1
sql	1	0	8
sqlFile	0	5392	1
tagDatabase	0	739	0
update	0	0	2
NChangeSets	10	6131	38
NRollbackableChangeSets	7	1545	24
NChanges	31	6131	50
NRollbackableChanges	23	739	34

Table B.1: Liquibase operations from 3 ING applications

Appendix C

Testing code

Listing C.1: HammerDB load testing changelog

```
<?xml version="1.0" encoding="UTF-8"?>

<changelog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.quantumdb.io/xml/ns/quantumdb-changelog"
        xsi:schemaLocation="http://www.quantumdb.io/xml/ns/quantumdb-changelog-0.3.xsd">

    <changeset id="changeset1" author="Nick Richter">
        <description>Changeset1</description>
        <operations>
            <addColumn tableName="order_line">
                <column name="ol_tax" type="numeric(3,2)" defaultExpression="0.21"/>
            </addColumn>
        </operations>
    </changeset>

    <changeset id="changeset2" author="Nick Richter">
        <description>Changeset2</description>
        <operations>
            <alterColumn tableName="district" columnName="d_state" newColumnName="d_country"
                newDataType="varchar(2)"/>
        </operations>
    </changeset>

    <changeset id="changeset3" author="Nick Richter">
        <description>Changeset3</description>
        <operations>
            <createIndex tableName="item" columnNames="i_name"/>
        </operations>
    </changeset>

    <changeset id="changeset4" author="Nick Richter">
        <description>Changeset4</description>
        <operations>
            <dropColumn tableName="customer" columnName="c_ytd_payment"/>
        </operations>
    </changeset>

    <changeset id="changeset5" author="Nick Richter">
        <description>Changeset5</description>
        <operations>
            <addColumn tableName="new_order">
```

```
41              <column name="no_description" type="varchar"/>
42          </addColumn>
43      </operations>
44  </changeset>
45
46  <changeset id="changeset6" author="Nick Richter">
47      <description>Changeset6</description>
48      <operations>
49          <alterColumn tableName="orders" columnName="o_all_local" newColumnName="
                o_partially_local" newDataType="bigint" nullable="true"/>
50      </operations>
51  </changeset>
52
53  <changeset id="changeset7" author="Nick Richter">
54      <description>Changeset7</description>
55      <operations>
56          <createIndex tableName="stock" columnNames="s_data"/>
57      </operations>
58  </changeset>
59
60  <changeset id="changeset8" author="Nick Richter">
61      <description>Changeset8</description>
62      <operations>
63          <dropColumn tableName="warehouse" columnName="w_street_2"/>
64      </operations>
65  </changeset>
66
67  <changeset id="changeset9" author="Nick Richter">
68      <description>Changeset9</description>
69      <operations>
70          <addColumn tableName="history">
71              <column name="seller" type="varchar" defaultExpression="'None'" nullable="true
                    "/>
72          </addColumn>
73      </operations>
74  </changeset>
75
76 </changelog>
```

Listing C.2: HammerDB SQL changes to accomodate QuantumDB

```
1  -- Drop existing primary key constraint
2  ALTER TABLE customer DROP CONSTRAINT customer_i1;
3  ALTER TABLE district DROP CONSTRAINT district_i1;
4  ALTER TABLE new_order DROP CONSTRAINT new_order_i1;
5  ALTER TABLE order_line DROP CONSTRAINT order_line_i1;
6  ALTER TABLE orders DROP CONSTRAINT orders_i1;
7  ALTER TABLE stock DROP CONSTRAINT stock_i1;
8
9  -- Add new bigserial auto-incrementing id column as new primary key
10 ALTER TABLE customer ADD COLUMN id bigserial PRIMARY KEY;
11 ALTER TABLE district ADD COLUMN id bigserial PRIMARY KEY;
12 ALTER TABLE new_order ADD COLUMN id bigserial PRIMARY KEY;
13 ALTER TABLE order_line ADD COLUMN id bigserial PRIMARY KEY;
14 ALTER TABLE orders ADD COLUMN id bigserial PRIMARY KEY;
15 ALTER TABLE stock ADD COLUMN id bigserial PRIMARY KEY;
16 ALTER TABLE history ADD COLUMN id bigserial PRIMARY KEY;
17
18 -- Add new unique index to replace the old automatic primary key index
```

```
19 CREATE UNIQUE INDEX idx_customer ON customer (c_id, c_w_id, c_d_id);
20 CREATE UNIQUE INDEX idx_district ON district (d_w_id, d_id);
21 CREATE UNIQUE INDEX idx_new_order ON new_order (no_w_id, no_o_id, no_d_id);
22 CREATE UNIQUE INDEX idx_order_line ON order_line (ol_o_id,ol_w_id,ol_d_id,ol_number);
23 CREATE UNIQUE INDEX idx_orders ON orders (o_id, o_w_id, o_d_id);
24 CREATE UNIQUE INDEX idx_stock ON stock (s_i_id, s_w_id);
```

Listing C.3: Python script to query new orders per minute from database

```
1  import psycopg2
2  import time
3  import decimal
4
5  conn = psycopg2.connect(host="localhost", database="tpcc", user="tpcc", password="tpcc")
6
7  def get_nopm():
8      with conn.cursor() as cur:
9          cur.execute('SELECT SUM(d_next_o_id) FROM district;')
10         nopm_count = cur.fetchone()[0]
11         conn.commit()
12     return nopm_count
13
14 def get_tpm():
15     with conn.cursor() as cur:
16         cur.execute('SELECT SUM(xact_commit + xact_rollback) FROM pg_stat_database;')
17         tpm_count = cur.fetchone()[0]
18         conn.commit()
19     return tpm_count
20
21
22 if __name__ == '__main__':
23     try:
24         old_time = time.time_ns() // 1000000
25         old_nopm = get_nopm()
26         old_tpm = get_tpm()
27
28         nopm_file = open("nopm.txt", "a")
29         tpm_file = open("tpm.txt", "a")
30
31         nopm_file.write("--New run--\n")
32         tpm_file.write("--New run--\n")
33
34         while True:
35             if (time.time_ns() // 1000000000) != (old_time // 1000):
36                 new_nopm = get_nopm()
37                 new_tpm = get_tpm()
38
39                 new_time = time.time_ns() // 1000000
40
41                 nopm = round((new_nopm - old_nopm) * decimal.Decimal(60 / ((new_time -
                       old_time) / 1000)))
42                 tpm = round((new_tpm - old_tpm) * decimal.Decimal(60 / ((new_time - old_time)
                       / 1000)))
43
44                 print("" + str(new_time) + ": NOPM: " + str(nopm) + " TPM: " + str(tpm))
45
46                 nopm_file.write(str(new_time) + "," + str(nopm) + "\n")
47                 tpm_file.write(str(new_time) + "," + str(tpm) + "\n")
48
```

```
49              nopm_file.flush()
50              tpm_file.flush()
51
52              old_time = new_time
53              old_nopm = new_nopm
54              old_tpm = new_tpm
55
56          time.sleep(0.01)
57
58      finally:
59          if conn is not None:
60              conn.close()
61              print('Database connection closed.')
62
63          tpm_file.close()
64          nopm_file.close()
```