TU Delft

Delft University of Technology

Zero-Downtime SQL Database Schema Evolution for Continuous Deployment

de Jong, Michael; Deursen, Arie van; Cleve, Anthony

DOI
10.1109/ICSE-SEIP.2017.5

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings of the 39th International Conference on Software Engineering (ICSE)

Citation (APA)
de Jong, M., Deursen, A. V., & Cleve, A. (2017). Zero-Downtime SQL Database Schema Evolution for Continuous Deployment. In *Proceedings of the 39th International Conference on Software Engineering (ICSE): Software Engineering in Practice (SEIP) Track* (pp. 143-152). IEEE. https://doi.org/10.1109/ICSE-SEIP.2017.5

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Zero-Downtime SQL Database Schema Evolution for Continuous Deployment

Michael de Jong, Arie van Deursen, and Anthony Cleve

Report TUD-SERG-2017-005

TU Delft

SERG

TUD-SERG-2017-005

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 at Software Engineering in Practice (SEIP) track of the ACM/IEEE International Conference on Software Engineering (ICSE), held in Buenos Aires, May 2017.

© 2017 IEEE. Personal use of this material is permitted. Permission from IEEE 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.

Zero-Downtime SQL Database Schema Evolution for Continuous Deployment

Michael de Jong[1], Arie van Deursen[2], Anthony Cleve[3]
[1]Magnet.me, The Netherlands
[2]Delft University of Technology, The Netherlands
[3]University of Namur, Belgium

***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 tool-supported approach 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 95 industrial evolution scenarios to our open source implementation of QuantumDB. These experiments demonstrate that *QuantumDB* realizes zero-downtime migrations at the cost of acceptable overhead, and is applicable in industrial continuous deployment contexts.**

I. Introduction

Continuous deployment is becoming an increasingly popular technique to roll out new features as fast as possible [2], [9], [12]. Advantages of continuous deployment include rapid feedback, fast bug fixing, as well as increased business value thanks to the earlier avaiability of functionality. For organizations adopting continuous deployment, it is not uncommon to have multiple releases per day or even per hour.

For web services and applications with 24/7 uptime demands, continuous deployment typically uses load balancers and rolling upgrades to incrementally roll out features over servers without loss of availability [8], [9].

Such zero-downtime deployments, however, are substantially harder when structural changes to the application's database need to be made – as also indicated by Claps *et al.* [2]. Database schema changes, such as adding a column or changing its type may, depending on the database management server (DBMS), lock the entire table in order to adjust all rows – which depending on the size of the table, and activity can take minutes if not hours. This means that any application or web service attempting to query a table under change will either block, appear unresponsive, or even (partially) fail.

To date, this problem is addressed in practice in one of two ways. The first is to accept downtime, and conduct the deployment in a low traffic time window (e.g. midnight or weekend). The second is to adopt the *Expand-Contract* pattern [9], and conduct a series of deployments that first extend the database, and later reduce it again. During these deployments, the system effectively works with multiple database schemas (a so-called *mixed-state*), for which the software engineers need to provide programmatic support. This method is further complicated when schema revisions encompass multiple schema change operations(as they often do [13]), which would require even more coordinated deployments.

Neither of these solutions is in the spirit of continuous deployment: they introduce extra downtime or significantly more effort from developers and system administrators, thus slowing down the deployment of new features. Since we can confidently state that schema changes are common based on both previous literature [13], [3], as well as our case study, we can also state that this is a serious impediment to the adoption of continuous deployment.

To remedy this, we propose an approach to support *fully automatic* schema evolution with *zero-downtime*. Our approach provides *mixed-state* for every schema changeset, by carefully maintaining a set of synchronized "ghost tables". The mixed-state is entirely transparent to the software engineer, who only needs to specify the required schema changes. Our approach is resilient against crashes, and safeguards referential integrity by taking foreign key constraints into account. An open source implementation of our approach called *QuantumDB* is available for download.

We evaluate the proposed approach by means of 19 synthetic schema changes that we apply to a database under load, as well as to a set of around 95 schema changes from a commercial application. Our evaluation demonstrates that the approach can handle real life change scenarios without downtime for medium-sized databases (hundreds of columns, millions of records).

We start our paper by providing the required background in continuous deployment (Section II). We then formulate seven requirements a solution to the problem of zero-downtime schema evolution should adhere to (Section III). As a prelude to our solution, we then empirically assess the blocking nature of schema evolution operators in two common DBMSs, PostgreSQL and MySQL (Section IV). Equipped with that

knowledge, we describe in detail how to use ghost tables to offer mixed-state in a transparent manner (Section V). We then provide our evaluation, a discussion, a summary of related work, and our conclusions in Sections VI–IX.

Some initial ideas in this paper have been presented at the Release Engineering workshop held during ICSE 2015 [7]. The present paper provides a full description of the approach, a quantitative evaluation, and an industrial evaluation.

II. Background

In *Continuous Deployment* the web service must remain online and available to its clients while being replaced with a newer version. Thus, during a redeployment:

- The service must be available at all times;
- Clients using the service must be atomically switched from the old to the new version;
- Once switched, a client should not be able anymore to interact with the old version of the web service, unless an explicit rollback has been performed.

The atomic switch is typically conducted using load balancers, redirecting incoming requests to a specific server. Requests can be redirected based on the workload of each server, some property of the request, or some application-specific rules.

There are two important approaches for redeploying a web service without downtime using a load balancer [1]. With the *Rolling Upgrade* method it is assumed that there are several servers, each running an instance of the web service, and a load balancer distributing incoming user requests over these web service instances. Each web service instance is upgraded, one at a time, using the following steps:

1) Instruct the load balancer that the server is unavailable. No new 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 instance is once again available and may receive user requests again.

A consequence of this method is that the full system will be in a *Mixed-State*, in which both the old and the new version of the web service are processing user requests at the same time. Thus, the web service instances and other external systems need to be able to handle such a *Mixed-State*.

The alternative method is *Big-Flip*. As opposed to a *Rolling upgrade*, *Big-Flip* attempts to make the atomic switch *all at once*. In this case there are two separate server pools of equal size, all running web service instances. Both pools must have enough capacity and resources to handle all the incoming requests expected during the redeployment period. Of these two pools only one is actively handling requests while the other pool is idle. When the web service is redeployed, the following steps are performed:

1) Update the web service instances in the idle pool.
2) Start the web service instances in 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 new client requests, while the active pool becomes the inactive pool and stops handling requests.

As with the previous approach, there is a short period of time where the system is put in a *Mixed-State*. From the moment when the initial idle pool starts up new versions of the web service until the moment when the initially active pool shuts down the older web service instances, both versions are running side-by-side. Thus, also with *Big-Flip*, all web service dependencies must be able to deal with this *Mixed-State*.

For the deployment of evolving database schemas specifically, Humble et al [9] describe two techniques: *Blue-Green Deployments* and *Expand-Contract*. *Blue-Green Deployments* are essentially the *Big-Flip* approach applied to the database server and schema where the application switches from using one database server to another.

With *Expand-Contract* the database schema is modified through incremental steps using non-blocking *DDL* statements, while maintaining compatibility with both older and newer versions of the web service in order to support a *Mixed-State*.

III. Problem Statement

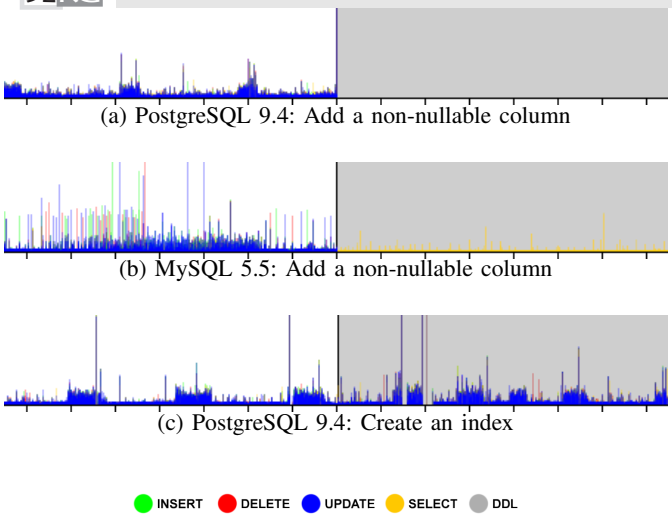
We distinguish three key challenges when using SQL databases in a *Continuous Deployment* environment.

The first challenge is that some schema change operations (*DDL* statements) are blocking in nature. Depending on the SQL database platform, a *DDL* statement may block other queries from being executed through the use of a table lock, preventing the web service from accessing and manipulating the data stored in the table under change. The scope of this problem will be further studied in Section IV.

The second challenge relates to the concept of *Mixed-State*. When deploying a new version of a web service, two different versions of the same web service may have to be active at the same time. In that case, the older version will still expect the database to be at one version of the schema, while the newer version will expect the database to be at a different version of the schema. Although the schema is explicitly defined in SQL databases, many web services assume that the database is of a certain structure. If this assumed schema changes at runtime, there are typically no provisions in web services to deal with these changes. This places limitations on how much you can change and which deployment methods can be used.

The final challenge consists in preserving foreign key constraints during schema evolution. Such constraints are widely used, and any schema evolution tool should also preserve foreign key constraints. Existing approaches and tools (discussed in Section VIII) either have no support, or insufficient support for these constraints. In the latter case, foreign key constraints are temporarily disabled, or dropped and recreated later on, meaning that in the meantime client applications are free to violate referential integrity.

Given these challenges we formulate the following requirements for a solution to *zero-downtime* schema evolution:



(a) PostgreSQL 9.4: Add a non-nullable column

(b) MySQL 5.5: Add a non-nullable column

(c) PostgreSQL 9.4: Create an index

Fig. 1: Blocking behavior before (left) and after (right, grayed) executing DDL statements.

R1: *Non-Blocking Schema Changes*. Changing the schema should not block queries issued by any database client.

R2: *Schema Changesets*. It should be possible to make several non-trivial changes to the database schema in one go. This prevents software engineers from having to develop and deploy intermediate versions of the web service.

R3: *Concurrently Active Schemas*. Multiple database schemas should be able to be "active" at the same time, to ensure that different versions of the web service can access the data stored in the database according to their own chosen schema. This avoids putting restrictions on the method of deployment when upgrading the web service.

R4: *Referential Integrity*. The solution must support both the migration and evolution of foreign key constraints during both normal use and while evolving the database schema. These constraints should be enforced at all times.

R5: *Schema Isolation*. Any changes made to the database schema should be isolated from the database clients. In other words, any client should not see any other database schema other than the version it relies on.

R6: *Non-Invasiveness*. Any integration with the application should require as little change to the source code as possible.

R7: *Resilience*. The solution must ensure that the data stored in the database always remains in a consistent state. In other words, when the migration fails, it must be possible to rollback the changes and return to a consistent state without affecting the database clients.

IV. Measuring the Blocking Behavior of Schema Changes

A prerequisite to solving the problem of *zero-downtime* schema evolution is a thorough understanding of the blocking behavior of DDL operators (our requirement **R1**).

Some DDL operators, such as modifying the type of a column, may involve modifications to all rows, for example to convert a string value into an equivalent integer value. Some DBMS implementations acquire a read or write lock during this operation. With potentially millions of rows, such DDL operations may take substantial time to complete.

The precise locking behavior, however, not only differs per DBMS (e.g., PostgreSQL and MySQL have adopted different locking strategies), but also per version of the DBMS[1].

Therefore, in this section we propose an approach allowing the empirical analysis of the blocking behavior of DDL schema evolution operators, use this approach to establish the blocking behavior of the three most recent versions of MySQL[2] and PostgreSQL[3], and finally (Section VI) use the same approach to evaluate the non-blocking behavior of our solution for *zero-downtime* schema evolution.

A. Experimental Setup

To conduct our experiments, we simulate an application which continuously queries a SQL database, while performing various *DDL* statements used for schema evolution:

1) Prepare a "users" table with 50 million random records.
2) Simulate an application operating on the database by spawning a number of threads each performing INSERT, SELECT, UPDATE, and DELETE queries to the "users" table. For each query, the start and end times are logged. We used 6 threads – 2 for SELECT, 2 for UPDATE, 1 for INSERT and 1 for DELETE.
3) Perform one of the *DDL* statements, and log when it started and finished.
4) When this statement completes, restore the database to its original database schema, and proceed to test the next schema operation using steps 2 through 4.

By having a sizeable dataset in the table under change, and stressing the database by continuously querying the table using multiple threads, the schema evolution process is slowed down. By recording the start time and end time of each query and then graphing this, we can visualize if altering the table blocks the queries issued by the simulated application. This allows us to determine which *DDL* statements are blocking, and if so for how long, and which type of *DML* queries they block.

The actual *DDL* statements come from a set of 19 typical schema evolution scenarios, composed of common DDL operations that we expected to be possibly blocking (based on our experience with these systems).

We run our experiments on the three most recent versions of both MySQL and PostgreSQL, giving six result sets in total. For each database/scenario combination we first run the queries for one minute, and then apply the scenario. Depending on the scenario and database, some scenarios may take just a few seconds, while others may take hours to complete.

[1] http://www.postgresql.org/docs/9.5/static/release-9-5.html
[2] https://www.mysql.com/
[3] http://www.postgresql.org/

	MySQL			PostgreSQL		
	5.5	5.6	5.7	9.3	9.4	9.5
Non-blocking	3	11	13	14	13	14
Read only	11	4	3	0	0	2
Blocking	1	0	0	5	6	3
Not applicable	4	4	3	0	0	0

TABLE I: Blocking Scenarios in different databases.

We ran all series of experiments on a hosted Virtual Machine featuring 8 CPU cores at 2.40GHz, 16 GB memory, and a 160 GB SSD for storage.

To present our results, we use graphs as shown in Figure 1. Each vertical spike represents a *DML* query. The height of each spike represents the duration of the query, and its position on the horizontal axis represents its starting time. The black spike in the center of the graph is the start of the DDL statement, followed by the grey area marking the duration of the statement. To permit easy comparison of these plots, the plots just show the 7 seconds before and 7 seconds after executing the *DDL* statement.

B. Results

Conducting all experiments took a total of approximately 4 days of compute time. We discuss three typical scenarios in more detail, after which we present our overall findings.

Figure 1a shows the scenario of adding a new column to the "users" table with a NOT NULL constraint. We can clearly see that in the case for PostgreSQL 9.4 this statement blocks all queries issued by the simulated application.

When repeating the same scenario for MySQL 5.5, we see slightly different results (Figure 1b). While the *DDL* statement is being executed, all the queries issued by the simulated application are blocked (as in the previous case), with the exception of (yellow) SELECT statements. In other words, this particular table is put in a read-only state. When we examine the results for creating an index on an existing column (Figure 1c), we see that none of the queries issued by the simulated application are blocked by this *DDL* statement.

The overall outcomes for all scenarios are shown in Table I. Observe that in PostgreSQL one of the operators that used to be *non*-blocking (rename an index in 9.3) changed into a *blocking* operator in 9.4. In 9.5 this was reversed again.

Overall, we conclude that blocking behavior differs substantially both per DBMS and per version of the DBMS. Furthermore, we observed in one case that behavior that used to be non-blocking becomes blocking in a later release.

V. Transparent Schema Evolution with Ghost Tables

In this section, we propose an approach, dubbed *QuantumDB*, to evolve a database schema without downtime, meeting all requirements **R1-R7** formulated in Section III. Our approach provides developers with *non-blocking* schema changes, *despite* the presence of blocking DDL statements identified in Section IV. Furthermore, the approach provides *Mixed-State* in a way that is transparent to the developer. This permits the adoption of *Rolling Upgrades* (see Section II), as well as the seamless deployment of multiple versions of a service with different database schemas.

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

Fig. 2: Change set adding a new "referred_by" column with a foreign key to "customers" table.

Fig. 3: Representation of the change set of Figure 2.

The approach relies on the temporary use of several active schema versions. This entails creating ghost tables for those tables directly or indirectly affected by each schema change.

In this section, we explain when and how to construct ghost tables; how to keep ghost tables and their originals in sync; and how to drop the original tables once the transition to the new schema is completed. We also explain how the approach is resistant to system crashes, and transparent to the developer.

A. Schema Versioning

To reason about schema changes and their effects, we represent these changes as a logical sequence of basic operations which alter the database schema sequentially. As an example, Figure 2 shows a changeset which adds a new "referred_by" column with a foreign key constraint to an already existing "customers" table.

Each changeset is a series of schema changes which are applied to a certain database schema (Figure 3). Each version of the database schema is labeled with a unique ID, which allows us later to identify a particular version of the schema.

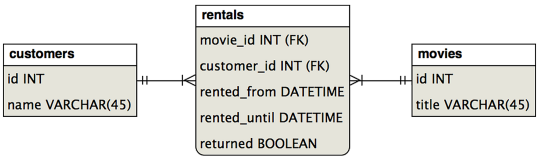
B. Mixed-State with Ghost Tables

Our approach embraces *Mixed-State* by construction (requirement **R3**). This means that the database may contain several active database schemas at any time, yet this *Mixed-State* is abstracted away from database clients.

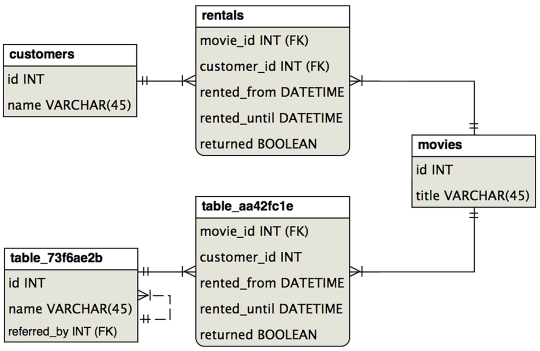
To illustrate how we achieve *Mixed-State* we execute the changeset from Figure 2 on a simple database schema containing three tables as shown in Figure 4a.

1) Creating Ghost Tables: Given a changeset, the first step is to create new ghost tables for the tables affected by the schema modifications. In our example, the **customers** table is the table directly under change, so we create a new ghost table from that table.

To support foreign key constraints (requirement **R4**), we also mirror tables that depend, directly or indirectly, on any of the tables subject to change. Thus, besides the **customers**



(a) Database schema at version **82fba53**.



(b) *Mixed-State* of versions **82fba53** and **80bfa11**.

Fig. 4: Before and after achiving *Mixed-State*

table, we mirror the **rentals** table, as it has a dependency on the **customers** table through a foreign key constraint.

Figure 4b shows the end result for our example schema after applying the changeset from Figure 2. Note that the tables that are not affected, e.g., the **movies** table, are not mirrored.

Since we cannot use the same table name twice, we assign unique names to every ghost table created. Furthermore, for every ghost table, we keep track of the mapping between the original tables and columns and the created ghost tables and columns. This mapping allows us to create ghost tables from a sequence of changesets, according to requirement **R2**.

2) Constructing Forward Triggers: Having achieved *Mixed-State* structurally, we fill the ghost tables with data from the original tables. Since we can determine that table **table_73f6ae2b** is based on table **customers** and that **table_aa42fc1e** is based on table **rentals**, we can copy the data from the source tables to their respective ghost tables.

During this copying the database is still in use, as we want to avoid locking (requirement **R1**). Thus, the records in these source tables might change while the data are being copied from a source to a ghost table. To address this, we adopt a *forward* database trigger on each source table so that whenever a database client inserts, modifies, or deletes records in the source table, these changes are propagated to the ghost table.

Thus, for our running example, if a new record is *inserted* into the original **customers** table, that same data will be inserted into **table_73f6ae2b** by the trigger. Since this ghost table differs structurally from its source table, the trigger must account for this. Through the column and table mappings we maintain, we can determine that one must insert the values of the id and name columns from the 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 value.

If an existing record in the **customers** table is *updated*, the database trigger needs to update the corresponding record in **table_73f6ae2b**. We copy the values of the id and name column of the updated record to the ghost table using an upsert — an 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 have the database trigger issue a simple DELETE 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 our example, 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.

3) Migrating Data: With the forward database triggers in place, we can copy data from the original source tables to the ghost tables. In doing so, we must account for structural differences between source and ghost tables.

In addition, some tables may contain large numbers of records. Therefore, we copy these records in small batches to avoid negatively affecting disk IO. Otherwise, we might need to access the disk continuously for a significant period of time, thereby reducing the performance of the database, and hence of the database clients.

We use an upsert to copy records from the source tables to their associated ghost tables. This, combined with the database triggers, ensures that all records in the source table have a matching and up-to-date record in the ghost tables.

4) Constructing Backward Triggers: Besides copying the data from the original tables to the ghost tables, we need to install *backward* database triggers on the ghost tables. These triggers have the same functionality as the forward triggers, but do the exact reverse. Whenever a record is inserted into, modified, or deleted from the ghost tables, the corresponding records are inserted into, modified, or deleted from the source tables. In our example the structural difference between the **customer** and **table_73f6ae2b** tables means that the value of the “referred_by” column is not copied to the source table.

5) Intercepting and Rewriting Database Queries: Once all data has been copied and the forward and backward triggers are in place, the database is in a *Mixed-State*. It can be used in either the original or the modified (ghost) schema. It contains the same data represented in two database schemas in the same database. Any change made to the contents of the source tables will be reflected in the associated ghost tables, and vice-versa.

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

Fig. 5: Table name mapping for the *Mixed-State* of versions **82fba53** and **80bfa11**

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

(a) Example query sent from the web service.

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

(b) Rewritten query sent to the database.

Fig. 6: Query transformation example

However, we want this (temporary) *Mixed-State* nature of the database to be fully transparent to the software engineers and the source code (requirement **R5**). Therefore, we introduce an additional layer between the application source code and the actual database driver. Like other approaches to intercept SQL queries executed by client web services [4], our approach relies on an abstraction layer positioned between the web service and the SQL database. In this way, the software engineer uses the normal table names, and our wrapper transforms the queries before they are sent to the database.

The wrapper uses the mapping from table names to ghost table names (shown in Figure 5) to rewrite subsequent SQL queries issued by the web service. Each query is parsed and the table names specified in the query are replaced with the table aliases applying for this database schema version. An example of rewritten query can be found in Figure 6b. The rewritten query is then sent to the SQL database, which will operate on the mirror tables instead of the original source tables. The query result is then relayed back to the web service, which is not aware that the query has been rewritten.

All the developer then needs to do is specify once which version of the database schema should be used. In our Java implementation of this approach, the software engineer can specify to which version the web service should connect in the JDBC connection url, as shown in Figure 7. This abstracts away the *Mixed-State* from the web service, and limits the invasiveness of the solution (requirement **R6**).

C. Transitioning out of 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. This allows us to deploy a new version of the web service, with any method of deployment. However, this state adds some 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 an 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 that is no longer in use after completing the deployment of a new version of the web service.

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

Fig. 7: Creating a database connection with *QuantumDB*.

1) Tracking Connected Clients: We need to know which versions of the database schema are still being used. Different approaches can be followed to achieve this. We propose to store meta-data for every connection that is created through the abstraction layer. This allows us to determine which connection operates on which specific version of the schema.

When we want to drop a certain version v of the database schema, we check whether there are active connections to the SQL database still using v or not. If this is the case, we terminate and return an error stating that we cannot drop this schema version since there are still some database clients that are using it. If this is not the case we can safely assume that it is no longer in use, and we can proceed with the next step.

2) Dropping a Database Schema: Once determined that it is safe to drop a particular version of the database schema, we must first identify which tables should be dropped. We do this by using the table name mapping (Figure 5). Assuming we want to drop version **82fba53**, we need to drop all the tables used by version **82fba53** that are not used by any other version. In our example this means we want to drop the **customers** and **rentals** tables, since they are both used in version **82fba53** but not in version **80bfa11**. We first drop all the database triggers which copy records to and from these tables, after which we can safely drop the two tables. From that point onwards the database is no longer in a *Mixed-State*.

3) Fault Tolerance: Should any part of this process be interrupted by the user or aborted because of a fault before the process has completed, the changes made so far with this approach will leave the newly created ghost tables and their respective database triggers and foreign keys behind in the database. Since it is impractical to make all changes in a single database transaction and rolling that transaction back, we need to have a slightly more elaborate way of rolling back from the failed *Mixed-State* back to the *Single-State* where only the original database schema is active. We can do this by dropping all ghost tables, foreign keys and database triggers which were created during this process. This is the exact same process as dropping a completed database schema which we described in the previous paragraph, and yields a database containing only the original active database schema. Since we never modify the schema of any of the original tables when attempting to achieve *Mixed-State*, we can simply drop newly created ghost tables, foreign keys, and database triggers when a fault occurs, providing the ability to rollback the changes when needed (requirement **R7**).

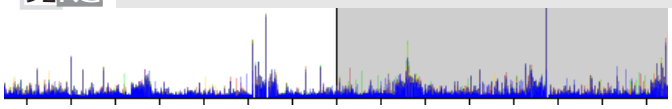


Fig. 8: QuantumDB running on PostgreSQL 9.4: Add a non-nullable column

D. Implementation: QuantumDB

We implemented the approach in an open source tool called *QuantumDB*[4], released under the Apache license. It provides a fluent API for representing changesets, an example of which is shown in Figure 9. *QuantumDB* presently supports PostgreSQL, yet has been designed to be easily extensible to alternative backends. *QuantumDB* presently supports 11 schema operations, such as adding, altering, and dropping columns, adding and dropping foreign keys, adding and dropping indices, and copying, adding, renaming and dropping tables. *QuantumDB* uses database triggers for keeping ghost and original tables in sync. Furthermore, *QuantumDB* provides a query wrapper, intercepting DML queries and transforming them in accordance with the table mappings of the selected schema version.

QuantumDB has been implemented in Java, and comes with over 300 unit and integration tests.

VI. Evaluation

We evaluate our approach by assessing the non-blocking behavior, relevance, applicability, and performance of our *QuantumDB* implementation. To that end we first re-do the experiments described in Section IV to assess the blocking nature of our solution. Subsequently, we apply *QuantumDB* to a collection of changesets obtained from industry.

A. Non-Blocking Schema Evolution

To verify that our approach does *not* block any *DML* queries (requirement **R1**), we re-ran the suite of 19 schema evolution scenarios from Section IV-A on PostgreSQL 9.4. We re-used *Nemesis* and instructed it to use *QuantumDB* to intercept and rewrite the *DML* queries issued by the application running inside *Nemesis*. In addition, instead of issuing *DDL* statements directly on the table under change, *Nemesis* was instructed to apply these schema changes using the *QuantumDB* API.

The measurements confirm that *QuantumDB* is non-blocking for all scenarios. For instance, Figure 8 shows the graph for the first scenario (Figure 1a) with PostgreSQL. Unlike Figure 1a, Figure 8 shows no difference between the *DML* spikes before and after the *DDL* statements execution.

B. Schema History from Industry

To verify that *QuantumDB* can handle real life changesets and databases, we applied it to a series of historical changesets and backup data from industry.

[4] https://github.com/quantumdb/quantumdb

```
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"));
```

Fig. 9: Example of a changeset in the Magnet.me changelog described in the API of *QuantumDB*.

The dataset we used comes from startup Magnet.me[5], a rapidly growing online platform for connecting students with (future) employers, jobs, and internships with presently over 60,000 registered students. From their Git history we obtained all the schema changes performed during a 11-month period (Sept. 2014 – Aug. 2015). By combining them with backups of production data, we could execute schema changes on populated databases.

To manage database schema changes, Magnet.me had already been using Liquibase. Therefore, recreating the change sets using the *QuantumDB* API was easy to do. An example Magnet.me changeset is given in Figure 9, which applies 4 schema operations to 2 different tables.

The Magnet.me changelog for this period consists of 95 changesets, containing 532 schema operations in total. On avereage, each changeset contains 5.6 schema operations, which it applies to 3.05 unique tables. These 95 changesets span a period of 46 weeks, which amounts to roughly two deployable schema changes *per week*.

The earliest historical backup starts with 77 tables, 499 columns, 111 foreign key constraints, 42 sequences, and 28 indices not related to identity columns. This backup contains over 3.8 million records with the biggest table containing over 1.4 million records. In this changelog, 22 tables are created, 13 tables are dropped, 42 new foreign key constraints are introduced, and 37 foreign key constraints are dropped and recreated using different properties.

C. Replaying Schema History

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 proceed 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 (using *QuantumDB*'s "drop" feature described in Section V-C).

[5] https://magnet.me

D. Relevance: Avoiding Blocking DDL

From our analysis of PostgreSQL in Section IV-B, we know which *DDL* operators are blocking. Analyzing the 95 changesets from Magnet.me, we find that 35 out of these rely on blocking DDL operators. These operators modify already in-use tables, and thus would incur down time.

In summary, schema changes are blocking in about one third of the Magnet.me changesets, but *QuantumDB* allows these changesets to be deployed without downtime.

E. Applicability

To verify that *QuantumDB* can be successfully applied to the 95 industrial changesets, we replay the full Magnet.me schema history, and inspect the outcomes of the application.

QuantumDB could be successfully applied immediately to about two third of the changesets (61 out of 95). These changesets fit within the implementation constraints of *QuantumDB*, they made use only of the 11 schema operators currently supported. For many of the remaining 34 cases, the changeset not only modified the structure, but also required *DML* statements to subsquently provide additional modifications to the data (such as creating a new table and populating it). Mixing such *DML* statements into changesets is presently not implemented by *QuantumDB*, but would increase its applicability. We are currently adding support for this. 24 out of these 34 could be *partially* executed, i.e., *QuantumDB* could handle the schema manipulations, but the *DML* queries in between were executed by hand once *QuantumDB* had achieved *Mixed-State*. The remaining 10 were discarded due to current implementation limitations of *QuantumDB*. For example, support for VIEWs or custom FUNCTIONs is not yet provided.

In summary, *QuantumDB* can be applied to two thirds of all changesets; Providing support for mixing *DML* statements in changesets would add 25% to that number.

F. QuantumDB Performance

To assess *QuantumDB*'s performance, we apply the $95 - 10 = 85$ changesets, and measure the duration of the forking operation when applying one changeset.

Note that we do not measure while the database is under load from any database client. The present experiment is primarily intended to verify that *QuantumDB* is able to execute changesets of an industrial level. Analysis of *QuantumDB* under load conditions was provided in Section VI-A.

We ran our 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. This VM was supplied by Magnet.me and is used internally by developers.

It took a combined total of 2 hours, 25 minutes, and 16 seconds to execute the 85 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 means that on average, each changeset could be executed within 1.7 minutes. Note that to avoid negatively impacting the web service performance when using the database, we can limit the rate at which data is being copied from the original tables to the ghost tables.

VII. Discussion

Rolling Upgrades The Magnet.me web service uses Liquibase to manage database schema changes. While it only takes it 7 minutes and 57 seconds to execute all 95 changesets using production data, Liquibase issues blocking *DDL* statements in 35 of the 95 changesets. This means that 35 changesets will block the web service's access to certain tables, thereby becoming unresponsive. Liquibase does not expose the database as multiple schemas, meaning that only *Big-Flip* and *Expand-Contract* methods can be used.

Although Magnet.me practices *Continuous Deployment*, when a database schema change is required, the automated deployment process halts. When this happens, a software engineer has to do a manual deployment, taking all web services offline, apply the schema changes with Liquibase, and then deploy and restarts the new version of the web service.

By adopting *QuantumDB* Magnet.me would no longer need an engineer to manually deploy database schema changes. Instead the new schema could be created during the day automatically because it wouldn't affect the active web service instances. After achieving *Mixed-State*, the new version of the web service could be automatically deployed into production, and the switch over could be done with either the *Big-Flip* or *Rolling Upgrade* method.

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

QuantumDB now supports up to two concurrently active database schemas. This allows us to use either the *Rolling Upgrade* or *Big-Flip* deployment method. If we could increase this limit we could support additional techniques such as *Canary Releases*, where one or more experimental branches could each operate on their own schema version, running side-by-side the "master" version of the web service and database schema. This can be interesting for testing different schemas to see which one performs better.

Scalability We tested *QuantumDB* with databases of 50 millions records in a single table, and upwards from 3.8 million using backups from Magnet.me. Although there are undoubtedly much bigger databases 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*.

We also discovered that the database server load is an important factor in achieving *Mixed-State*. The busier the database system the slower the copying process becomes. In addition *QuantumDB* (like other tools mentioned in Section

VIII), 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 the queries issued by database clients slower. For certain use-cases, applications, and companies this overhead introduced by *QuantumDB* and other tools, might be entirely acceptable if downtime can be avoided with it, but for others this might not be an acceptable trade-off.

More research is required to examine the exact overhead of using *QuantumDB* in write-intensive circumstances.

Data Loss *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 dropping the new database schemas. *QuantumDB* never modifies the data inside the original tables. It does manipulate data in the ghost tables, but only before installing the backward triggers which update the original tables whenever the ghost tables are manipulated.

This approach is safe to use in a production environment. It allows users to deal with failures or abort the schema evolution process.

Implementation Limitations At the time of writing *QuantumDB* is implemented for PostgreSQL and JVM-based languages. In the near future, we plan to support other popular relational database management systems such as Oracle and MySQL. Since those systems may differ in terms of supported features we might need to employ other implementation strategies for some of the *QuantumDB* mechanisms.

Another limitation is that *QuantumDB* does not support *DML* statements as part of the schema evolution process. This means that we cannot version both the structure and the data of a database with the current version of *QuantumDB*. Since, as seen in Section VI, such statements are used in practice. Work is already underway to support them.

Foreign Keys Chains The duration of the "forking" process to achieve *Mixed-State* can vary greatly. It depends on the number of ghost tables that must be created and filled, which in turn depends on which tables are under change. The more a table under change is referenced to by foreign keys (directly or transitively), the more ghost tables need to be constructed. In the Magnet.me database we observed that the "users" table is such a pivotal table that changing it requires the creation of ghost tables for the majority of tables in the database.

Conversely, we observed that changing a link table — having a primary key composed of multiple columns, referring to other tables —, or changing a relatively new table, requires the creation of less ghost tables.

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 the same features, and if no alternative strategy can be found to achieve *Mixed-State* using this approach, then *QuantumDB* might not be the general solution for schema evolution we believe it to be.

We tested *QuantumDB* on the database backups and changesets from Magnet.me. Although they span the entire lifetime of the product's latest reincarnation — making them representative for Magnet.me — they may not be representative of the database size and schema changes for the rest of the industry.

Replication The source code of both *Nemesis* and *QuantumDB* are available for download at: http://github.com/quantumdb. A full replication package, including the data as well as the software infrastructure required to conduct the *Nemesis* measurements is available from our web site[6]. The results of the *Nemesis* experiments are due to their size not available for download, but are available upon request. The Magnet.me schema modifications and database backups are confidential but may be made available upon request for research purposes.

VIII. Related Work

Database schema evolution has been widely investigated by the scientific community [14]. For co-evolution of schemas and client applications, tool-supported approaches rely on transformational and generative techniques. Several authors attempt to contain the ripple effect of changes to the database schema, e.g., by generating *wrappers* that provide backward compatibility [17]. Bidirectional transformations [16] can also be used to decouple the evolution of the database schema from the evolution of the queries.

PRISM [6] provides integrated support to *relational* schema evolution. This tool suite includes (1) a language for the specification of Schema Modification Operators (*SMOs*) for relational schemas, (2) impact analysis tools that evaluate the effects of such operators, (3) automatic data migration support, and (4) translation of old queries to work on the new schema. Query adaptation derives from the *SMOs* and combines SQL view generation and query rewriting techniques.

The above-mentioned approaches support joint evolution of schemas, data and queries *offline*. Thus, they require a system shutdown before it can be automatically adapted, recompiled and redeployed. In contrast, *QuantumDB* supports this process at *runtime*, avoiding downtime of client applications.

There have been various attempts in industry to create tools which could deal with database schema evolution at *runtime*, thereby limiting downtime. Notable mentions are OpenArk Kit[7], Percona[8], TableMigrator[9], and Large Hadron Migrator[10]. These all follow a similar strategy where a structural copy of the table under change is created. The schema operation is then applied to this ghost table. Data is copied from the original to the ghost table and kept synchronized using database triggers. When the copy phase has completed, the original table and the ghost table atomically switch names, and the original table is dropped.

[6] https://github.com/quantumdb/nemesis
[7] http://code.openark.org/forge/openark-kit
[8] http://www.percona.com/software/percona-toolkit
[9] https://github.com/freels/table_migrator
[10] https://github.com/soundcloud/lhm

Facebook's OSC[11] takes a similar approach, but uses the triggers to store the data changes in a separate table, to replay them asynchronously on the ghost table. Github's ghost[12] takes an alternate approach: It reads the binlog of the database, and replicates data changes to the ghost table asynchronously. These asynchronous approaches trade transactional consistency (changes to both tables being done in the same transaction) for performance, and the ability to pause data migration. These tools all require a hard cut-over, where either the old table, or the new table can be used, as opposed to *QuantumDB* which has a soft cut-over phase where both tables can be used concurrently.

Researchers have suggested to modify the source code of existing relational databases [11], using column-oriented databases [10], or creating entirely new databases specifically tailored for more modern requirements like Google's F1 [15] and Spanner [5].

All the above tools and approaches are limited to making changes to *one* table at a time. By contrast, *QuantumDB* supports modifying multiple tables in a single set of schema changes. Furthermore, unlike the tools surveyed above, *QuantumDB* embraces *Mixed-State*. This allows developers to freely choose the continuous deployment method, while existing approaches require *Big-Flip*. Last but not least, *QuantumDB* offers full support for referential integrity preservation. All the techniques and tools we have surveyed either have no support or unsafe support for foreign key constraints.

IX. Conclusion

The goal of this research is to identify and evaluate an approach which allows us to evolve the schema of a SQL database in a *Continuous Deployment* setting. To that end we identified a set of requirements (**R1-R7**) 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 popular SQL databases, through an infrastucture (*Nemesis*) which measures this blocking behavior. Based on the requirements, and existing tools and approaches, we developed a novel 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 *QuantumDB* can deal with the complexity of changesets produced in practice, and we suggested future directions.

The key contributions of this paper are:

- *QuantumDB*: A tool-supported approach to deploy multiple concurrently active schemas and migrate data between those schemas in a way that is transparent to the developer and incurs zero-downtime.
- Empirical evidence that *QuantumDB* effectively supports changesets as used in industry for medium-sized databases (hundreds of columns, millions of records).

[11] https://www.facebook.com/note.php?note_id=430801045932

[12] http://githubengineering.com/gh-ost-github-s-online-migration-tool-for-mysql/

QuantumDB advances the state of the art through its support for *Mixed-State* and foreign key constraints preservation. The *QuantumDB* tool is available from GitHub. Development teams can use it to support the continuous deployment of features, and to experiment with multiple features in an A/B testing setting, even if those features require database schema changes affecting millions of records. Furthermore, researchers can use the open source *QuantumDB* and *Nemesis* infrastructure to experiment with refined solutions for zero-downtime schema evolution.

Acknowledgments This work was partially supported by the EU Project STAMP ICT-16-10 No.731529.

References

[1] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[2] G. G. Claps, R. B. Svensson, and A. Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21 – 31, 2015.

[3] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97:113–121, 2015.

[4] A. Cleve, N. Noughi, and J.-L. Hainaut. Dynamic program analysis for database reverse engineering. In *GTTSE*, volume 7680 of *LNCS*, pages 297–321. Springer, 2013.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[6] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. *Proc. VLDB Endowment*, 4(2):117–128, Nov. 2010.

[7] M. de Jong and A. van Deursen. Continuous deployment and schema evolution in SQL databases. In *Proc. of 3rd International Workshop on Release Engineering (RELENG)*, pages 16–19. IEEE, 2015.

[8] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it? Toward dependable, online upgrades in enterprise system. In *Proc. of Middleware '09*, pages 18:1–18:20. Springer-Verlag, 2009.

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

[10] Z. Liu, B. He, H.-I. Hsiao, and Y. Chen. Efficient and scalable data evolution with column-oriented databases. In *Proc. of EDBT/ICDT '11*, pages 105–116, New York, NY, USA, 2011. ACM.

[11] I. Neamtiu, J. Bardin, M. R. Uddin, D.-Y. Lin, and P. Bhattacharya. Improving cloud availability with on-the-fly schema updates. In *Proc. of COMAD '13*. ACM, 2013.

[12] H. H. Olsson, H. Alahyari, and J. Bosch. Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Proc. of SEAA '12*, pages 392–399. IEEE, 2012.

[13] D. Qiu, B. Li, and Z. Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proc. of ESEC/FSE 2013*, pages 125–135, New York, NY, USA, 2013. ACM.

[14] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, Dec. 2006.

[15] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. *Proc. VLDB Endowment*, 6(11):1068–1079, Aug. 2013.

[16] J. Terwilliger, A. Cleve, and C. Curino. How clean is your sandbox? : Towards a unified theoretical framework for incremental bidirectional transformations. In *Proc. of ICMT 2012*, volume 7307 of *LNCS*, pages 1–23. Springer, 2012.

[17] P. Thiran, J.-L. Hainaut, G.-J. Houben, and D. Benslimane. Wrapper-based evolution of legacy information systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359, 2006.

TUD-SERG-2017-005
ISSN 1872-5392