Zero-Downtime SQL Database Schema Evolution for Continuous Deployment

Version of August 19, 2015

Michael de Jong

Zero-Downtime SQL Database Schema Evolution for Continuous Deployment

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Michael de Jong
born in Schiedam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

This work was financially supported by
TU Delft's Universiteitsfonds

Zero-Downtime SQL Database Schema Evolution for Continuous Deployment

Author: Michael de Jong
Student id: 1314793
Email: M.deJong-2@student.tudelft.nl

Abstract

When a web service or application evolves, its database schema — tables, constraints, and indices — often need to evolve along with it. Depending on the database, some of these changes require a full table lock, preventing the service from accessing the tables under change. To deal with this, web services are typically taken offline momentarily to modify the database schema. However with the introduction of concepts like *Continuous Deployment*, web services are deployed into their production environments every time the source code is modified. Having to take the service offline — potentially several times a day — to perform schema changes is undesirable. In this paper we introduce *QuantumDB* — a middleware solution that abstracts this evolution process away from the web service without locking tables. This allows us to redeploy a web service without needing to take it offline even when a database schema change is necessary. In addition *QuantumDB* puts no restrictions on the method of deployment, supports schema changes to multiple tables using changesets, and does not subvert foreign key constraints during the evolution process. We evaluate QuantumDB by applying 19 synthetic and 81 industrial evolution scenarios to our open source implementation of *QuantumDB*. These experiments demonstrate that *QuantumDB* realizes zero-downtime schema evolution at the cost of acceptable overhead, and is applicable in industrial *Continuous Deployment* contexts.

Thesis Committee:

Chair:	Prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Prof. dr. A. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Prof. dr. A. Cleve, Faculte d'informatique, University of Namur

Contents

Chapter 1

Introduction

In this chapter we will briefly cover some background information on web services, SQL databases, schema changes, and how these topics relate to each another. Using this information we will examine some of the problems software engineering and system operation teams face when deploying new versions of their web services into a production environment. From this, we will distill some of these challenges into a clear problem definition and explain why this problem is relevant for the future of *Continuous Deployment* and *Continuous Delivery*.

1.1 Zero-Downtime Web Services

Web services have become ubiquitous in the past decade as our use of the internet has changed. We have come to rely on many of these web services. Ranging from using search engines to search for contents online, connecting with friends on various social media sites, or even streaming movies over the internet to our TVs, mobile phones, and other devices. Some of these web services provide such an important function to their users that the user's tolerance of downtime of these services — the time wherein the service is not available — has declined [16]. Users expect these services to be continuously available to them, despite the fact that some are offered for free. This has also driven companies which develop and operate these web services to invest in reducing downtime. Many companies even offer Service Level Agreements (*SLAs*) to their customers, promising that their services will remain fully functional for at least a certain minimum amount of time each month. Failing to meet these Service Level Agreements — because of unexpected or planned downtime — would result in financial penalties or reimbursements.

Another interesting aspect of web services is that they do not have to be run on-premise by the customer. Many web services are operated by the companies that develop them. Depending on the business model, instead of having the customer pay for a license to run the service on-premise and the servers on which the web service has to run, the customer might pay a monthly subscription fee in exchange for the ability to access the service remotely. This eliminates the need for the customer to make a big upfront investment to buy the license and servers required to run the service on-premise. Such a concept is typically

called Software as a Service (*SaaS*). Since these services are developed and operated by the company, a service can be patched or upgraded by the company without requiring any action from its users.

There is one significant disadvantage to this new business model: How does one deploy a new version of a web service so that customers can take advantage of bug-fixes, improvements, and new features without disrupting them, or incurring downtime which impacts *SLAs* and may result in financial penalties or unhappy customers? While customers can typically upgrade on-premise services at an opportune moment of their choice (outside of working hours, or even in the weekends), companies offering a *SaaS* web service typically do not have this advantage since their customers might be spread all around the world, which means that no such service window exists where taking the service offline affects little to none of the users.

The most important observation we can make here is that it is becoming increasingly important to both the users and operators of web services, that these web services are always online and available. Users expect that the web services they rely one will remain online and available at any time. It is therefore in the interest of the operators to ensure that their web services are always available: zero-downtime web services.

1.2 Continuous Deployment

This ability to upgrade the production environment to a new version of the web service in *SaaS* offerings brings many advantages, but before we can discuss the challenges and implications of this we must first make some clear distinctions between various software practices and how they are related. Comparable to Olsson's "Stairway to Heaven" [27] which describes a ladder which all software engineering teams must climb in order to improve their software engineering methods (see Figure 1.1). Olsson et al. describes steps which transition from a "traditional" organization to ultimately a learning organization. We can use a similar structure to define the evolution of our software practices.

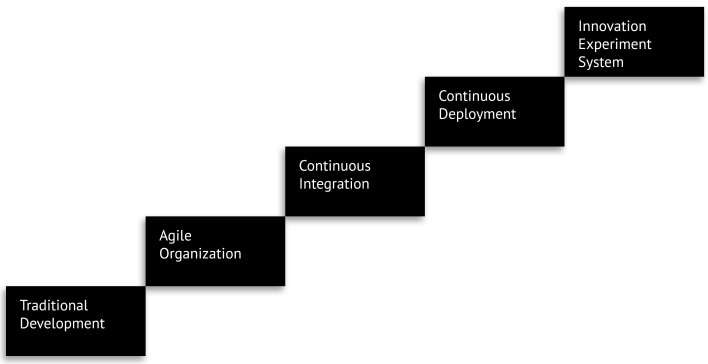


Figure 1.1: The ladder as described by Olsson et al.

1.2.1 Continuous Integration

The practice of *Continuous Integration* aims to reduce the negative effects of a software engineer committing or pushing a bug into the source code repository. Since this contaminates the source code repository, anyone pulling this change would then also have this bug in their local development environment. If the bug is severe enough, it could mean that the other software engineers cannot continue with their work because the code fails to compile or fails when running it.

To overcome this, the practice of *Continuous Integration* introduces a gatekeeper which aims to keep the bug from spreading to other branches in the source code repository. Typically this means that a build server monitors the source code repository for changes (or is notified of changes). Whenever a change in the source code has occurred it triggers a background task which attempts to compile, test, and analyse the source code of that particular branch. When this fails the software engineers are notified and have to correct the problem by making another change to the source code which fixes the bug. This triggers another build, which when successful notifies the software engineers that they have successfully solved the problem and can continue to work on something else. This continuous feedback loop aims to ensure that the source code is always in a compile-error free and functionally correct state. This allows software engineers to merge their changes into the master branch, with some level of confidence that their changes will not break the software, or prevent others from continuing their work.

To achieve this, software engineers must ensure that the source code contains unit and integration tests, which assert that the program and its components function correctly. In a sense, the quality and coverage of these tests give the software engineers a certain level of confidence that the program will operate correctly, and that their changes have not introduced any bugs.

Some elaborations of this concept go even further, by stating that all software engineers should avoid using branches as much as possible, and instead commit and push small changes to the master branch or mainline branch throughout the day. In this case the changes must be tested locally before sharing the changes with other software engineers to avoid a newly introduced bug from spreading to other software engineers.

1.2.2 Continuous Delivery

Once *Continuous Integration* has been achieved, the next step is *Continuous Delivery*. With *Continuous Delivery* the build server is also responsible for creating the final product from the source code. This can be anything from an installer, to a set of binary files which can be used to run the software. Each version of the source code is only built once, and this deliverable is then stored or archived for future use.

At this stage, the decision to deploy or release the software is made by humans. When it has been decided which version of the software is going to be deployed or released, they take the deliverable of that specific version from storage. This deliverable is then typically subjected to (manual) acceptance testing, to verify that this release is truly ready

for customers. If a deliverable passes this stage, it may be released or deployed either manually or automatically, but there is always a human initiating the release.

1.2.3 Continuous Deployment

The next evolutionary step is *Continuous Deployment*. Where *Continuous Delivery* was about creating the final deliverable, *Continuous Deployment* is about taking that deliverable and putting it into a production environment in a fully automated fashion. If this step is reached successfully by a team, it means that whenever a new deliverable is produced by the build server it is automatically deployed into the production environment. From then on customers are able to use the newest version of the program.

To reach this stage, a team must have considerable confidence in their unit, integration, and automated acceptance tests as the decision to deploy is now also automated. Since at this point there is no human action required to deploy or release the product, the product must now also undergo the acceptance test in an automated fashion. For web applications and services this typically means that a set of test cases are run using a testing framework such as Selenium[1] which starts a browser and executes the test cases as a human would do.

1.2.4 Learning Organization

The final step to reach, is to become a *Learning Organization*. If a team (or even a company) has successfully implemented the previous three steps it means it is able to take a change in source code, verify that it functions properly, package it, and deploy it into a production environment all in an automated fashion. Depending on how quickly this can be done, a change in source code might be running in production in as little as 10 minutes. This allows software engineers to automatically release their bug-fixes, improvements, and new features several times a day. This final step is about evaluating the changes they make, measuring impact, and making decisions based on those measurements. One example would be to create a prototype of a new feature, roll it out into production but only let a small percentage of users make use of the new feature. Based on their usage of the new feature, its performance, and various other metrics and measurements, the team or company might decide to roll out this new feature to all its users eventually, or remove the feature because it failed to meet expectations. Such an experiment can typically be achieved using techniques such as *Staged Deployments* (also known as *Canary Deployments*) or *Feature Flags*, which we will further discuss in Chapter 2.

Attaining this last stage promises to offer significant advantages in reducing time-to-market for new features, reducing release cycles, and reducing feedback cycles between the team and its customers. However this does imply that the web service potentially needs to be redeployed several times a day. Combine this with the need for zero-downtime web services, and attaining the stage of *Continuous Deployment* becomes significantly more challenging.

[1] http://www.seleniumhq.org/

1.3 Schema Evolution

When software engineering teams are trying to achieve *Continuous Deployment* with a web service which relies on a SQL database, they are likely to encounter the problem of schema evolution. SQL databases use rigid schemas which define the structure of the data they can store. It allows software engineers to reason about the stored data. However software is continuously subject to change [16]. This includes changes to how persistent data is stored into and retrieved from databases. Changing the schema of a SQL database becomes more challenging when additional requirements from *Continuous Deployment* are in place. One such challenge is the fact that modifying the schema of a SQL database is often a blocking operation — preventing the web service from accessing or manipulating the data stored in that database for a prolonged period of time. When a software engineering team is successfully using *Continuous Deployment* they are able to deploy changes to their software multiple times a day in an automated fashion. Potentially every change they make could require a change in the database schema. If this change prevents the web service from accessing the data stored in the database for more than a certain period of time, multiple times a day, this could easily negatively impact the user's experience.

1.4 Problem Statement

Continuous Deployment promises to offer much (business-) value for organizations, giving them an advantage over their competition. However attaining *Continuous Deployment* comes with many challenges. The essence of all these challenges is figuring out how to deal with external systems such as databases, queues, caches, and other (micro-)services during the redeployment of a web service on which the service relies upon.

In particular this thesis focusses on dealing with SQL databases, as they are a common component in many web services, and have additional limitations. One such limitation is that some Data Definition Language (*DDL*) statements — schema operations which alter the structure of the database schema — exhibit a blocking behavior. In addition, SQL databases store persistent data, on which an application often relies on to such a degree, that it may never be unavailable. If the SQL database is not available or not able to process queries, the web service could fail or appear broken to its users.

The goal of this thesis is to make *Continuous Deployment* more attainable for teams working on web services which rely on SQL databases.

To realize this goal we seek to find a solution meeting the following requirements:

- **R1 — Non-blocking DDL Statements**: A method or tool to evolve the structure of the database should not block queries being issued by any database client.

- **R2 — Concurrently Active Schemas**: It should be possible to expose multiple the data in the database in at least two database schemas. This allows for A/B testing, and rolling out new versions of the web service regardless of deployment method.

- **R3 — Schema Isolation**: A database client may only connect to one database schema, and should only be able to interact with tables, columns, etc. that are present in that chosen database schema.
- **R4 — Schema Changesets**: Software is continuously subject to change, both large and small. Writing and managing changesets comprised of multiple schema operations is needed to facilitate both small and larger changes.
- **R5 — Non-Invasiveness**: A method or tool should not require extensive integration in either the infrastructure or source code of the web service.
- **R6 — Referential Integrity**: A method or tool should support the use of foreign key constraints in the database. As we will see, existing approaches either do not, or do not sufficiently support foreign key constraints.

1.5 Approach

To realize this goal we will propose a novel tool called *QuantumDB* which implements a novel method while satisfying each of these requirements.

To evaluate our proposed method, we will conduct a series of experiments. These evaluate *QuantumDB*'s ability to keep tables under change in a modifiable and readable state, run two isolated database schemas in parallel, execute complex schema changesets, and maintain referential integrity constraints while requiring a minimal amount of changes to the original source code of the web service.

1.6 Thesis Overview

We start by covering existing methods of deployments, and schema evolution, as well as covering existing tools and research related to schema evolution without downtime in Chapter 2. We will then proceed to examine the blocking properties of existing *DDL* statements in both PostgreSQL and MySQL in Chapter 3. In Chapter 4 we will explain our approach of evolving the database schema without causing downtime. Following up we will discuss the implementation of our approach in Chapter 5. In Chapter 6 we will evaluate our approach and tool, to show that they meet all of the requirements we proposed in this Chapter. We will discuss the threats to validity and additional thoughts on the future of our tool in Chapter 7. Finally in Chapter 8 we will summarize our findings and contributions.

Chapter 2

Background and Related Work

One of the key challenges in the adoption of *Continuous Deployment* is dealing with schema evolution in databases while continuously performing redeployments of newer versions of the software into production environments. To be able to reason more about this problem and its challenges we should have a clear understanding of how web services are being deployed and what methods and tools there are for dealing with the evolution of the database schema.

2.1 Dynamic Upgrading of Zero-Downtime Web Services

Web services are complex systems, usually comprised of several subsystems like web servers, load balancers, databases, caches and others. This complexity translates into various challenges which come with operating and updating such complex systems. One of the key challenges to solve is keeping the web service available to its users while the running web service is being replaced with a newer version. Essentially from a user's perspective the following three conditions should always hold:

- The web service must always be accessible and in working order.
- A user must be atomically switched over from the old to the new version.
- When switched, a user should never be able to send requests to the old version of the web service (unless the deployment is being rolled back).

2.1.1 Load Balancing

Making such an atomic switch can be done using load balancers. A load balancer is a piece of software which may run either on commodity hardware or dedicated/specialized hardware. Load balancers redirect incoming network traffic to a specific server in a pool of available server. Requests can be redirected based on the workload of each server, on some property of the request, or on some other user-defined set of rules.

There are two main methods available to make an atomic switch using load balancers. The first is *Rolling Upgrade* and the second is *Big-Flip*.

2.1.2 Rolling Upgrade

This method is commonly used in practice to make an atomic switchover [16] from one version to the next (or previous version in case of a rollback). This method assumes that there are several servers each running an instance of the web service, and a load balancer distributing user requests among the running instances of the web service. This method consists of redeploying another version of the web service instance on every server sequentially (see Figure 2.1). This can be done by performing the following steps:

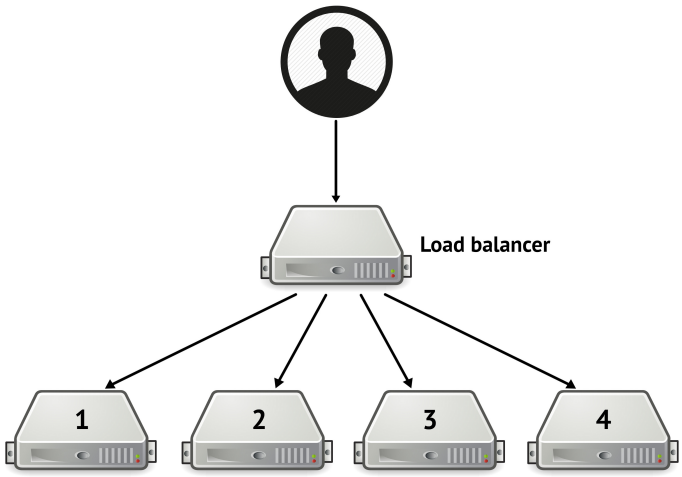


Figure 2.1: A rolling upgrade setup, where each sever is upgraded one at a time.

For every server in the server pool:

1. Instruct the load balancer that the server is unavailable. No new user requests will be sent to it.
2. Stop the web service instance running on that server.
3. Upgrade the web service instance to the new version.
4. Start the new version of the web service instance.
5. Instruct the load balancer that the web service instance is once again available and may process user requests again.

The first condition is met using this method, because it guarantees that as long as the server pool consists of at least 2 web service instances, there is always at least one web service instance available during the migration to handle user requests as the other is being updated. The second and third condition can be ensured by using a technique called *Sticky Sessions*. This ensures that requests coming from a certain user — typically identified by

his or her IP address or a cookie — are always redirected to the same specific version of web service instances.

One important side-effect to keep in mind is that when performing a *Rolling upgrade*, the entire system will be in a sort of *Mixed-State* where both versions of the web service are active at the same time. The web service instances and the related (external) systems like message queues, databases, caches etc, all need to be able to deal with this *Mixed-State* somehow.

2.1.3 Big-Flip

An alternative method is *Big-flip*. As opposed to *Rolling upgrade* where all servers are updated in sequence, *Big-flip* attempts to update all servers in parallel. In this case the servers running the web service instances are divided in two separate server pools of equal size (see Figure 2.2). Each pool has enough capacity and resources to handle all the incoming user requests. Of these two pools only one is actively handling user requests while the other pool is idle.

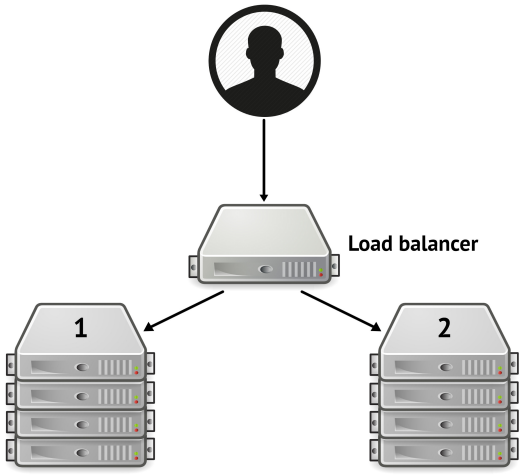


Figure 2.2: A Blue-Green setup where only one cluster handles all the incoming requests.

1. Update the web service instances in the idle pool.
2. Start the web service instances in the idle pool.
3. Instruct the load balancer to switch the roles of the two server pools. The idle pool becomes the active pool and starts handling user requests, while the active pool becomes the inactive pool and stops handling user requests.

One of the key differences between the *Big-flip* method and the *Rolling upgrade* method is that with the *Big-flip* method the switch happens atomically for every user at the same

time, whereas with *Rolling upgrade* a batch of users is switched to the new version whenever a server completes the update. This means that *Rolling upgrade* can be used to test a new version of the web service by halting the deployment after having upgraded only one server. This technique is known as a *Canary Release* which we will cover later in this Chapter. Another key difference is that the *Big-flip* method is wasteful as it requires twice the amount of minimum computing resources at the time of the deployment when compared to the *Rolling Upgrade* method.

This method however does not require the load balancer to use *Sticky Sessions* or the software engineers to employ a *Dark Releases* strategy (see Section 2.1.4), which could allow for more efficient user request routing algorithms based on server load. In addition, this method also features a small period of time where the entire system is put in a *Mixed-State*. From the moment when the initial idle pool is starting up new versions of the web service instances until the moment when the initially active pool is shutting down the older web service instances, both versions will be running side-by-side and all dependencies of the web service must be able to deal with this *Mixed-State*.

2.1.4 Additional Techniques

As organizations progress to the *Learning Organization* stage of software engineering as discussed in Chapter 1, they may want to employ certain techniques to test changes in production on a subset of their users or servers. That change can then be observed and compared to the base case to see if that change is an improvement worth rolling out to the rest of the users or servers, or to roll the change back if the results are not satisfactory. There are two techniques one can use to achieve this.

Dark Releases

Dark Releases is a technique which is typically used if software engineers wish to test out a new feature in a web service on a small subset of their users. This is done by adding conditional if-statements in the source code of the web service which checks if the user whose request it is handling is in this selected subset for that particular feature. Hence this technique is also commonly referred to as *Feature Flagging*. If the user is in this selected subset for this particular feature, the web service will perform some different action than what it would normally do. The major advantage of this technique is that it allows software engineers to see if a new feature works correctly (by monitoring errors in the log), performs well enough (by monitoring various metrics such as system load, and response times), and has the desired effect on the user experience (by monitoring the user's behavior). By starting out with an empty subset, and slowly putting users into this subset one can control how many users have access to a new feature. Using this technique one could test out a new feature on 1% of the users first and if no problems arise slowly increase this to 100%, or reduce it back to 0% in case problems do arise.

Canary Releases

The *Canary Releases* technique is similar to the *Rolling Upgrade* method we have discussed earlier in this Chapter. This technique involves taking one or a small subset of servers out of the server pool, and updating the web service instance to a different version, and then reinserting them back into active service. This allows the software engineers to test their changes out in the production environment. For instance changes to the source code with the aim of improving performance are well suited to this approach. Verifying that a certain change in the source code actually improves performance in production is not possible to do without actually trying it out in a production environment. If the change proves to be unstable, or not yielding the desired performance improvements it is easy to revert the change by taking these servers out of the pool and restoring the web service instances on them to the current version. In fact this technique may be used to run multiple experiments side-by-side in a production environment. One can choose how many users to expose to these experiments using the rules in the load balancer.

Side-Effects

One aspect to keep in mind with both of these techniques, is that they are affected by the same problem of *Mixed-State* as we have seen previously with the *Rolling Upgrades* and *Big-flip* methods. If either technique is used to, for instance, alter the schema of the database, there may be two or more versions of the web service active, each expecting a slightly different database schema as the other versions. The same applies to other systems the web service may rely on: caches, micro-services, APIs, or NoSQL databases.

2.2 Deploying Evolving Schemas

We will start this section by taking a look at two methods that can be used to deal with the problems of evolving the database schema without downtime. These methods are described in the seminal book "Continuous Delivery" [21], and are typically the go-to solutions for teams which rely on a SQL database and are starting out with *Continuous Deployment*.

2.2.1 Blue-Green Deployments

The first method for dealing with schema changes in SQL databases we will discuss, is *Blue-Green Deployments*. With this method there are two identical SQL databases. Depending on the setup these databases might reside on different physical servers for various reasons. One database is labeled with the color **Blue** while the other is labeled with the color **Green**. Comparable to *Big-flip*, where only one pool serves incoming requests and the other pool is idling, with *Blue-Green* deployments either the **Blue** or the **Green** database is used to handle incoming SQL queries. The other database is typically not in use. This idle database can be used to restore a backup, and apply blocking schema changes since it is not actively handling incoming queries. Once the schema changes have been applied, the idle database needs to catch up with all the changes in the data that occurred to the active database while applying these schema changes. To that end these changes need to be recorded somewhere

and played back after the schema changes have been applied. Once the idle database is once again synchronized with the active database, it is possible for the idle database to take over from the active database.

The following procedure describes the process of applying schema changes using this method.

1. Create a snapshot or backup of the active database, and start logging all modifications to the data stored in the active database.
2. Restore the snapshot or backup on the idle database.
3. Apply the schema changes to snapshot or backup on the idle database.
4. Replay the logged modifications on the idle database.
5. When the idle database has caught up sufficiently, temporarily block all clients from making further modifications to the active database.
6. When the modification log has been replayed entirely — meaning that both databases now contain the same data but structured differently — the roles of the active and idle database can be switched.
7. Finally, all clients may resume sending queries to the new active database.

Disadvantages of Using Blue-Green Deployments

Similarly to the *Big-flip* approach, this method also requires two separate SQL databases to prepare and perform the switch — doubling the computing resources required when using this method to prepare, and switch databases. Logging and replaying modifications might also be prohibitively expensive in write-intensive web services. Depending on the rate of inserting and manipulating records, the replay log can grow quickly which would make replaying the modifications take longer.

Also since the idle database is always attempting to catch up with the active database by replaying the logged modifications, a short period of time where data modification is prohibited is required to get the two databases ultimately in sync. Combined with instructing all web service instances to switch from the old to the new database, this could lead to some short service degradation. Some features of the application might fail or appear not to work correctly if this step takes too long.

Finally the applications issuing SQL queries to these databases needs to be aware of this process. Firstly it needs to know when to temporarily pause sending data modification queries when the two databases are trying to get in sync, and secondly know when to switch from sending queries to either the **Blue** or **Green** SQL database.

2.2.2 Expand-Contract Deployments

Another method for dealing with schema evolution in SQL databases is the *Expand-Contract* method. Where the previous method mainly complicated the deployment phase, this method

mainly complicates the development phase. This method is generic as its implementation can vary greatly depending on the used SQL database, the requirements of the web service, and the deployment process. Instead of using a secondary database, and atomically switching over between databases, this method uses only one database and applies all schema changes to this database. This method is divided into three distinct phases:

- **Expand**: In this phase the schema of the database is modified using a series of non-blocking *DDL* statements. Allowed operations during this phase vary per SQL database implementation, but typically include: creating new tables, adding nullable columns, and adding new indices.
- **Migration**: The second phase is about migrating data. In case in the previous phase it was not possible to apply the schema operations to an existing table because of blocking behavior, there are alternative options. For instance adding a non-nullable column to an existing table might exhibit a blocking behavior for a particular database. Alternatively, in the previous phase, one could create a structural copy of the table with this new column already in place. In this phase such a table should be filled with the same data as is stored in the original table, and kept synchronized. New columns, should either be set to the NULL value, or their default value if their constraints do not allow NULL values. Such a table is typically referred to as a ghost table, mirror table, or shadow table.
- **Contract**: In this final phase the schema of the database is modified again in a non-blocking way. During this phase one can delete no longer used tables, columns, or indices as long as the database can do this in a non-blocking fashion. Essentially this phase is used to clean up the database schema, and removing elements which should no longer be present in the new database schema. This step may also involve atomically renaming tables to switch older tables with their copies.

The idea behind this method is to decouple the deployment cycle and the database schema changes from each other. In other words, deploying a new version of the web service and up- or degrading the database to a newer or older schema should no longer be done at the same time. To achieve this, the web service must be able to deal with the database schema being in the original state or a *Mixed-State* — where the database schema is in some intermediate state between the current version and the next version.

Example of Evolving the Schema

In order to clearly explain how this method works we will perform an example upgrade. Imagine that a certain version of an imaginary web service uses a SQL database with a certain database schema which contains a “users” table. In addition let us assume that our database server does not allow us to add new non-nullable columns to existing tables without acquiring a full table lock — preventing the application from accessing or modifying the data stored in those tables. Now if we wish to add a new non-nullable column to the “users” table to store the user’s email address for instance, we will need to take the following action:

1. Expand the original database schema by creating a new table “users_v2” which is a structural copy of the original “users” table, and also includes the new non-nullable “email” column. Since this column is non-nullable, we are not allowed to use NULL, but will have to insert the column’s default value.

2. The next phase is migrating the existing data from the original “users” table to the “users_v2” table. This entails ensuring that every record in the “users” table is also present in the “users_v2” table, and that these two tables are kept synchronized. There are several approaches possible. One approach is to create a database trigger which copies the record from the “users” table to the new “users_v2” table whenever a record is created, updated, or deleted. Many databases support database triggers, or have an equivalent concept. In addition, a SQL query can be used to copy the old records in the “users” table to the new “users_v2” table. This should be done in batches to avoid long-lasting locks or degrading the performance of the database server for a prolonged period of time.

3. Once both tables are synchronized we can continue by creating a new version of the web service which reads and writes only to the new “users_v2” table and no longer relies on the “users” table. This new version must be deployed using a *Big-flip* method. If this is not possible, one can also use the *Rolling Upgrade* method but this requires that the database triggers are bi-directional — ie. if something changes in the new “users_v2” table, this change is also applied to the original “users” table.

4. In the contract phase we can then proceed to remove the original “users” table and the triggers which were created for this migration. The new “users_v2” table can then be renamed to its original name “users”. This in turn results in the final database schema.

Disadvantages of Expand-Contract Deployments

Please note that the previously illustrated example is a simple database schema change. More elaborate changes will require more work and perhaps even a different approach of the *Expand-Contract* method might be required. For example, the described process requires that we deploy a new version of the web service using the *Big-flip* method. If this is not possible, then we can also resort to using the *Rolling upgrade* method if we make some alterations to our approach. Since with *Rolling upgrade* there is a non-trivial period of time during the redeployment when there will be two versions of the web service active at the same time, we will need to ensure that every record that is updated by the newer version of the web service in the “users_v2” table is then also modified in the original “users” table, so that it becomes readable for the older version of the web service. To achieve this we can setup some additional database triggers which copy modified data from the “users_v2” table to the “users” table. Obviously these triggers should not interfere with the other triggers and they too should be removed in the contract phase.

There can also be cases where database triggers are not sufficient. For example when the new schema and the old schema are simply too different and the task of creating these triggers is simply too complex for a software engineer. The implementation and capabilities

of database triggers also differs between SQL servers. Both complicate the writing and testing database triggers for migrating data between two schemas. For these reasons some web services may not use database triggers but instead choose to synchronize the data between the two tables themselves in their own source code. This approach typically requires a series of deployments each handling either the expand phase, contract phase, or one of the steps in the migration phase. Since this introduces a lot of complexity — the web service is made responsible for upgrading its database schema in a non-blocking way — this complexity is usually abstracted into the database layer of the web service and hidden from the rest of the web service.

2.3 Schema Evolution Tool Support

In the previous section we examined two conceptual methods which can deal with evolving the database schema without downtime. Although these methods can theoretically also deal with the problem of blocking behavior during schema evolution, they are complex to setup and maintain. In this section we will cover tooling which has been developed specifically to tackle this complexity problem. We will see that all of these tools are to some degree based on the methods described in the previous section.

2.3.1 OpenArk Kit

The first tool we will cover is the OpenArk kit [26] — also referred to as *OAK* sometimes. OpenArk kit is a toolkit for the maintenance of MySQL databases and servers. Amongst one of its features there is a utility called "oak-online-alter-schema" which focusses on modifying the database schema using non-blocking *DDL* statements. It basically does this by implementing the *Expand-Contract* approach:

1. Create a ghost table — a "copy" of the original table — with the new structure.
2. Migrate the data from the original table to the ghost table using database triggers and a background process which copies data from the original table to the ghost table.
3. In one atomic operation, drop the original table and rename the ghost table to the original table's name.

This basic implementation of the *Expand-Contract* approach is used in practice but lacks any support for foreign key constraints. A foreign key constraint is a special kind of constraint which states that when a record refers to another record — much like the concept of a pointer in the context of computer memory — the referred record must exists at all times. In addition, all SQL databases allow the user to specify what to do when the referred record is removed: remove the referring record as well (*Cascading Delete*), prevent the removal by throwing an error (*No Action* or *Restrict*), or updating the reference to point to nothing (*Set Default* or *Set NULL*). Foreign key constraints are often used in SQL schema designs to ensure that the data in the SQL database remains in a consistent and correct state. By not supporting foreign key constraints, this toolkit's practical value is limited to

cases where no foreign key constraints are used on the table under change, or foreign key constraints are not used at all.

2.3.2 Percona Toolkit

The Percona toolkit [5] is another toolkit intended for the maintenance of MySQL databases and servers. Similar to the *OAK* toolkit, *Percona* uses the *Expand-Contract* method to evolve the schema of the database. However, *Percona* does have some rudimentary support for foreign key constraints. It can either drop and recreate foreign key constraints on referencing tables, or it temporarily disables the foreign key constraints database-wide. The former only works if the operation blocks the tables for a short — almost unnoticeable — period of time. The latter can be considered unsafe since it allows database clients to insert inconsistent data referring to non-existing records [4].

2.3.3 Facebook — Online Schema Change

Facebook has taken a slightly different approach in handling the problem of schema evolution without downtime. Their solution Online Schema Change (*OSC*) [3] is also aimed at MySQL databases, but instead they follow a slightly different procedure than *OAK* and *Percona* do. This tool employs concepts from both the *Contract-Expand* method and the *Blue-Green* method:

1. Create a ghost table of the original table including its data. This is done by writing the contents of the table to several files and then inserting the contents of those files into the ghost table. In some cases indices may be dropped and recreated during this process to speed it up.

2. At the same time start recording changes to the data in the original table in a special statement logging table.

3. Apply the schema changes to the ghost table. This operation blocks but does not interfere with user generated queries as those are directed at the original table, and this ghost table is not yet being used.

4. Replay all the recorded statements from the logging table onto the ghost table.

5. When the statement log has been emptied, atomically drop the original table and rename the ghost table to the original table's name.

The main difference between Facebook's approach and *OAK*, and *Percona* is that changes in data in the original table are logged and replayed at a later time when the schema changes have completed. This differs from the other tools which first apply the schema changes to the ghost table and then start copying data. It is worth noting that Facebook's *OSC*, like *OAK* has no support for foreign key constraints.

2.3.4 TableMigrator

TableMigrator [6] is comparable to the previous *Expand-Contract*-based tools but takes yet another approach at copying the data from the original table to the ghost table. Instead of using triggers to detect and propagate changes to the ghost table, *TableMigrator* requires each row in every table to carry a timestamp in an additional column. By repeatedly scanning the original table and the ghost table these timestamps can be compared. If a row in the original table and the related row in the ghost table have mismatched timestamps then the row in the ghost table must be updated with the contents of the original table. Once a full scan of the original table no longer detects any modifications the original table can be dropped and the ghost table can be renamed to the original table's name in one atomic step. The reason why this approach works is because each scan acquires a write lock on the original table for a short period of time, preventing other database clients from interfering with this process, but like many other previously covered tools this approach also has no support for foreign key constraints.

2.3.5 SoundCloud — Large Hadron Migrator

SoundCloud's solution called *Large Hadron Migrator* [1] is another attempt to solve this problem. It is comparable to *OAK* and is intended to be used with MySQL databases and Rails applications using ActiveRecord or DataMapper. This approach copies the original table using MySQL's "COPY TABLE" statement, and then proceeds to synchronize the data using database triggers like some of the other tools do as well. This tool does not support foreign key constraints.

2.3.6 A Brief Evaluation

When examining tools which aim to solve the problem of schema evolution without downtime, we can make some observations:

- All of the discussed tools have none to insufficient support for foreign key constraints.
- All of the discussed tools, although different, essentially implement the *Expand-Contract* method in some form.

2.4 Schema Evolution Research

In this section we will examine what efforts have been undertaken in the academic community. We will compare proposals and concepts which resulted from these efforts with the tools described in the previous section and the methods described at the beginning of this chapter. Although this thesis focusses on zero-downtime schema evolution, which entails modifying the database schema "online", there are various efforts to aid in the database schema evolution process "offline".

2.4.1 Panta Rhei Framework — PRISM++

The most influential work regarding schema evolution of SQL databases has been done on the Panta Rhei framework. In particular the tool *PRISM* and its successor *PRISM++* have been an influential research effort [12, 13, 14]. The Panta Rhei framework is a long-running research effort into SQL databases and schema evolutions. The authors of *PRISM* have defined several Schema Modification Operators — *SMOs*, which describe how a SQL schema can be modified. Later they added Integrity Constraint Modification Operators — *ICMOs* — which define modifications on constraints like primary and foreign keys. A listing of both *SMOs* and *ICMOs* can be found in table 2.1. The simplest modifiers are easily translated into simple *DDL* statements, which can be executed by SQL databases. The more complex modifiers can be translated into a set of *DDL* statements. Using these *SMOs* and *ICMOs* the authors have been able to verify that these seem to cover all of the required operations to evolve the schema of several popular SQL databases. Most notably Wikipedia's database schema. These modifiers (especially *SMOs*) have been widely adopted by the academic community and have reappeared in several papers.

SMOs
CREATE TABLE
DROP TABLE
RENAME TABLE
COPY TABLE
MERGE TABLE
PARTITION TABLE
DECOMPOSE TABLE
JOIN TABLE
ADD COLUMN
DROP COLUMN
RENAME COLUMN

ICMOs
ADD PRIMARY KEY
ADD FOREIGN KEY
ADD VALUE CONSTRAINT
DROP PRIMARY KEY
DROP FOREIGN KEY
DROP VALUE CONSTRAINT

Table 2.1: SMOs and ICMOs as defined by Curino et al. [14].

But *Schema Modification Operators* are not the only contribution that *PRISM* produced. *PRISM* itself — although it does not solve our problem of transitioning to a new database schema without disruption — does provide a useful alternate perspective on our problem. *PRISM* is designed to aid a Database Administrator (*DBA*) in his or her job to evolve a database schema. It does this in the following way:

- The database schema evolution is described using *Schema Modification Operators*. These steps are stored and versioned so that the DBA may optionally rollback to a previous version.
- Based on this versioning of *Schema Modification Operators*, *PRISM* can deduce and produce SQL migration scripts which can be run automatically or manually.

- In addition, queries produced by an application based on an older database schema can be automatically rewritten for the newer database schema with a high success rate. Few queries need to be rewritten manually.

So instead of focussing on transitioning from one database schema to another without downtime, this framework focusses on translating queries based on an older schema into equivalent queries which can operate on the current schema. All of this is done to aid the Database Administrator (*DBA*) with transitioning from one database schema to another database schema.

2.4.2 IMAGO

The next project we will cover is *IMAGO* [16, 17]. *IMAGO* is a research project which in contrast to *PRISM* does attempt to avoid downtime which can be cause by database schema operations. It uses the same definition of operators — *Schema Modification Operators* — as they have been defined by *PRISM*. But as opposed to *PRISM* which could be regarded as a tool for a *DBA*, *IMAGO* is a system which has to be deployed on a production environment in order to operate. *IMAGO* tightly integrates at two points of the environment:

- **Ingress point**: The point where the production environment accepts incoming HTTP traffic from users.
- **Egress point**: The point where the web service instances send requests to the SQL database.

Furthermore *IMAGO* employs the previously discussed *Blue-Green Deployment* method to perform database schema updates without downtime, and the *Big-flip* method to coordinate the redeployment of the web service. To do this *IMAGO* requires a copy of the production environment — including servers where the web service instances are running. The authors of *IMAGO* have chosen this method because they believe certain *Schema Modification Operators* cannot be executed when using the *Rolling upgrades* method. This in turn is why they require integration at the ingress point — to be able to atomically switch all incoming user requests from one environment to another.

When the database schema needs to be modified *IMAGO* performs the following steps:

1. Create a copy of the production environment.
2. Transform the database schema in the duplicated environment to its new structure.
3. Lazily transfer all data from the SQL database in the production environment to the SQL database in the duplicated environment.
4. When all data has been transferred, atomically switch from the production environment to the duplicated environment and redirect all incoming traffic to the duplicated environment. Web service instances running in the original production environment can be gracfully terminated.

However since *IMAGO* relies on the *Blue-Green Deployment* method — where the migration of data is continuously catching up with the writes performed by the system — there is a period during the atomic switchover where it is not possible for the system to write the database until the migration has finally caught up entirely. This period of quiescence does not lead to downtime but does degrade the performance of the web services temporarily — the web service instances can only run in read-only modus and modifications of data block during the switchover.

2.4.3 On-The-Fly Schema Updates

Another interesting approach is that of Neamtiu et al. [25]. The authors of this paper have decided to take the source code of an existing open-source SQL database "SQLite" — which is programmed in the C programming language — and modify it in such a way that it can perform schema operations in a non-blocking fashion. To achieve this they intercept *DDL* statements issued by the database client and execute the following steps:

1. The first step is to determine the difference between the old and the new schema. If multiple *DDL* statements are issued, they are batched together.
2. The next step is to modify the schema of the database in a non-blocking way. To do this, most of the low-level operations on the database schema have been altered by the authors to avoid blocking behavior when applying these modifications. Once this operation or set of operations completes, the client's request can be completed which will allow the client to continue normal operation.
3. The final step is to fix any inconsistencies that might have been caused in the previous step. One such example is that of modifying the data type of a column. In those cases a background process is started to bring the database back into a consistent state.

One important aspect to note about SQLite is that it is an embedded SQL database. It is intended to be used and packaged inside desktop and mobile applications since these typically do not have access to a SQL database hosted on a secure server. As a consequence these types of databases usually only allow for one client to connect at any time. This means that these databases experience far less concurrency issues and make the comparison to a hosted SQL database unfair.

2.4.4 Column-Oriented Databases

Another approach to solve downtime caused by database schema changes is to ensure that the SQL database itself can deal with these changes. One notable effort in this area is that of Ziyang Lui et al. [22]. The authors of this paper have attempted to define how data schema changes could work in a column-based database. To do this they have defined algorithms in pseudo-code which perform the schema changes for all *Schema Modification Operators* as previously defined in *PRISM*. To evaluate the viability of their idea they created a small prototype using their algorithms and compared it with other SQL databases in respect to the efficiency and required time to perform schema changes.

Although this nameless prototype does not focus on actually preventing downtime during schema changes, the nature of column-based databases may be more suitable for applying schema changes in terms of locking. Instead of having to lock an entire table when executing a schema modification, locking a specific column may be enough to perform the change. This could mean that queries not relying on a new or modified column could in theory avoid the lock and still continue.

Column-based databases are not new. There are already several column-based databases available — both SQL and NoSQL databases. The authors for instance named *LucidDB* which is a column-based SQL database. However to the best of our knowledge none of the column-based SQL databases solved the problem of how to deal with downtime or service degradation for clients.

2.4.5 Google's Spanner and F1

Another effort comes from Google. Their *Spanner* database [11] and *F1* database [29, 31] — which is built on top of *Spanner* — are impressive undertakings. Although these databases are not truly SQL databases, they do accept queries expressed in a SQL-like language.

Google claims that these databases support atomic zero-downtime schema changes while operating distributed across many servers, located in multiple datacenters, and on multiple continents. To achieve this they employ various servers with the sole task of keeping time using internal GPS and/or atomic clocks. Using these highly accurate clocks the cluster is able to schedule and execute a schema change across every server in the cluster atomically at the same time. It does this with the following approach.

This approach starts by picking a timestamp in the future at which the schema change should be performed across the cluster. Then for each server in the cluster the following rules apply:

1. Allow any incoming query to proceed until we have passed the chosen timestamp.
2. Perform the schema change at the chosen timestamp.
3. Queries which do not depend on the schema may proceed. Queries which do depend on the active schema may only proceed once the schema change completes.

The schema change process itself is a complex process consisting of multiple smaller schema changes. For instance, adding a new non-nullable column to an existing table *F1*, performs the following steps:

1. Add the column to the schema and mark it as delete-only — records can only be deleted, not updated, read or inserted.
2. Once that operation has completed the column is marked as write-only — allows all but read-operations on the column.
3. A background process fills all records containing this column with a default value.

4. Once the previous operation has completed the column is marked as public — allowing all read and write operations on the column.

By splitting this process up into smaller non-blocking steps the database can continue to accept and process incoming queries while also performing schema changes in the background. In addition the data is stored in a key-value like fashion in *Spanner* which helps reducing the dependence on all columns being defined, much like the previous subsection described on "column-oriented databases".

As an interesting side note, *F1* could theoretically be put in a state where multiple schemas apply but the developers of *F1* decided to limit the number of concurrent states to two. This means that at any time either the current schema is active, or that both the current schema **and** its predecessor are active.

2.4.6 A Short Evaluation

In this section we have surveyed various academic efforts for dealing with downtime caused by the evolution of the database schema. Some projects were purposefully built to deal with this problem, while others were meant to solve related issues.

Although it does not solve the problem, *PRISM++* introduced the concept of *Schema Modification Operators*. The operators which combine multiple tables or split tables into multiple tables are a novel addition to the already existing *DDL* operations SQL databases already support.

IMAGO, although it claims to be a solution to our problem, seems to be excessively invasive in the production environment. The fact that it needs to integrate both at the connection between the web service instances and the database, and the connection between the web service instances and the load balancer (in order to facilitate the atomic switchover) makes it arguably too complex, and certainly violates **R5**. In addition this approach does not support the other type of deployment method: *Rolling Upgrades*. Ideally a solution to our problem should not impose any limitations on which deployment method is used.

Another approach was rewriting an existing database in order to be able to execute schema changes without blocking other queries. Since there are many different kinds of SQL databases in use by web services, this would mean that each would have to be partially adapted in order to deal with this. Some are open-source while others are not, preventing others from rewriting these parts of the code. In addition for some databases it might prove to be too difficult to rewrite certain aspects of it, or it might not be possible to solve this on a low level. Instead it would be ideal if the problem of schema evolution is solved outside of the SQL database, in such a generic way that the solution could support many different kinds of SQL databases. But as we have discussed earlier, only solving the blocking behavior of some *DDL* operations, only deals with **R1** and does not address the other requirements.

The last option was to move to column-oriented databases or write entirely new databases to fit your needs. Moving to a different kind of (SQL) database, or writing your own can be costly and time-consuming. It might not even be possible to switch to a different database if the new database does not support all the functionality used by the web service instances of the old database. Likewise rewriting an existing SQL database to store the data internally

into a column-oriented data format might prove equally difficult as rewriting the low-level schema operations. Google's *Spanner* and *F1* databases are unfortunately out of reach for smaller web services as they require expensive atomic clocks, and are designed to run globally distributed across multiple datacentres. Interestingly enough, *F1* does support up to two concurrently active database schemas (requirement **R2** and **R3**)

2.5 Additional Related Work

SQL databases are a mature technology, which is well-understood, commonly used, and covered regularly in literature. The evolution of database schemas and data in SQL databases is covered less so. Even so, we have identified the following contributions in related areas.

2.5.1 Schema Evolution

In the previous section we briefly covered a tool called *PRISM++*. *PRISM++* is part of a larger academic research project called the *Panta Rhei Framework*, which researches other aspects of schema evolution. Another interesting tool in the *Panta Rhei Framework* is a tool called *PRIMA* [23, 24]. *PRIMA* is a tool which allows users to query archived data stored in older database schemas as if that data was stored in the current database schema. The authors argue that this retains the original quality of the archived data which potentially fades when evolving it along with the schema. The authors achieve this by transforming queries according to changes of the structure of the database.

Besides tools from the *Panta Rhei Framework*, there are other efforts which also focus on "offline" schema evolution. One example is the tool *DB-MAIN* [20], which also aids in the evolution and maintenance of SQL databases. Using the concept of *Co-Transformations* — transformations to both structure and data — it is able to aid in rewriting existing legacy software, and generating wrappers to let legacy systems written in programming languages such as COBOL, communicate with data stored in relational databases. These *Co-Transformations* in turn have been researched further by others over time [10, 8].

2.5.2 Foreign Key Constraints

Qiu et al. [28] surveyed 10 "popular large open-source database applications" which exhibited a lack of using foreign keys in their database schemas. Since there is virtually no support for foreign key constraints in the existing tools discussed in this chapter, according to this paper there would be no need for a tool which would only add support for foreign key constraints. The authors of that paper noted that their survey only covered web applications written in the PHP programming language which might have influenced this.

2.5.3 Live Migration

Finally there have been some interesting developments with SQL databases in regards to scalability and multi-tenancy in cloud-based database services. Four systems have been developed and tested to see if it is possible to migrate a SQL database between physical

machines without becoming unavailable to its clients. This becomes important when multiple SQL databases reside on a single server and due to limits on capacity or computing resources, one or more SQL databases may have to be moved to a different machine.

The systems *Albatross* [15], *Zephyr* [18], and *ProRea* [30] all seem to have similar approaches in solving this problem. They introduce a piece of middleware which sits between the SQL databases and their clients and act much like load balancers do. These solutions redirect and manipulate incoming queries while also migrating both the schema and the data from one physical server to another. *Slacker* [7] takes it one small step further by introducing a controller which adapts the speed of the data migration between servers based on the load that is generated by the database clients. The busier the server is, the slower data will be migrated. *Slacker*'s motivation for this is to reduce the chance of potential *SLA* violations during the migration.

The idea to use such special database layers dates back to 1995: Brodie et al. [9] already proposed a special database layer — they suggested the term *Gateway* — which is able to migrate data from one database system to another. Note, however, that their emphasis was on migrating to a different kind or type of database, and not on zero-downtime.

Chapter 3

Blocking Behavior of Schema Operations

In order to understand the extent of the problem of zero-downtime schema evolution, we conducted a series of experiments. These experiments assess to what extent key schema evolution operators, such as adding a column, exhibit a blocking behavior which leads to downtime for web services when applied in a production environment.

3.1 Experimental Setup

We ran all series of experiments on a Virtual Machine hosted at Digital Ocean. This machine features 8 CPU cores at 2.40GHz, 16 GB memory, and a 160 GB SSD for storage. On this machine we installed MySQL 5.5, MySQL 5.6, PostgreSQL 9.3, and PostgreSQL 9.4. These are the latest two stable versions of both database at time of writing.

In order to understand the behavior of *DDL* statements for various databases we start by creating a table in the database of choice. This table will be filled with random data were each record represents a user of a fictional web service. We then simulate a web service by having multiple threads continuously querying this table using random *DML* statements — INSERT, SELECT, UPDATE, and DELETE statements.

For each *DML* statement we record the start and end times. Then when we apply a *DDL* statement on the table in order to change the database schema, we can detect if that statement blocks the *DML* statements, by checking that there is a non-trivial period of time where these statements are no longer completing within a reasonable time.

After the *DDL* statement completes, the schema change is reverted, and the next evolution scenarios is tested.

We defined the following list of common evolution scenarios to evolve the database schema:

- **S1**: Adding a non-nullable column to an existing table
- **S2**: Adding a nullable column to an existing table
- **S3**: Renaming a non-nullable column

- **S4**: Renaming a nullable column
- **S5**: Dropping a non-nullable column
- **S6**: Dropping a nullable column
- **S7**: Modifying the data type of a non-nullable column
- **S8**: Modifying the data type of a nullable column
- **S9**: Modifying the data type of a non-nullable column from integer to text
- **S10**: Making a non-nullable column nullable
- **S11**: Making a nullable column non-nullable
- **S12**: Modifying the default value of a non-nullable column
- **S13**: Modifying the default value of a nullable column
- **S14**: Creating a foreign key constraint on a non-nullable column
- **S15**: Creating a foreign key constraint on a nullable column
- **S16**: Creating an index on an existing non-nullable column
- **S17**: Renaming an existing index
- **S18**: Dropping an existing index
- **S19**: Renaming an existing table

Each scenario can be executed by running a single *DDL* statement in both MySQL as well as PostgreSQL. By continuously querying the table, we are load stressing the database which in turn slows down the process which performs these schema operations. Then examining the measurements of *DML* statement durations, we should be able to infer which *DDL* statements are blocking and which are not.

3.2 Nemesis

To conduct our experiments we developed *Nemesis* [2]. *Nemesis* is available under the Apache 2.0 License. With *Nemesis* we can execute scenarios **S1** through **S19** for any particular version for either MySQL or PostgreSQL. In addition *Nemesis* allows us to specify how many threads are concurrently issuing random SELECT, INSERT, UPDATE, and DELETE statements to influence the amount of load we put on the database server.

Nemesis logs all start and end times of each statement to disk. After running the entire set of scenarios *Nemesis* can take these log files, and produce graphical representations which make it easy to conclude which *DDL* statements block and which do not.

In this chapter we will briefly discuss the results of running *Nemesis* on MySQL 5.5, MySQL 5.6, PostgreSQL 9.3, and PostgreSQL 9.4. We will use *Nemesis*, using the same approach, to assess our own solution in Chapter 6.

3.3 Experimental Results

The full set of results of this series of experiments can be found in Appendix A. Below we highlight just three representative results.

The produced measurements are plotted in a graph, where the x-axis represents time, and the y-axis represents the duration of a query. In particular we focussed on the period of 15 seconds before evolving the schema, up to 15 seconds into the process of evolving the schema. In this graph vertical lines are plotted where the x position of the line represents the starting time of the query, and the length representing the duration of the query. Each different color represents a different type of query. **Green** represents INSERT statements, **yellow** represents SELECT statements, **blue** represents UPDATE statements, **red** represents DELETE statements, and finally **black** represents *DDL* statements. For emphasis, the **grey** highlighted area shows the period during which the *DDL* statement is being executed. The y-axis is not labelled as we are not interested in the actual duration of the queries, but merely the relative duration compared to before, during, and after the *DDL* statement has executed.

Adding a Non-nullable Column in PostgreSQL 9.4

In the scenario of adding a new column to the "users" table with the additional condition that its contents may never be NULL, we can clearly see that in the case for PostgreSQL 9.4 this operation blocks queries being issued by the simulated application. This is being indicated by the grey area not containing any spikes: meaning that while the *DDL* statement is being executed, none of the *DML* queries are being processed.

This shows us that this operation is not safe to use in a production environment, as it would also block such queries in the production environment.

Figure 3.1: Adding a non-nullable column in PostgreSQL 9.4

Adding a Non-nullable Column in MySQL 5.5

When repeating the same schema evolution scenario for MySQL 5.5, we see slightly different results. While the *DDL* statement is being executed, we can see that all the queries being issued by the simulated application are being blocked, with the exception of SELECT statements. This is being indicated by the grey area containing only yellow spikes: meaning that while the *DDL* statement is being executed, only SELECT statements are being processed.

This is interesting as it suggests that the data in the database can still be read by the application while the schema is being altered, but the data stored in the database cannot be modified. In essence it shows that the database is put in a sort of read-only state when the evolution is being performed.

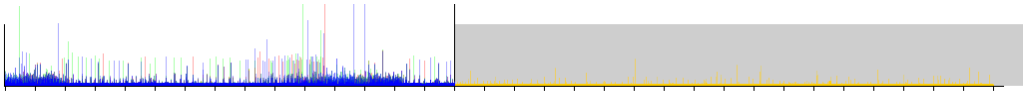


Figure 3.2: Adding a non-nullable column in MySQL 5.5

Creating an Index in PostgreSQL 9.4

The last result we will take a look at is creating an index on an existing column. The interesting conclusion we can draw from this graph is that none of the queries issued by the simulated application are being blocked by this *DDL* statement. This is being indicated by the fact that the grey area has a similar density of spikes as the period before the *DDL* statement is being executed.

As a result we can conclude that this operation would be safe to perform in a production environment.

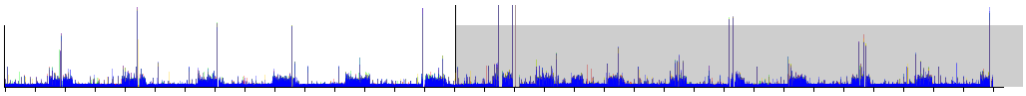


Figure 3.3: Creating an index on an existing column in PostgreSQL 9.4

3.4 Conclusion

We can tell from these series of experiments which of these *DDL* statements exhibit a behavior which is blocking other *DML* queries from being executed (see Table 3.1). This should give an indication as to which operations are safe to execute on a live database, and which are not. Note that some scenarios could not be tested with MySQL, since this database either does not support a specific operation, or it was simply not practical to test.

Equipped with this understanding of blocking behavior, we next proceed to devise an approach which would allow us to alter the database schema, without experiencing this blocking behavior in database clients.

Scenario	MySQL 5.5	MySQL 5.6	PostgreSQL 9.3	PostgreSQL 9.4
S1	Read-Only	Non-Blocking	Blocking	Blocking
S2	Read-Only	Non-Blocking	Non-Blocking	Non-Blocking
S3	Read-Only	Non-Blocking	Non-Blocking	Non-Blocking
S4	Read-Only	Non-Blocking	Non-Blocking	Non-Blocking
S5	Read-Only	Non-Blocking	Non-Blocking	Non-Blocking
S6	Read-Only	Non-Blocking	Non-Blocking	Non-Blocking
S7	N/A	N/A	Non-Blocking	Non-Blocking
S8	N/A	N/A	Non-Blocking	Non-Blocking
S9	N/A	N/A	Blocking	Blocking
S10	Read-Only	Read-Only	Non-Blocking	Non-Blocking
S11	Read-Only	Read-Only	Blocking	Blocking
S12	Non-Blocking	Non-Blocking	Non-Blocking	Non-Blocking
S13	Non-Blocking	Non-Blocking	Non-Blocking	Non-Blocking
S14	Read-Only	Read-Only	Blocking	Blocking
S15	Read-Only	Read-Only	Blocking	Blocking
S16	Read-Only	Non-Blocking	Non-Blocking	Non-Blocking
S17	N/A	N/A	Non-Blocking	Blocking
S18	Blocking	Non-Blocking	Non-Blocking	Non-Blocking
S19	Non-Blocking	Non-Blocking	Non-Blocking	Non-Blocking

Table 3.1: Blocking behavior for all scenarios which were tested with Nemesis.

Chapter 4

The QuantumDB Approach to Schema Evolution

In this chapter we propose a novel approach for zero-downtime schema evolution that meets requirements **R1-R6**. We will discuss how this approach works internally, how it can be used, and finally we will cover how it satisfies our requirements.

4.1 Versioning Schema Changes

All of the tools and methods we surveyed in Chapter 2 are either only able to make a single modification to a table, or apply multiple modifications to a single table. None of these tools and methods have the means to modify multiple tables in one set of schema changes. *QuantumDB* does allow for this, but in order to be able to reason about such sets of schema changes we need to have a concept of versioning the database schema.

We start by explaining how software engineers of a web service are able to define schema changesets. Using these defined changesets, we can infer how columns and tables relate to each other from one version to another.

4.1.1 Defining Schema Changes

There are various tools already available for versioning database schemas such as Liquibase[1] and Flyway[2]. These tools allow software engineers to define schema evolution in terms of changesets, which are comprised of a logical sequence of operations which alter or refactor the database schema. Typically each changeset contains all the required schema changes to modify the database schema to accommodate for a new feature, or fix a bug. These tools track already executed changesets in a special metadata table in the same database, and execute any not yet executed changesets which are defined in the changelog. In case of Liquibase this is defined in an XML file, and for FlyWay this is defined in either SQL files or Java classes.

[1] http://liquibase.org
[2] http://flywaydb.org

In order to be able to reason about schema changes and their effects on the database schema we also need to track this. We can use a similar approach with *QuantumDB* as these tools use. Due to time constraints and simplicity we have opted for a simple implementation in Java which models these changesets. In Figure 4.1 an example change set is described which adds a new column with a foreign key constraint to an already existing "customers" table.

```
changelog.addChangeSet("Michael de Jong",
    "Add referral column to customers table",
    addColumn("customers", "referred_by", int()),
    addForeignKey("customers", "referred_by").referencing("customers", "id"));
```

Figure 4.1: Example of a change set which adds a new column with a foreign key.

Conceptually we can think of our changeset as a series of schema changes which are applied to a certain database schema (see Figure 4.2). Each operation can be applied to a database schema, and yields another database schema, upon which the next operation can be performed until the final database schema is produced. We label each version of the database schema with a unique hexadecimal hash, comparable to Git SHA-1 hashes for commits. Later on this will help us to state from which version of the database schema we want to evolve to another version of our database schema by simply specifying these version hashes.

Figure 4.2: Conceptual representation of the change set of Figure 4.1.

4.1.2 Reasoning About Schema Changes

Determining the final database schema is not enough to construct a schema evolution strategy which entails multiple schema operations. In order to support complex change sets composed of multiple schema operations we also need to track how every table and column evolves over time through these operations, and what transformations they undergo. For instance using the "COPY TABLE" operation, we are supposed to create a new additional table with the same structure as the source table. However this new table should also contain the same data as the source table. If we then apply the "DROP COLUMN" schema

operation to this new table, we will need to account for this when we want to achieve a *Mixed-State*, because we would have to copy all the records from the original source table to the new ghost table minus the deleted column.

So it becomes important to track how tables and columns in the current database schema relate to other tables and columns in the desired database schema after applying all of these schema operations. Vice-versa we also need to know how columns and tables in the desired database schema relate to other columns and tables in the current database schema.

To achieve this in *QuantumDB* we track these relations and transformations for every table and column. Given the current database schema version, the desired database schema version, and the associated sequence of consecutive schema operations to evolve from one to the other, we are able to determine for every table and column on what other table or column they are based on. It is important to note, that tables and columns may (in theory) be based on multiple tables and columns, and tables and columns may be the basis for multiple tables and columns if we are to support all of the *Schema Modification Operators* as proposed in *PRISM*.

4.2 Mixed-State

Achieving *Mixed-State* — ensuring that the database contains the data represented in multiple versions of the database schema — is our key challenge. While our approach follows the *Expand-Contract* approach like many of the other tools do, this approach has to deal with additional challenges in order to meet requirements:

- **R6**: Maintaining *Referential Integrity* while transitioning to *Mixed-State*, while in *Mixed-State*, and while transitioning out of *Mixed-State*.

- **R4**: Allowing for complex changesets composed of multiple schema operations operating on multiple tables, at the same time.

To better illustrate how *QuantumDB* achieves this, we will execute the changeset from Figure 4.1 on a simple example database containing only three tables: **customers**, **rentals**, and **movies** as is described in Figure 4.3. This database describes a store which rents out movies to its customers.

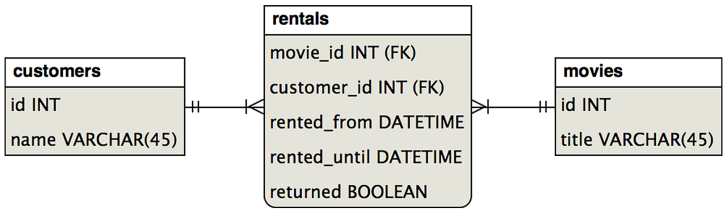


Figure 4.3: Example database schema on which we will apply the changeset of Figure 4.1.

4.2.1 Creating Ghost Tables

The first step to take is to create all the ghost tables we will need for schema evolution. The tools that we have surveyed only allow for one table to be altered at a time. In addition, this table may not refer to any other table or be referred to by other tables using foreign key constraints in most tools. These tools only create one ghost table per schema evolution. For our example database schema they would mirror the **customers** table, but they would fail since this table is referred to by the **rentals** table.

Our approach handles this differently. Although the **customers** table is the table under change and will mirror that table just like the other tools would, *QuantumDB* also mirrors any tables which depend (even indirectly) on any of the tables under change due to the way that foreign key constraints work. For our example database schema this entails that not only is the **customers** table mirrored, also the **rentals** table is mirrored. The **movies** table will not be mirrored as it does not depend directly or indirectly on any of the tables which will be mirrored. In addition, besides tables, foreign key constraints are also mirrored during this process. Since we cannot reuse the same table name during the mirroring process, we assign random table names. Figure 4.4 shows the structural end result for our example database schema after apply the changeset from Figure 4.1.

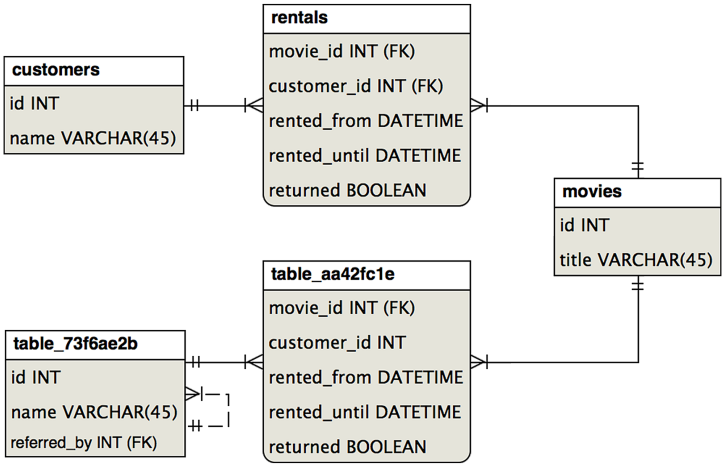


Figure 4.4: *Mixed-State* of the database schema of versions **82fba53** and **80bfa11**.

Thus, in the general case, we need to mirror the following tables:

- Any table that is subject to change by any operation specified in the changeset.
- Any table containing a foreign key constraint which directly refers to any other table which is to be mirrored. This rule must be applied recursively.

4.2.2 Constructing Forward Triggers

Now that we have achieved *Mixed-State* structurally, we also need to fill the ghost tables with data from the original tables. Since we can know that table **table_73f6ae2b** is based on the **customers** table and that **table_aa42fc1e** is based on the **rentals** table through the process as described in Section 4.1.2, we must now copy the data from the original source tables to their respective ghost tables.

But since the database is still in use, there is a chance that the records in these source tables might change while we are copying the data from a source table to a ghost table. To solve this we must first install a database trigger on the source table which will ensure that whenever a database client inserts, modifies, or deletes records in the source table, these changes will be reflected in the ghost table.

- If a new record is inserted into the **customers** table, that same data needs to be inserted into the **table_73f6ae2b** table. However since this ghost table differs structurally from its source table, the trigger must account for this. Using the process described in Section 4.1.2, we can determine that this trigger must insert the values of the id and name column from this new record into the ghost table. The new referred_by column may assume the NULL value since it is a nullable column with no default expression defined.

- If an existing record in the **customers** table is updated, we need to update the corresponding record in the **table_73f6ae2b** table. We copy the values of the id and name column of the updated record to the ghost table using an upsert — a operation which ensures that either an existing record is updated if a record with the same primary key is already present in the table, or a new record is inserted if no record with the specified primary key exists yet. We have to use an upsert because whenever a process updates a record in the source table, we may or may not yet have copied the record from the source table to the ghost table.

- If an existing record in the customer table is deleted, we also need to delete the corresponding record from the ghost table. For this we can issue a simple DELETE FROM statement which deletes the record with same specific primary key. If the record has not yet been copied, no record will be deleted.

We repeat this process for every ghost table that was constructed in Section 4.2.1. So in this case, we also create a database trigger which ensures that the corresponding record is inserted into, updated, or deleted from the **table_aa42fc1e** table, whenever a record is inserted into, updated, or deleted from the **rentals** table.

4.2.3 Migrating Data

With the forward database triggers in place, we can now start copying data from the original source tables to the ghost tables. Since some ghost tables may differ structurally from their source tables, we must account for this when copying data. We use the same process as described in Section 4.1.2 to determine which columns must be copied. Since some tables

may contain many records, we need to copy these records in batches. This avoids negatively affecting disk IO. There is a limit to the amount of data that can be read from and written to disk per second. If we do not copy records in multiple small batches, but instead copy everything in a single batch, we may need to write continuously to disk for a prolonged period of time. This can easily overwhelm the hardware, with the result of reducing the query performance of the database clients running on the database. For this purpose we copy 2,000 records per batch at intervals of 50 milliseconds. Same as the database triggers we will need to use an upsert to copy records from the source tables to their associated ghost tables. This, combined with the previously constructed database triggers ensures that all records in the source table will have a matching and up-to-date record in the ghost tables.

4.2.4 Constructing Backward Triggers

Finally, when the copying process has completed we need to install some additional database triggers on the ghost tables. These database triggers have the exact same functionality as the forward database triggers constructed in Section 4.2.2, but do the exact reverse. Whenever a record is inserted into, modified, or deleted from the ghost table, the corresponding records will be inserted into, modified, or deleted from the source tables. In this particular case the structural difference between the **customer** table and the **table_73f6ae2b** table means that the value of the referred_by column is not copied over to the source table.

If these triggers have been successfully installed, it means that the database is from that point onwards in a *Mixed-State*. It contains the same data represented in two database schemas in the same database. Any change that is made to the contents of the source tables will be reflected in the associated ghost tables, and vice-versa.

4.3 Intercepting Database Access

Now that the database is in a *Mixed-State*, the database is in a state which is opaque and too complex for connected database clients to use. This *Mixed-State* is something software engineers and the web service source code should not have to deal with as it should be abstracted away. It is clear there needs to be some sort of abstraction layer which handles this complexity for us and hides the *Mixed-State* from database clients. We essentially have two options where this abstraction layer can be implemented. At the database client's side, or on the database server's side.

4.3.1 Rewriting Queries

With *QuantumDB-driver* positioned between the web service source code and the SQL database of choice, we can intercept queries before they are sent to the SQL database. To do this correctly, whenever we create a connection with the SQL database on behalf of the web service, we first retrieve the mapping of table names to table aliases for Figure 5.3. For the previous example this mapping would look like Figure 4.5.

Version	Table name	Table alias
82fba53	customers	customers
82fba53	rentals	rentals
82fba53	movies	movies
80bfa11	customers	table_73f6ae2b
80bfa11	rentals	table_aa42fc1e
80bfa11	movies	movies

Figure 4.5: Table name mapping for the *Mixed-State* of versions **82fba53** and **80bfa11**

Then using this information, subsequent queries issued by the web service are intercepted and rewritten according to this mapping. The query is parsed, and the specified table names in the query are replaced with the table aliases which apply for this database schema version. An example of such a rewritten query can be found in Figure 4.7. The rewritten query is then sent to the SQL database, which will then operate on the ghost tables instead of the original source tables. The result of the query is then relayed back to the web service, which has no knowledge of the query having been rewritten in the process.

```
SELECT * FROM rentals WHERE customer_id = 2372
    AND return_date < NOW() AND returned = false;
```

Figure 4.6: An example query which may be sent from the web service.

```
SELECT * FROM table_aa42fc1e WHERE customer_id = 2372
    AND return_date < NOW() AND returned = false;
```

Figure 4.7: The rewritten query which will be sent to the SQL database.

Thus for every *DML* query submitted through *QuantumDB-driver*, we must parse the query, and replace the table names present in that query with the associated table IDs for that specific version of the database schema which was specified in the JDBC connection url. This ensures that every query is redirected to the appropriate table on the database server, depending on which version of the database schema was picked.

Since the table name mapping will remain unchanged as long as the specified version of the database schema is supported and the query can be quickly parsed and rewritten using the table name mapping, the overhead is minimal.

4.4 Tearing Down Mixed-State

Once we have successfully moved the SQL database into a *Mixed-State* we can effectively run two different versions of the same web service side-by-side by using *QuantumDB-*

driver. This allows us to deploy a new version of the web service whenever we have achieved this *Mixed-State* with any deployment method. However, this state adds complexity, requires more storage for storing all the additional ghost tables, and reduces performance since every change in a record in either a source or ghost table triggers a equivalent change to the corresponding records in other source or ghost tables. It is therefore important to be able to deprecate and ultimately drop tables and triggers associated with a particular version of the database schema which is no longer in use after completing the deployment of a new version of the web service.

4.4.1 Requirements

We need some method of identifying which versions of the database schema are still used by the active database clients. There are different approaches we can take to achieve this. One simple solution is to instruct the *QuantumDB-driver* to store a bit of metadata for every connection it creates with the database in the database itself. Another option would be to add some metadata to the database connection itself, and retrieve that using the administrative functions of the SQL database itself. For PostgreSQL we can store this information in the "application_name" property when creating a connection. This allows us to see which connection operates on which specific version of the database schema by listing all the properties of active database connections.

When we want to drop a certain version of the database schema, we check that there are no connections to the SQL database still using that particular version of the database schema. If there are, we terminate and notify the user that we cannot drop this version since there are still some active database clients using this version of the database schema. If none of the connections are using this version, we can safely assume that it is no longer in use, and we can proceed with the next step.

4.4.2 Dropping a Database Schema

When it is safe to remove a database schema, we must identify which tables to drop. We can do this by using the table name mapping (see Figure 4.5). Assuming we want to drop version **82fba53**, we want to drop all the tables being used by version **82fba53** and not being used by any other version. In this case that means we can drop the **customers** table and **rentals** table, since both tables are used in version **82fba53**, but not in version **80bfa11**. We then proceed to drop all the database triggers which copy records to and from these tables, after which we can safely drop these two tables, and update the table name mapping. From that point onwards the database is no longer in a *Mixed-State* and only contains data according to version **80bfa11** of the database schema.

Thus when dropping a database schema, one should remove the tables which are (according to the table mapping) present in the version of the database schema that is being dropped, and not present in any of the other active versions of the database schema.

4.5 Assessment

The proposed *QuantumDB* approach meets the requirements **R1-R6** in the following way:

R1 — Non-Blocking DDL Statements: *QuantumDB* creates ghost tables for each table under change. Any schema operations which may potentially block the execution of *DML* queries, are applied to these ghost tables. Since these tables are — during the evolution process — not yet in use by any database client, they do not hinder the performance of queries issued by any database client.

R2 — Concurrently Active Schemas: Using *QuantumDB* we can create effectively two database schemas inside the database, which are kept synchronized through database triggers. These can be accessed independently by using *QuantumDB-driver*.

R3 — Schema Isolation: *QuantumDB-driver* hides the tables and other structures present in other versions of the database schema, effectively isolating the changes a database client can make to tables existing only in their chosen database schema.

R4 — Schema changesets: Using *QuantumDB* we can version the evolution of the database schema using changesets by describing them in terms of multiple schema operations.

R5 — Non-Invasiveness: The only changes required to an existing web service written in a JVM programming language is to include a library on the classpath, and specify which version of the database schema this web service operates.

R6 — Referential Integrity: *QuantumDB* achieves this by mirroring additional tables which depend on the tables under change including their foreign key constraints, and by never loosening or disabling any foreign key constraints in the database.

Chapter 5

Implementation: The QuantumDB Tool

In this Chapter we will briefly cover some of the technical details of our implementation of the *QuantumDB* approach, and how this implementation can be used in combination with various methods of deployment of a web service.

5.1 Implementation Details

Both *QuantumDB* and *QuantumDB-driver* are written in Java. Currently *QuantumDB* only supports PostgreSQL databases. Support for other databases could be added, but might require some modifications if these databases do not support the same features as PostgreSQL does or impose additional limitations on the database schema. *QuantumDB* relies on features such as deferrable foreign key constraints, database triggers, and user-definable functions/procedures.

It features an extensive test suite containing a total of 343 tests. These tests vary from unit tests, to complex integration tests which test that a database on a PostgreSQL is correctly modified.

QuantumDB is available under an Apache 2.0 License.

5.2 Changelog API

To illustrate what kind of schema change operations we currently support with *QuantumDB* below we listed these operations with the associated DSL statement. Many of these statements follow a builder-style pattern [19], which allows a software engineer to repeat certain statements or to construct an operation which changes multiple properties in one go. In the future we may add additional operations — perhaps even the equivalents of the remaining *Schema Modification Operators*. Please note that for now it is not possible to set, or refer to the names of indices. This is due to a limitation in PostgreSQL which does not allow multiple indices with the same name to exist in the database regardless of the table they

apply to. This limitation does not apply to foreign key constraints in PostgreSQL, hence the operations for foreign key constraints uses the constraint name.

Create Table

```
createTable(<tableName>)
    .with(<columnName>, <columnType>, <hints...>)
    .with(<columnName>, <columnType>, <defaultValue>, <hints...>);
```

Drop Table

```
dropTable(<tableName>);
```

Rename Table

```
renameTable(<tableName>, <newTableName>);
```

Copy Table

```
copyTable(<tableName>, <newTableName>);
```

Add Column

```
addColumn(<tableName>, <columnName>, <columnType>, <hints...>);
addColumn(<tableName>, <columnName>, <columnType>, <defaultValue>, <hints...>);
```

Alter Column

```
alterColumn(<tableName>, <columnName>)
    .rename(<newTableName>)
    .modifyDataType(<newDataType>)
    .modifyDefaultExpression(<newDefaultExpression>)
    .dropDefaultExpression()
    .addHint(<hint>)
    .dropHint(<hint>);
```

Drop Column

```
dropColumn(<tableName>, <columnName>);
```

Create Foreign Key Constraint

```
addForeignKey(<tableName>, <columnNames...>)
    .named(<foreignKeyName>)
    .onDelete(<onDeleteAction>)
    .onUpdate(<onUpdateAction>)
    .referencing(<otherTableName>, <otherColumnNames...>);
```

Drop Foreign Key Constraint

```
dropForeignKey(<tableName>, <foreignKeyName>);
```

Create Index

```
createIndex(<tableName>, <unique>, <columnNames...>);
```

Drop Index

```
dropIndex(<tableName>, <columnNames...>);
```

5.3 QuantumDB-Driver

There are several approaches with which we can abstract away the *Mixed-State* from both the web service and the software engineer. We can either abstract this away on the database client's side, or on the database server's side. Due to time constraints we opted for the former option by creating a simple wrapper for JVM languages to showcase the concept. *QuantumDB-driver* is a small Java library which implements all required classes to act like a JDBC driver but delegates all calls to another JDBC driver. JDBC drivers are special libraries which handle communication to a specific SQL database. In essence *QuantumDB-driver* intercepts queries being issued to the database by the web service source code, and alters the queries before handing them to the actual JDBC driver (see Figure 5.1).

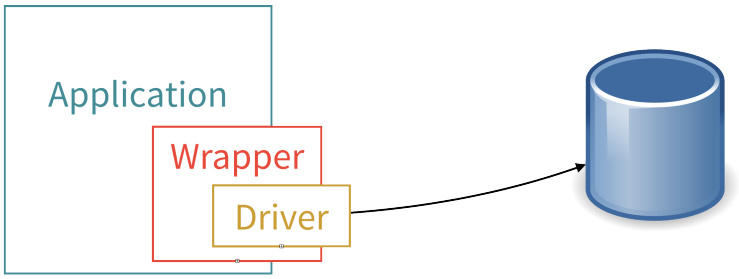


Figure 5.1: *QuantumDB-driver* intercepts calls to the actual JDBC driver.

This approach requires minimal changes to existing web services written in JVM-based programming languages. The only alterations required to use the SQL database when in or not in *Mixed-State* is to include the *QuantumDB-driver* library on the classpath, and altering the JDBC connection URL to allow *QuantumDB-driver* to intercept the calls to the intended JDBC driver, and to specify on which version of the database schema the web service source code should be operating on (see Figures 5.2 and 5.3).

```
Connection connection = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/my_database",
    "username", "password");
```

Figure 5.2: How a connection with the database is typically created in Java.

```
Connection connection = DriverManager.getConnection(
    "jdbc:quantumdb:postgresql://localhost:5432/my_database?version=80bfa11",
    "username", "password");
```

Figure 5.3: How a database connection is created using *QuantumDB-driver*.

5.4 Integration with Deployment Methods

The fact that this approach can enter and exit *Mixed-State* without the source code of the web service having to deal with this, gives us freedom in choosing a deployment method. In this section we will examine how this approach fits in with deployment methods described in Chapter 2.

5.4.1 Big-flip Deployments

The idea of *Big-flip* deployments is to have two separate but identical environments which can both assume the role of production environment. The roles of both environments is atomically switched upon every deployment. *QuantumDB* can be used in this setup.

When a deployment needs to be done using a *Big-flip*, we first prepare the new database schema by achieving *Mixed-State* using *QuantumDB*. Then when we have successfully achieved *Mixed-State*, we can atomically switch over all database clients from one version to the next using the *Big-flip* method. The new version of the web service simply connects to a different database schema version. As soon as we are sure that we do not want to roll back, we can safely exit *Mixed-State* by dropping the previous database schema.

5.4.2 Rolling Upgrades

The deployment method with which *QuantumDB* fits best together is the *Rolling upgrades* method. From updating the first server, until having completed the update of the last server, the servers are running different versions of the web service, each potentially requiring a dif-

ferent database schema to operate on. We can do this, because *QuantumDB* can effectively handle the SQL database schema being in a *Mixed-State* during the deployment.

For example, if a new version of a web service needs to be deployed which relies on a new database schema we first ensure that the database enters a *Mixed-State* of the current version and the new version of the database schema. When this has been achieved, we can perform a *Rolling upgrade* using the steps described in Chapter 2. Then finally when these steps have been completed and we are sure that we do not want to roll back, we can exit the *Mixed-State* by dropping the previous version of the database schema.

Chapter 6

Evaluation

The goal of this chapter is to provide an experimental evaluation to see to what extent our approach meets the stated requirements. We reuse *Nemesis* to assess key properties of our approach quantitatively. Furthermore, we apply QuantumDB retrospectively to a number or real life industrial schema modifications.

6.1 Measurements with Nemesis

In Chapter 3 we covered how *Nemesis* was used to profile *DDL* statements and the effects they have on database clients executing regular *DML* queries — INSERT, UPDATE, SELECT, and DELETE statements. To check if these queries are no longer blocked when using *QuantumDB*, we adapted *Nemesis* to use the *QuantumDB-driver* to communicate with the database, and instead of issuing *DDL* statements directly, an equivalent changeset of schema operations was executed to migrate the database into a *Mixed-State*. While executing these changesets we again recorded the running times of each *DML* query, and plotted the results in various graphs (see Appendix A).

We can gather from each of the scenarios that were tested using *Nemesis* that *QuantumDB* completely avoids any blocking behavior from affecting the *DML* queries executed by the mocked application using *QuantumDB-driver*. In addition, judging from the graphs, there appears to be no obvious loss of performance before, or during the transition when compared to the base cases of using *PostgreSQL 9.4* without *QuantumDB*. This loss of performance would be identifiable by higher reaching spikes in the graph.

Only one scenario could not be tested: **S19** — Renaming an existing index. This is because PostgreSQL requires all index names to be unique. Meaning that when creating ghost tables of tables which contain indices, we also need to recreate these indices on the ghost table. These new indices can however not use the same name as the indices on the original table. To solve this *QuantumDB* assigns the index a random name, making the operation which renames an index useless. Perhaps in a future version of *QuantumDB* we can translate these random names to the intended names in the abstraction layer *QuantumDB-driver*.

Figures 6.1, 6.2, 6.3 are representative graphs produced by this series of experiments. As one can see, with *QuantumDB* evolving the schema, does not block the *DML* queries oc-

curring on the original table. In fact, the graphs produced by *Nemesis* based on the measurements from benchmarking scenarios **S1-S18** indicate that each of these operations exhibits no blocking behavior. We can conclude from this that *QuantumDB* meets requirement **R1** — Non-Blocking DDL Statements.

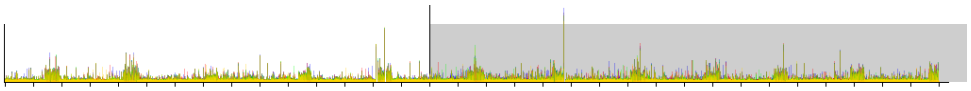


Figure 6.1: QuantumDB: Adding a non-nullable column.

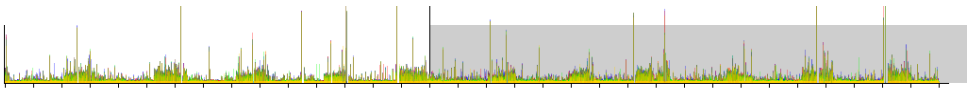


Figure 6.2: QuantumDB: Adding a foreign key constraint on a non-nullable column.

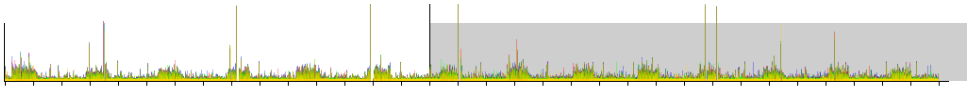


Figure 6.3: QuantumDB: Modifying a column's type from integer to text.

6.2 Deployment Scenarios

Besides using *Nemesis* to evaluate *QuantumDB*, we also decided to evaluate *QuantumDB* in a more realistic setting.

6.2.1 Experimental Setup

By taking the changelog of a tool such as Liquibase or Flyway of an existing web service or application, we can recreate this changelog in *QuantumDB* using our small Java DSL for constructing changesets. We can then use *QuantumDB* and a PostgreSQL server to run each changeset individually (using historical backups of the production data) to verify that we can perform all the operations that are being used in practice, and that *QuantumDB* can handle these typically more complex changesets.

We ran a series of experiments on a machine with 4 CPU cores, 8 GB memory, and a 256 GB SSD. The PostgreSQL database was running in a Virtual Machine assigned 2 CPU cores, 2 GB of memory, and 8 GB of storage.

We can simulate the transition from each changeset to the next as if redeploying a new version of a web service. This is done by first seeding the database with a backup of production data. We then proceeded by "forking" the existing database schema by applying one changeset to it using *QuantumDB*. After achieving *Mixed-State* with *QuantumDB* the older

database schema is discarded by dropping tables which only exist in that particular version of the database schema. This procedure can be executed using *QuantumDB*'s "drop" command. This procedure was covered in more detail in Section 4.4.

Note that we do not perform these tests while the database is under load from any database client. These tests are merely to verify that *QuantumDB* is able to perform more complex changesets. We have already studied *QuantumDB* under load conditions using *Nemesis* in the previous Section.

6.2.2 Experimental Data

With the help of a start-up called Magnet.me — an online platform for connecting students with (future) employers, jobs, and internships with over 30,000 registered users — we were able to simulate all the database schema changes they performed in their production environment since the 25th of September 2014 up to the 7th of July 2015 using database backups of production data. Since Magnet.me uses Liquibase to version the database schema and perform schema operations, we were able to recreate their entire database schema changelog with *QuantumDB*.

The Magnet.me changelog consists of 81 changesets, containing 465 schema operations in total. Each changeset on average contains 5.74 schema operations and on average each changeset applies these operations to 3.22 unique tables.

The database backup of the 25th of September 2015 starts out with 77 tables, 499 columns, 111 foreign key constraints, 42 sequences, and 28 indices not related to identity columns. The total number of records in this backup is just over 3.8 million records with the biggest table containing just over 1.4 million records.

An example changeset in this changelog can be found in Figure 6.4. This changeset applies 4 schema operations to 2 tables.

```
changelog.addChangeSet("Alex Nederlof", "Adding support for email-based invites",
    createTable("email_opt_out")
        .with("email", varchar(250), NOT_NULL, IDENTITY)
        .with("created", timestamp(true), "NOW()", NOT_NULL),
    createTable("invites")
        .with("id", uuid(), NOT_NULL, IDENTITY)
        .with("email", varchar(250), NOT_NULL)
        .with("created", timestamp(true), "NOW()", NOT_NULL)
        .with("invited_by", bigint())
        .with("viewed_landing_page", timestamp(true))
        .with("joined", timestamp(true)),
    addForeignKey("invites", "invited_by").named("invited_by_user")
        .onDelete(SET_NULL)
        .referencing("users", "id"),
    createIndex("invites", false, "email"));
```

Figure 6.4: Example of a changeset in the Magnet.me changelog.

6.2.3 Experimental Results

QuantumDB was able to execute 54 of the 81 changesets from the Magnet.me dataset without any problems. This means that in its current state, *QuantumDB* is able to deal with two-thirds of all Magnet.me changesets.

8 of the 81 changesets could not be executed by *QuantumDB* as they either only executed *DML* statements to modify the data but not the structure of a table, or to create a VIEW, or custom FUNCTION — something which the *QuantumDB* implementation does not **yet** support due to time constraints. Since the approach itself can be applied to these concepts, we could easily introduce these new schema operations in *QuantumDB* which would allow us to (re)define, or drop views and custom functions.

An additional 19 of the 81 changesets could only be partially executed as they modified both the structure, and the contents of tables using *DML* statements. For instance, a changeset which creates a new table, and inserts a set of records into this new table. In these cases *QuantumDB* was used to modify the structure, and the contents was modified by manually executing *DML* statements after *QuantumDB* achieved *Mixed-State*. Like tools such as Liquibase and Flyway, *QuantumDB* should also support versioning inserting, updating, and deleting records. This limitation is something that will need to be addressed in a future version of *QuantumDB*.

It took a combined total of 3 hours, 48 minutes, and 46 seconds to execute the 73 changesets which *QuantumDB* either fully, or partially supported. Not counting any records which had to be inserted, deleted, or updated manually, or schema operations which were not yet supported by *QuantumDB*. This meant that on average, each changeset could be executed within 3 minutes and 8 seconds.

Chapter 7

Discussion

As we have seen *QuantumDB* is able to evolve existing SQL databases using non-trivial changesets without exhibiting blocking behavior which could result in downtime. Using *QuantumDB-driver* the database schemas are isolated, and can be used with minimal changes to existing source code of web services. In this chapter we reflect on our findings, positioning them in a broader context.

7.1 Continuous Deployment

We have tested *QuantumDB* with *Nemesis* in order to examine the performance of each schema operation it supports under simulated load. We have also tested *QuantumDB* using the Magnet.me changelog to see if it could handle the more complex changeset commonly found in practice. The next step is to test *QuantumDB* out in practice, where it is subjected to these complex changesets under real-life load generated by a web service. We plan to roll *QuantumDB* out at Magnet.me, and evaluate how it will perform under these conditions.

QuantumDB currently supports up to two concurrently active database schemas. This allows us to use either the *Rolling Upgrade* or the *Big-flip* deployment method. However this means that *QuantumDB* can only deal with the complexity of having two database schemas active during *Mixed-State*. If we are able to increase this limit we might be able to support additional techniques such as *Canary Releases*, where one or more experimental branches could each operate on their own version of the database schema, running side-by-side the "master" version of the web service and database schema. This might be interesting for testing different database schemas to see which format performs better, and aid in performing A/B testing of various features which store data in different forms.

7.2 Scalability

We have tested *QuantumDB* with databases of 50 millions records in a single table with *Nemesis*, and upwards from 3.8 million using the database backups from Magnet.me. Although there are undoubtedly much bigger databases in use in practice, we expect that the

performance scales in a linear fashion: double the database size, double the time it takes to achieve *Mixed-State*.

In addition, with *Nemesis* we discovered that the database server load is an important factor in achieving *Mixed-State* as well. The busier the database system the slower the copying process becomes. In addition *QuantumDB* installs database triggers to ensure that writes, updates, and deletes are applied to two tables when in *Mixed-State*. These triggers add a small overhead, making these queries issued by database clients slower. Especially for write-intensive web services this may be noticeable. In such a case, depending on the situation, it might be an option to replace it with hardware with more computing resources to offset this cost.

More research is required to examine the "cost" or overhead of using this approach in write-intensive web services with "large" databases under realistic circumstances.

7.3 Data Loss

One aspect we have not discussed is dealing with failures. *QuantumDB* is designed in such a way that failures do not cause loss of data. Should *QuantumDB* crash for any reason, we can revert a mid-failed schema evolution by following the same approach as dropping one of the database schemas. However in this case we drop the database schema which was in the process of being created.

QuantumDB never modifies the data inside the original tables. It does manipulate data in the ghost tables, but only before installing the final backwards database triggers which are meant to update the original tables whenever the ghost tables are manipulated.

This approach makes it safe to use *QuantumDB* in a production environment, and allows its users to gracefully deal with failures, or abort the schema evolution process. We are uncertain how other tools cope with this.

7.4 Implementation Limitations

Due to time constraints not all desired functionality was built into *QuantumDB* during this project.

Currently *QuantumDB* only supports PostgreSQL and JVM-based languages. Ideally we want to support other popular SQL databases as well, but since each SQL database is slightly different and offers different features, we might need to employ a different strategy in *QuantumDB* to add support for other databases based on the features they support.

Similarly we support JVM-based languages by offering a small library which intercepts and rewrites database queries being sent to the database. It is likely that we can also create something similar for other programming languages, but it all depends on the language in question and the ecosystem of that language. Alternatively we could look into creating a mock database server which understands the protocols of various database servers, but this undertaking seems substantially more complex and demanding than solving this at the database client's side.

Another limitation that was briefly addressed in Chapter 6, was that *QuantumDB* currently does not have any support for executing *DML* statements as part of the schema evolution process. This effectively means that one cannot version both the structure and the data of a SQL database with *QuantumDB*. As we have seen in that same chapter, this is however used in practice. Therefore this is something that will need to be supported in a future version of *QuantumDB*.

7.5 Additional Evolution Scenarios

We have also briefly discussed the use of views and custom functions in Chapter 6. *QuantumDB* does indeed not yet support these, but this is something we can add support for. We will need to expand our small DSL in some way to let users express these views and functions in order to version them.

Finally, we would like to add support for the *Schema Modification Operators*, that have not been implemented yet — *MERGE TABLE*, *PARTITION TABLE*, *DECOMPOSE TABLE*, and *JOIN TABLE*. We recognize their value in describing the evolution of the structure of data. It might even help reduce the need for *DML* statements in the changelog, since these are sometimes used in scenarios which could also be replaced with one or more of these operations.

7.6 NoSQL

QuantumDB focusses only on SQL databases. These databases have an explicit database schema which is enforced by the database, ie. you cannot store data of a different structure than is defined in the database schema.

Many NoSQL databases have an implicit database schema. One popular example would be the Document Store type databases which often store their data in a JSON-like data structure. This means that they essentially allow database clients to store data in any kind of structure, as long as it can be described in terms of that data structure.

These implicit database schemas might not be enforced by these databases, we can however define various schema operations that alter the structure of these documents. In a similar approach as to *QuantumDB* we could create a ghost "table", fill it with data in the alternative structure, and in that way offer a way to deal with schema evolution in NoSQL databases.

For databases with an explicit database schema, we could also use the *QuantumDB* approach. Graph databases for instance, could perhaps also benefit from operations which could "refactor" the structure of the graph.

7.7 Threats to Validity

7.7.1 External Validity

QuantumDB currently only supports PostgreSQL, and uses a number of features provided by this database in order to achieve *Mixed-State*. Should other databases not provide these same features, and if no alternative strategy can be found to achieve *Mixed-State* using this approach, than *QuantumDB* might not be the general solution for schema evolution we believe it to be.

We have tested *QuantumDB* on the database backups and changesets from Magnet.me. Although these span the entire lifetime of the product's latest reincarnation — making it representative for Magnet.me — it does not necessarily mean that these are representative of the database size, structure, and schema changes for the rest of the industry.

7.7.2 Internal Validity

We took great care to ensure that our measurements using *Nemesis* were correct. When we encountered a set of unrealistic measurements during one of our series of experiments using *Nemesis*, we found that this was caused by an exception being thrown when executing the schema operation. We added additional exception handling code, and log statements to avoid this from happening again. The experiments were repeated just to ensure that this problem had not affected other measurements.

In addition to this, *QuantumDB* employs a test suite to ensure that it works as intended, and prevents regressions when modifying the source code. Although not indicative of the quality of the test suite, this test suite achieves 80% of method coverage, and 83% of line coverage of *QuantumDB*'s core package, as measured with IntelliJ's internal coverage tool.

7.7.3 Replication

The source code for both *Nemesis* as well as *QuantumDB*, are available for download at: `http://github.com/quantumdb`. The results of the experiments done with *Nemesis* are due to their size not available for download, but are available upon request. The Magnet.me schema modifications are confidential, as are their database backups, but these may be made available upon request for research purposes.

Chapter 8

Conclusions

The goal of this thesis is to identify and evaluate an approach which allows us to evolve the database schema of a SQL database in a *Continuous Deployment* setting.

To that end we identified a set of requirements (**R1-R6**) which should be met by an approach to successfully solve this problem. We then verified that blocking behavior is indeed exhibited by several schema operations in common SQL databases, by creating a tool (*Nemesis*) which is able to measure this blocking behavior. Based on the requirements, and existing tools and approaches, we developed our own approach, and implemented a prototype (*QuantumDB*). We subjected this prototype to *Nemesis* to verify that this does indeed no longer exhibit a blocking behavior when evolving the database schema. We then also verified that this prototype is able to deal with the complexity of changesets produced in practice, and finally suggested a future direction to take this approach to.

The contributions of this thesis are the tools *Nemesis*, our measurements of various versions of PostgreSQL and MySQL using *Nemesis*, *QuantumDB* which is able to evolve the database schema without causing downtime, and empirical evidence that *QuantumDB* works in practice.

Given these contributions, we anticipate a bright future for the use and adoption of *Continuous Deployment* in zero-downtime web services. We are convinced that by further development of *QuantumDB* we are able to make *Continuous Deployment* more attainable for practitioners, without having to incur downtime whenever the database schema needs to be altered.

Bibliography

[1] Large Hadron Migrator. https://github.com/soundcloud/lhm, June 2014.

[2] Nemesis. https://github.com/quantumdb/nemesis, June 2014.

[3] Online Schema Change for MySQL. https://www.facebook.com/note.php?note_id=430801045932, June 2014.

[4] Percona Toolkit - pt-online-schema-change - foreign keys. http://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html#cmdoption-pt-online-schema-change--alter-foreign-keys-method, June 2014.

[5] Percona Toolkit for MySQL is a collection of advanced command-line tools (which our own Percona MySQL Support staff uses) to perform a variety of MySQL server and system tasks that are too difficult or complex to perform manually. http://www.percona.com/software/percona-toolkit, June 2014.

[6] TableMigrator. https://github.com/freels/table_migrator, June 2014.

[7] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. "Cut Me Some Slack": Latency-aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 432–443, New York, NY, USA, 2012. ACM.

[8] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data conversion for xml and sql. In *Practical Aspects of Declarative Languages*, pages 290–304. Springer, 2007.

[9] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Information technology. Morgan Kaufmann Publishers, 1995.

[10] Anthony Cleve and Jean-Luc Hainaut. Co-transformations in database applications evolution. In *Generative and Transformational Techniques in Software Engineering*, pages 409–421. Springer, 2006.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[12] Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. Automating Database Schema Evolution in Information System Upgrades. In *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM.

[13] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful Database Schema Evolution: The PRISM Workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

[14] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *Proc. VLDB Endow.*, 4(2):117–128, November 2010.

[15] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[16] Tudor Dumitraş and Priya Narasimhan. Why Do Upgrades Fail and What Can We Do About It?: Toward Dependable, Online Upgrades in Enterprise System. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 18:1–18:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[17] Tudor Dumitras and Priya Narasimhan. No downtime for data conversions: Rethinking hot upgrades. Technical report, Technical Report CMU-PDL-09-106, Carnegie Mellon University, 2009.

[18] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] J-L Hainaut, Vincent Englebert, Jean Henrard, J-M Hick, and Didier Roland. Database evolution: the db-main approach. In *Entity-Relationship ApproachâĂŤER'94 Business Modelling and Re-Engineering*, pages 112–131. Springer, 1994.

[21] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[22] Ziyang Liu, Bin He, Hui-I Hsiao, and Yi Chen. Efficient and Scalable Data Evolution with Column Oriented Databases. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 105–116, New York, NY, USA, 2011. ACM.

[23] Hyun J Moon, Carlo A Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proceedings of the VLDB Endowment*, 1(1):882–895, 2008.

[24] Hyun J Moon, Carlo A Curino, Myungwon Ham, and Carlo Zaniolo. PRIMA: archiving and querying historical data with evolving schemas. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1019–1022. ACM, 2009.

[25] Iulian Neamtiu, Jonathan Bardin, Md Reaz Uddin, Dien-Yen Lin, and Pamela Bhattacharya. Improving Cloud Availability with On-the-fly Schema Updates. 2013.

[26] Shlomi Noach. Openark kit, common utilities for MySQL. `https://code.google.com/p/openarkkit/`, June 2014.

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

[28] Dong Qiu, Bixin Li, and Zhendong Su. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 125–135, New York, NY, USA, 2013. ACM.

[29] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. Online, Asynchronous Schema Change in F1. *Proc. VLDB Endow.*, 6(11):1045–1056, August 2013.

[30] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 53–64, New York, NY, USA, 2013. ACM.

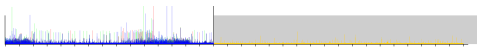
[31] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013.

Appendix A

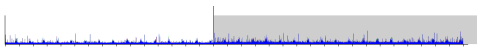
Nemesis results

In this Appendix, we present all the graphs produced by *Nemesis* based on measurements taken while benchmarking each evolution scenario and database combination. In each section we will cover one evolution scenario, and present the associated graphs for each database (whenever we were able to execute this scenario for that particular database).

A.1 S1: Adding a non-nullable column



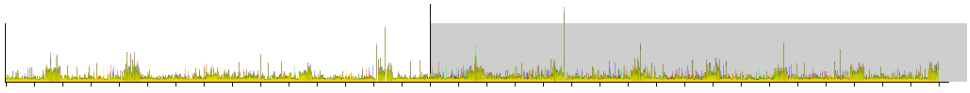
(a) MySQL 5.5



(b) MySQL 5.6

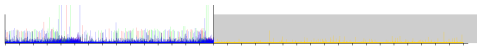
(c) PostgreSQL 9.3

(d) PostgreSQL 9.4



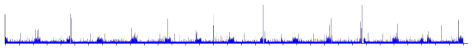
(e) QuantumDB on PostgreSQL 9.4

A.2 S2: Adding a nullable column



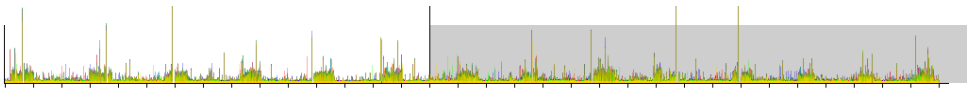
(a) MySQL 5.5

(b) MySQL 5.6



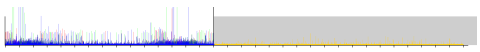
(c) PostgreSQL 9.3

(d) PostgreSQL 9.4



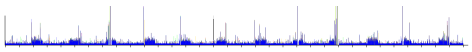
(e) QuantumDB on PostgreSQL 9.4

A.3 S3: Renaming a non-nullable column

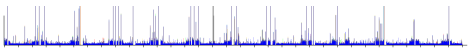


(a) MySQL 5.5

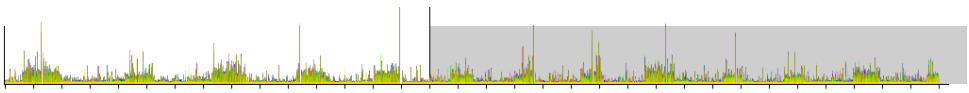
(b) MySQL 5.6



(c) PostgreSQL 9.3



(d) PostgreSQL 9.4



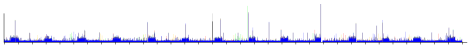
(e) QuantumDB on PostgreSQL 9.4

A.4 S4: Renaming a nullable column

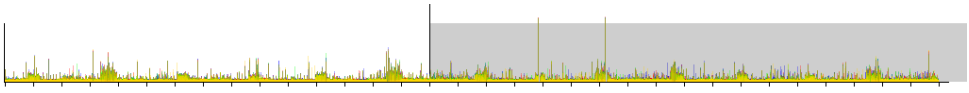
(a) MySQL 5.5

(b) MySQL 5.6

(c) PostgreSQL 9.3

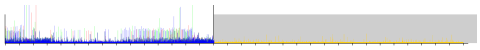


(d) PostgreSQL 9.4



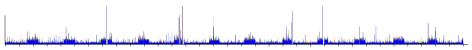
(e) QuantumDB on PostgreSQL 9.4

A.5 S5: Dropping a non-nullable column

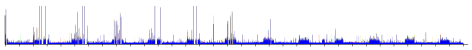


(a) MySQL 5.5

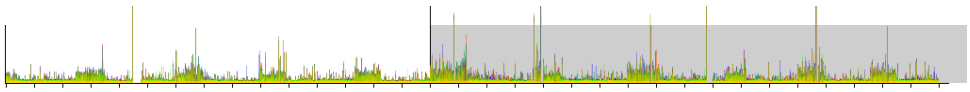
(b) MySQL 5.6



(c) PostgreSQL 9.3



(d) PostgreSQL 9.4

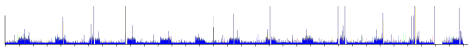


(e) QuantumDB on PostgreSQL 9.4

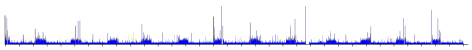
A.6 S6: Dropping a nullable column

(a) MySQL 5.5

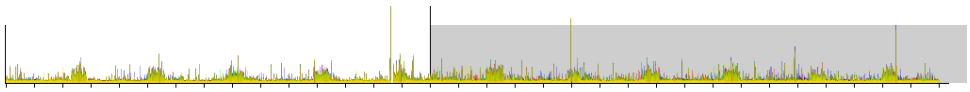
(b) MySQL 5.6



(c) PostgreSQL 9.3

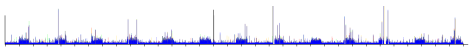


(d) PostgreSQL 9.4

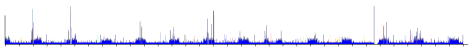


(e) QuantumDB on PostgreSQL 9.4

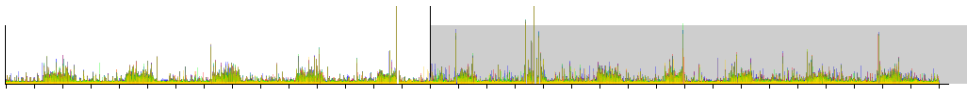
A.7 S7: Modifying the datatype of a non-nullable column



(a) PostgreSQL 9.3

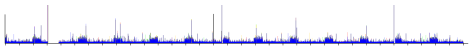


(b) PostgreSQL 9.4



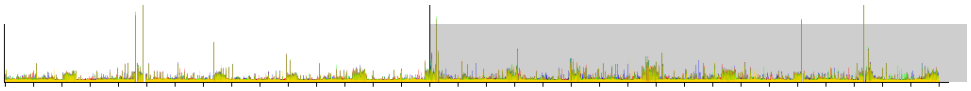
(c) QuantumDB on PostgreSQL 9.4

A.8 S8: Modifying the datatype of a nullable column



(a) PostgreSQL 9.3

(b) PostgreSQL 9.4

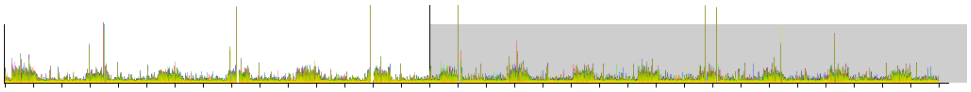


(c) QuantumDB on PostgreSQL 9.4

A.9 S9: Modifying the datatype of a column from integer to text

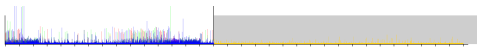
(a) PostgreSQL 9.3

(b) PostgreSQL 9.4

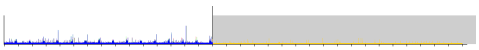


(c) QuantumDB on PostgreSQL 9.4

A.10 S10: Making a column nullable

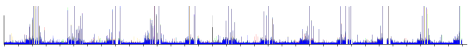


(a) MySQL 5.5

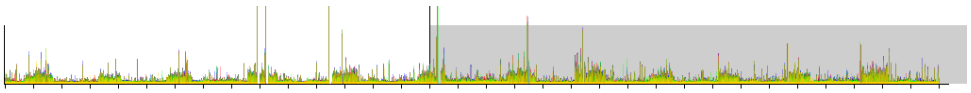


(b) MySQL 5.6

(c) PostgreSQL 9.3

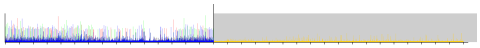


(d) PostgreSQL 9.4

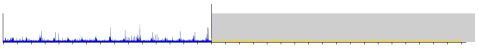


(e) QuantumDB on PostgreSQL 9.4

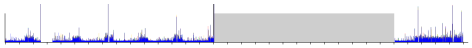
A.11 S11: Making a column non-nullable



(a) MySQL 5.5

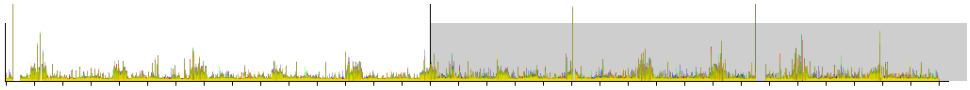


(b) MySQL 5.6



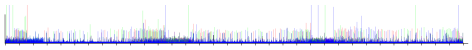
(c) PostgreSQL 9.3

(d) PostgreSQL 9.4



(e) QuantumDB on PostgreSQL 9.4

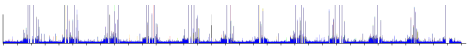
A.12 S12: Modifying the default value of a non-nullable column



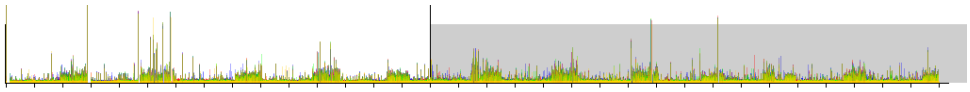
(a) MySQL 5.5

(b) MySQL 5.6

(c) PostgreSQL 9.3

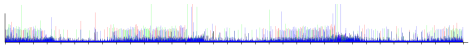


(d) PostgreSQL 9.4



(e) QuantumDB on PostgreSQL 9.4

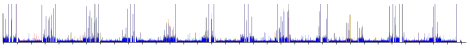
A.13 S13: Modifying the default value of a nullable column



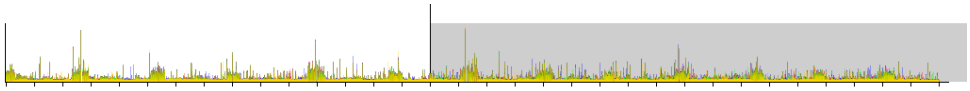
(a) MySQL 5.5

(b) MySQL 5.6

(c) PostgreSQL 9.3

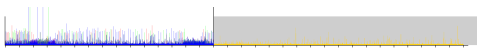


(d) PostgreSQL 9.4



(e) QuantumDB on PostgreSQL 9.4

A.14 S14: Creating a foreign key constraint on a non-nullable column

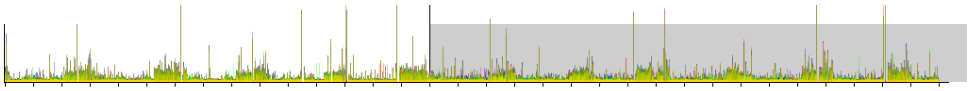


(a) MySQL 5.5

(b) MySQL 5.6

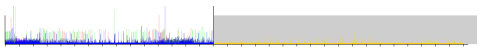
(c) PostgreSQL 9.3

(d) PostgreSQL 9.4

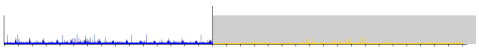


(e) QuantumDB on PostgreSQL 9.4

A.15 S15: Creating a foreign key constraint on a nullable column

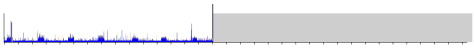


(a) MySQL 5.5

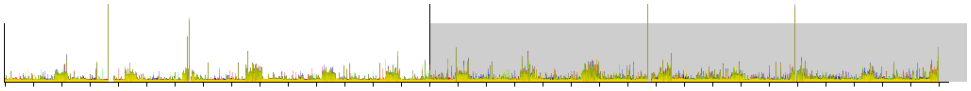


(b) MySQL 5.6

(c) PostgreSQL 9.3

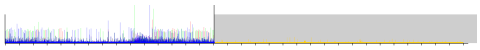


(d) PostgreSQL 9.4

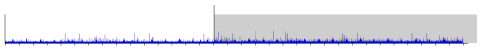


(e) QuantumDB on PostgreSQL 9.4

A.16 S16: Creating an index on an existing non-nullable column



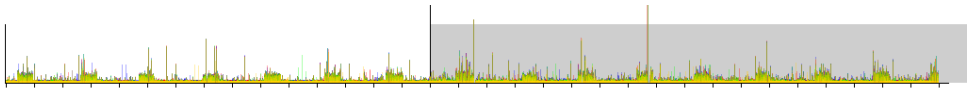
(a) MySQL 5.5



(b) MySQL 5.6

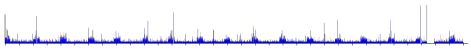
(c) PostgreSQL 9.3

(d) PostgreSQL 9.4

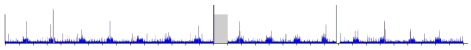


(e) QuantumDB on PostgreSQL 9.4

A.17 S17: Renaming an existing index

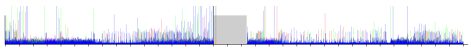


(a) PostgreSQL 9.3



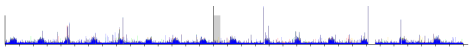
(b) PostgreSQL 9.4

A.18 S18: Dropping an existing index

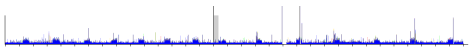


(a) MySQL 5.5

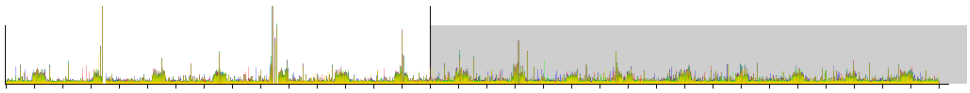
(b) MySQL 5.6



(c) PostgreSQL 9.3



(d) PostgreSQL 9.4



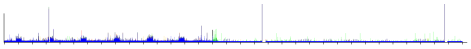
(e) QuantumDB on PostgreSQL 9.4

A.19 S19: Renaming an existing table

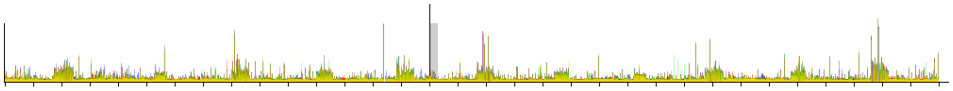
(a) MySQL 5.5

(b) MySQL 5.6

(c) PostgreSQL 9.3



(d) PostgreSQL 9.4



(e) QuantumDB on PostgreSQL 9.4