Delft University of Technology
Software Engineering Research Group
Technical Report Series

Continuous Deployment and Schema Evolution in SQL Databases

Michael de Jong and Arie van Deursen

Report TUD-SERG-2015-013

TU Delft

SERG

TUD-SERG-2015-013

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of RELENG 2015, the Third International Workshop on Release Engineering. ACM, 2015.

© 2015 ACM. Personal use of this material is permitted. Permission from ACM must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Continuous Deployment and Schema Evolution in SQL Databases

Michael de Jong*, Arie van Deursen†
Delft University of Technology,
Delft, The Netherlands

Email: M.deJong-2@student.tudelft.nl*
Email: Arie.vanDeursen@tudelft.nl†

January 29, 2015

***Abstract*—Continuous Deployment is an important enabler of rapid delivery of business value and early end user feedback. While frequent code deployment is well understood, the impact of frequent change on persistent data is less understood and supported. SQL schema evolutions in particular can make it expensive to deploy a new version, and may even lead to downtime if schema changes can only be applied by blocking operations.**

In this paper we study the problem of continuous deployment in the presence of database schema evolution in more detail. We identify a number of shortcomings to existing solutions and tools, mostly related to avoidable downtime and support for foreign keys. We propose a novel approach to address these problems, and provide an open source implementation. Initial evaluation suggests the approach is effective and sufficiently efficient.

I. Introduction

SQL databases are an integral part of many (web) applications. They provide strong guarantees about the storage and retrieval of persistent data which makes them useful for various software applications.

For release engineers, the persistence of the data and especially the structure of this data as embedded in the application's SQL schema is at odds with today's need for continuous deployment. Challenges include combining zero downtime constraints with evolving database schemas (affecting tables, columns, foreign keys, constraints, etc.), and the need to be able to rollback changes without loss of data.

We can distill the challenge of continuous delivery in the presence of schema evolution into two separate problems:

First, given the deployment of a new version of a (web) application, depending on the deployment strategy it might be required that both the old and the new version of the application run in parallel (*Rolling Upgrades*, or *Canary Releases* [2]). This means that the schema on the SQL database needs to be compatible with the schema expected by both the old and the new version of the application.

Second, operations which modify the database schema in a SQL database (*DDL statements*) can be either non-blocking or blocking in nature, depending on the implementation of the operations in that specific SQL database. In case of non-blocking *DDL statements*, SQL queries issued by the application may proceed, whereas with blocking *DDL statements* these queries may be put on hold for the duration of the operation. This disruption can range from milliseconds to hours depending on the schema operation, the size of the affected table(s), and the database implementation. One such example would be adding a new non-nullable column to an existing table in a PostgreSQL database. This operation is blocking in nature, preventing other queries from running while it is being executed. When applied to a table of 50 million records (see Section V-B), the queries from the application can be blocked for 2.5 minutes.

When service windows are short or even non-existent, these two problems make releasing new versions of applications frequently problematic for release engineers. To address this problem we aim to find or implement a tool which allows its users to not only alter their database schema in a non-blocking fashion, but also do this without affecting older applications still operating on the older version of the schema.

II. State of the Art

A. Continuous Delivery

We first examine existing approaches, tools, and research aiming to solve these problems. Our starting point is the seminal book "Continuous Delivery" [2]. This book describes three high-level approaches to deal with migrating data without loss of data or requiring downtime:

- **Approach 1**: Copy the database or schema and start recording transactions. When the copying has completed, apply the schema changes to the copied database or schema. Finally replay all the recorded database transactions on the copied database or schema to reach an equivalent state. Once this state has been achieved, it must either be maintained by continuing to record and playback transactions, or make the switch from the old to the new database or schema.
- **Approach 2**: When using *Blue-Green deployments*, two instances of the database are available. Take the inactive database and apply the schema operations to it. In the mean time these two databases need to be kept synchronized. This can be done by using approach 1, or somehow

ensure that the table under change remains in a read-only state until the active and inactive databases have switched roles.

- **Approach 3**: The last is an approach where schema operations are performed separately from the application deployments using operations which are non-blocking. As a consequence the application must be able to deal with either the old or the new database schema being in use, and construct SQL queries which can operate correctly on both schemas.

B. Current Tooling

Since evolving the schema of a database and coordinating the deployment of software are both complex tasks, we looked for tooling which could help automate this for software and release engineers.

1) Percona Toolkit [6] & Openark Kit [4]: These are tools designed to help DBAs perform administrative tasks on MySQL databases. The commands *oak-online-alter-table* and *pt-online-schema-change* create an empty copy of the a specified table (a ghost table) and apply schema operations to the ghost table. These commands then proceed to install triggers on both tables, which insert, update, or delete the corresponding record in the other table when a record in their own table is inserted, updated, or deleted. A background task iterates over all records in the original table (in batches) and lazily copies records from the original table to the ghost table. When all records have been copied the original table and ghost table switch names atomically, ensuring that clients now use the ghost table when querying the original table name.

2) TableMigrator [9]: This is a tool which works like Openark Kit and Percona Toolkit. Instead of using triggers to synchronize the two tables with each other, TableMigrator requires tables to have an additional column which stores the date and time that record was last modified. Using this information corresponding records in both tables are synchronized with each other by passing over all records in multiple passes. Once a pass can be done relatively quickly, TableMigrator acquires an exclusive write lock on the original table — blocking the entire table — and proceeds to do a final pass. Once this has been done, it atomically swaps the names of the original and ghost table before releasing the write lock. Since this final pass can be done quickly, any application executing queries on that table is only very briefly blocked.

3) SoundCloud's Large Hadron Migrator [3] & Facebook's Online Schema Change [5]: These are both tools which closely follow Approach 1 described earlier (Section II-A). They create a copy of a table (including its contents) and record modifications to records in the original table using triggers into a special "deltas" table. Once they have copied the original table and its contents, they apply the specified schema operations to the ghost table before proceeding to replay the recorded modifications from the "deltas" table onto the ghost table. When the "deltas" table is almost empty, these tools acquire an exclusive write lock on the original table — like *TableMigrator* does — and apply the last of the recorded modifications before atomically switch the names of both tables.

C. Research

1) Imago: Imago [1] is a tool which follows Approach 2. With Imago there are two production environments, called "universes". Imago integrates with the application and its environment at both the ingress point (load balancer) to be able to switch users from one "universe" to the other, and at the egress point where the (web) application sends queries to the SQL database. When a new version of the application (requiring a change to the database schema) needs to be deployed, Imago bootstraps the inactive production environment with a copy of the active database, applies the required schema operations and then copies over the data from the old universe to the new universe. When this process is almost complete, it enforces a short period of read-only access to the SQL database, while it is switching users from the old "universe" to the new "universe" using the load balancer. Imago makes no claims in regards to its support for foreign keys.

D. Limitations

- None of the tools used in practice support foreign keys.
- The tools primarily focus on MySQL databases.
- These tools are limited in the sense that they typically only support making small changes to the existing database schema. This forces software and release engineers to often release intermediate versions of the application, as the schema is altered in several steps.

III. Proposed Approach

A. Vision

We feel that the current situation is an impediment to the adoption of *Continuous Deployment*. Although *Continuous Deployment* has made great improvements on workflows, team dynamics, and tooling for deploying and monitoring applications, the area of schema evolution in databases has received little attention. Therefore we aim to address these problems by developing a new tool which adheres to the following principles:

- A solution should not put restrictions on how to deploy applications, or which methods of deployment can be used: *Blue-Green deployments*, *Rolling upgrades*, *Canary releases*, etc.
- Clients connected to the database must be able to access the contents of the database according to an active version of the database schema of their choosing. Their "view" should only contain tables, columns, foreign keys, etc which are present in their chosen version of the database schema, but contain the same data as is present in all other "views".
- Versioning and rolling back are important for release engineers. A solution should support versioning the changes being made to the database schema, and be able to rollback to a previous version if the need arises.

- Foreign keys are an integral part of SQL databases and should be supported.

B. Solution Direction

The previously discussed tools from practice all use Approach 1 or a variation of it. They prove that this is a viable way for simple evolutions of the database schema. However where these tools all create a single ghost table, we aim to create one ghost table for each table that is under change, and ghost tables for all tables which in turn are dependent on these tables under change. For instance adding a new non-nullable column "summary" to the "books" table (see Figure 1), means we should also create a ghost table for the "publications" table since there is a foreign key pointing from the "publications" table to the "books" table. The other tables do not require ghost tables, as they do not have any foreign keys referencing either the "books" or "publications" table. This approach ensures that we can support foreign keys.

In addition, the tools from practice all allow release engineers to switch the table names manually after the the original and the ghost table have been synchronized. This allows for two or more versions of the application to co-exist on the same database as long as each version correctly queries the correct (ghost) table. Although the current tools from practice do not solve this specific problem, we think that this is easily solved using a small wrapper around database drivers for most programming languages.

Finally, with this approach it is possible to perform several schema evolutions at once. In the example, we could for instance also add even more columns to either the "books" or "publications" table.

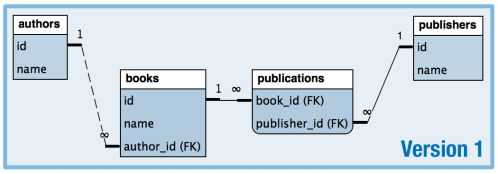
IV. Implementation Status

A. Current Status

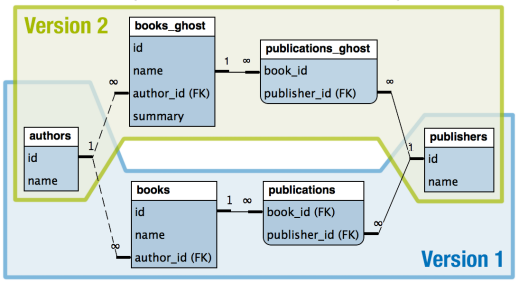
We are in the process of implementing and experimenting with the approach just described via a new tool that we have named QuantumDB [7]. The aim is to make *Continuous Deployment* more attainable for software and release engineers for applications relying on SQL databases. Although not yet ready for production environments, we have implemented enough to perform a quick evaluation in a small experiment.

QuantumDB currently consists of two parts:

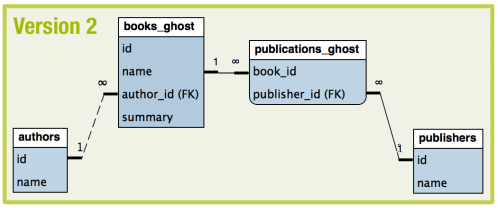
- **qdb:core** is the program responsible for performing the actual migration tasks on the database. It does this by creating ghost tables of tables under change, using triggers to keep the ghost tables and the original tables synchronized with each other, and a background task which copies the data from the original tables to the ghost tables in small batches to prevent degrading the performance of the database.
- **qdb:driver** is a Java JDBC driver which can wrap another JDBC driver. When used, the software engineers can specify a database schema version on which that version of their application operates. The driver then rewrites SQL queries sent to it by the application, and passes them on to the wrapped database JDBC driver which in turn sends them to the database. When rewriting queries, it replaces table names with the names of the associated (ghost) table names where applicable.



(a) Original database schema, before migration



(b) Database schema in mixed-state, during migration



(c) New database schema, after migration

Fig. 1: Query performance of demo application before and during migration

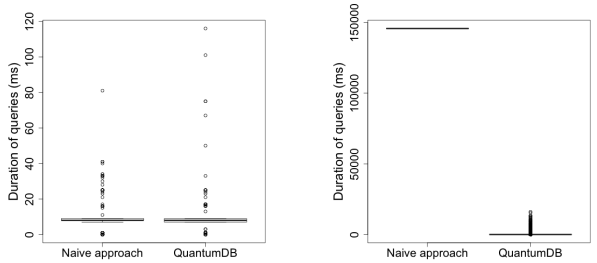
With QuantumDB we can currently take one of the active database schemas, apply an arbitrary number of schema operations to it, and expose the resulting database schema in addition to the original database schema. Once the last application connected to a particular version of the database schema disconnects, the tables associated with that database schema version can be safely discarded.

V. Initial Measurements

In order to validate our work at an early stage we have setup a small experiment to verify that the proposed approach and our implementation of it behaves as expected.

A. Setup

We have installed PostgreSQL 9.3 and a small test application — made available on GitHub [8] — on a server with a 4-core Intel Core i7-3770, a 1TB hard disk, and 16GB memory. The test application inserts 50 million records into a single table called "users" and starts a simple demo application with two threads which continuously query and manipulate the data in that table using SELECT, INSERT, UPDATE, and DELETE statements. It then uses one of two methods to add a non-nullable column to the "users" table. During this process we



(a) Before migration (b) During migration

Fig. 2: Query performance of demo application before and during migration

keep track of the duration of the queries issued by the demo application and how long each method takes to complete.

The "naive" approach simply executes an "ALTER TABLE users ADD COLUMN" statement to add the new column to the database. Once this statement has completed, we can start a new demo application which can operate on the new structure of the "users" table. Since the new structure of the "users" table is compatible with both the old and the new version of the demo application they can run in parallel to each other, allowing for any kind of deployment method to be used.

The second approach uses QuantumDB to add the new column. It first creates a ghost table based on the "users" table with the new column. It then iteratively copies records from the "users" table to the ghost table in batches of 2,000 records at intervals of 50 milliseconds. With this we attempt to avoid negatively affecting the performance of the queries issued by the demo application while we are copying the records. Unlike with the previous approach, with this approach the demo applications use the QuantumDB JDBC driver which exposes each demo application to a different version of the database schema (containing either the "users" table or the ghost table). The driver ensures that each query is rewritten in such a way that the correct table is queried for that specific version of the schema.

B. Results

The total migration time for the "naive" approach was 2 minutes and 26 seconds, during which the demo application was unable to read from or write to the database. The total migration time for the QuantumDB approach was 50 minutes and 41 seconds. Although significantly slower than the "naive" approach, the demo applications remained operational throughout this period (see Figure 2).

It is worth noting that we have not invested any time in improving the performance of QuantumDB or reducing the performance impact it has on the database and its clients. During this experiment we noticed that the hard disk was periodically unable to write data to disk quickly enough. As a result queries were taking significantly longer than normal to complete. We are confident that we can greatly reduce this by tweaking the PostgreSQL config, switching to a faster hard disk, or lowering the rate of migration during the copy phase.

VI. Road ahead

As stated before, work on QuantumDB is not finished. To be able to use QuantumDB effectively in production environments we still need to work on the following issues:

- QuantumDB can already deal with foreign keys referring from tables under change to other tables not under change. To fully support foreign keys QuantumDB would also need to be able to handle foreign keys referring to tables under change. This is currently being worked on.
- Use performance metrics to throttle the copy phase. When the database is (too) busy (high disk I/O, high CPU usage, or memory starvation), we should slow down the iterative copy process. Vice-versa we can speed it up when the database is not under load.
- QuantumDB currently only supports PostgreSQL, but since most SQL databases have support for triggers and procedures of some kind, this should prove easily extendable to other popular SQL databases.
- The driver currently uses a very rudimentary query rewriting algorithm to replace table names under change with the names of ghost tables. It does not yet support the entire PostgreSQL query syntax.

VII. Conclusion

Although development of QuantumDB is still ongoing, initial measurements are promising. With our small test application we have shown that we can migrate from one database schema to a state where both the old and the new database schemas are active, while still enabling database clients to remain operational. Applications can connect with the database according to a specific schema version, and no longer have to deal with database schemas being in an in-between state. This should effectively allow further decoupling of the release process from the development process. We believe that this approach can put *Continuous Deployment* and *Canary Releases* in reach of more software and release engineers.

References

[1] 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.
[2] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.
[3] Large Hadron Migrator. https://github.com/soundcloud/lhm, January 2015.
[4] Shlomi Noach. Openark kit, common utilities for MySQL. https://code.google.com/p/openarkkit/, January 2015.
[5] Online Schema Change for MySQL. https://www.facebook.com/note.php?note_id=430801045932, January 2015.
[6] Percona Toolkit for MySQL. http://www.percona.com/software/percona-toolkit, January 2015.
[7] QuantumDB. http://quantumdb.io, January 2015.
[8] QuantumDB demo for RelEng 2015. http://github.com/quantumdb/RelEng-demo, January 2015.
[9] TableMigrator. https://github.com/freels/table_migrator, January 2015.

TUD-SERG-2015-013
ISSN 1872-5392

SERG