A Benchmark for Online Non-Blocking Schema Transformations

Lesley Wevers, Matthijs Hofstra, Menno Tammens, Marieke Huisman and Maurice van Keulen
University of Twente, Enschede, the Netherlands
l.wevers@utwente.nl, {m.hofstra, m.j.tammens}@student.utwente.nl, {m.huisman, m.vankeulen}@utwente.nl

Keywords: Online Schema Transformations, Database Transactions, Benchmark.

Abstract: This paper presents a benchmark for measuring the blocking behavior of schema transformations in relational database systems. As a basis for our benchmark, we have developed criteria for the functionality and performance of schema transformation mechanisms based on the characteristics of state of the art approaches. To address limitations of existing approaches, we assert that schema transformations must be composable while satisfying the ACID guarantees like regular database transactions. Additionally, we have identified important classes of basic and complex relational schema transformations that a schema transformation mechanism should be able to perform. Based on these transformations and our criteria, we have developed a benchmark that extends the standard TPC-C benchmark with schema transformations, which can be used to analyze the blocking behavior of schema transformations in database systems. The goal of the benchmark is not only to evaluate existing solutions for non-blocking schema transformations, but also to challenge the database community to find solutions that allow more complex transactional schema transformations.

1 INTRODUCTION

For applications storing data in a database, changes in requirements often lead to changes in the database schema. This often requires changing the physical layout of the data to support new features, or to improve performance. However, current DBMSs are ill-equipped for changing the structure of the data while the database is in use, causing downtime for database applications. For instance, an investigation on schema changes in MediaWiki, the software behind Wikipedia, shows that 170 schema changes are performed in less than four years time (Curino et al., 2008). Many of these changes required a global lock on database tables, which made it impossible to edit articles during the schema change. The largest schema change took nearly 24 hours to complete.

Downtime due to schema changes is a real problem in systems that need 24/7 availability (Neamtiu and Dumitras, 2011). Unavailability can lead to missed revenue in case of payment systems not working, economic damage in case of service level agreements not being met, to possibly life threatening situations if medical records cannot be retrieved. Not only is this a problem in its own right, evolution of software can also be slowed down as developers tend to avoid making changes because of the downtime consequences.

Goals We have specified criteria for non-blocking schema transformations, and we have developed a benchmark to evaluate DBMSs with regard to these criteria. Our benchmark extends the standard TPC-C benchmark with basic and complex schema transformations, where we measure the impact of these transformations on the TPC-C workload. This benchmark can be used to evaluate the support for online schema changes in existing DBMSs, and serves as a challenge to the database community to find solutions to perform non-blocking schema changes.

Problem Statement The SQL standard provides the data definition language (DDL) to perform basic schema changes such as adding and removing of columns and relations. However, in current DBMSs, not all of these operations can be executed in parallel with other transactions. The effect is that concurrent transactions completely halt until the (possibly long) execution of the schema change has finished.

Moreover, not all realistic schema changes can be expressed using a single DDL operation. Important examples of such transformations include splitting and merging of tables (Ronström, 2000; Løland and Hvasshovd, 2006), changing the cardinality of a relationship, and changing the primary key of a table. Many of these schema changes involve bulk transformation of data using *UPDATE* statements, which is

blocking in most DBMSs to avoid concurrency conflicts. To complicate matters further, schema transformations can affect existing indices, as well as (foreign key) constraints.

To illustrate the problem, PostgreSQL can perform many DDL operations instantaneously, and allows DDL operations to be safely composed into more complex schema changes using transactions. This approach allows composed schema changes to be performed without significant blocking as long as they consist only of operations that can be performed instantaneously. However, bulk UPDATE operations and some DDL operations cannot be performed instantaneously, and all DDL and UPDATE operations take a global table lock for the duration of the transformation. The effect is that many complex transformations cannot be performed without blocking.

Another example is MySQL, which recently added limited support for online DDL operations [1]. However, while MySQL supports transactional schema transformations, it auto-commits after every operation, which requires the application program to handle all intermediate states of the transformation. Moreover, UPDATE operations cannot be performed online, which makes it impossible to perform many complex transformations online.

Tools such as pt-online-schema-change[2], oak-online-alter-table[3], and online-schema-change[4] have been developed in industry to allow online schema transformations on existing DBMSs using a method developed by Ronström (Ronström, 2000). While these tools are limited in capability, they show that there is a strong need for solutions, and that there is room for improvement in current DBMSs.

Overview In Section 2 we provide an overview of the state of the art in online schema changes, and show that current techniques are not sufficient for complex schema changes. To address the limitations of existing approach, we have specified general criteria for non-blocking schema changes, as discussed in Section 3. Additionally, as discussed in Section 4, we have identified important classes of relational schema transformations that we believe should be supported by schema transformation mechanisms. In Section 5 we discuss our benchmark which evaluates how DBMSs cope with the schema changes that we have identified against our criteria. In Section 6 we discuss our implementation of the benchmark, and show example results for PostgreSQL and MySQL.

[1] dev.mysql.com/doc/refman/5.6/en/innodb-online-ddl.html

[2] www.percona.com/doc/percona-toolkit

[3] code.openark.org/forge/openark-kit

[4] www.facebook.com/notes/430801045932

Contributions The contributions of this paper are:

- A challenge for the database community to find solutions for non-blocking transactional schema transformations.
- Criteria for evaluating online schema transformation mechanisms in general, and for the relational data model in particular.
- A concrete benchmark based on TPC-C to evaluate the blocking behavior of online schema transformations in existing database systems.

2 STATE OF THE ART

This section provides a brief overview of the state of the art in non-blocking schema transformations. While we show that existing solutions are insufficient for complex transformations, we use their characteristics as a basis for our criteria for online schema transformations, which are discussed in the next section.

Synchronization Based Methods Ronström proposed a method that allows changing of columns, adding indices, and horizontally and vertical splitting and merging of tables by using database triggers (Ronström, 2000). A benefit of Ronström's method is that it can be implemented on top of existing DBMSs that provide support for triggers. A number of tools have already been developed in industry, such as pt-online-schema-change. Løland and Hvasshovd provide *log redo* as an alternative implementation approach to Ronströms method, which reduces impact on concurrent transactions, but requires built-in support from the DBMSs (Løland and Hvasshovd, 2006).

While Ronströms method can perform many schema transformations without blocking, it can take a long time until a transformation commits. Moreover, Ronströms methods only provides ACID guarantees for individual transformations. For complex transformations that consist of multiple schema transformations, Ronström proposes the use of SAGAs. The idea of SAGAs is to execute the individual operations of a transaction as a sequence of transactions, where for each operation an undo operation is provided that can be used to rollback the complete sequence (Garcia-Molina and Salem, 1987). While SAGAs provide failure atomicity for composed transactions, they expose intermediate states of the transformation to database programs. This requires database programs to be adapted to handle these intermediate states, which requires additional development effort, and which is potentially prone to mistakes.

On-the-fly Schema Changes Neamtiu et al. propose on-the-fly schema changes in databases (Neamtiu et al., 2013). In their approach, a schema change command can specify multiple operations on multiple tables. Upon access of a table, a safety check is performed to see if a schema change is in progress. If so, data is transformed before it can be accessed. The authors have implemented a prototype in SQLite, and have performed experiments, showing very short time to commit and low overhead. However, their implementation does not allow a new schema transformation to start before any running schema transformation has completed. Their approach allows for composition of schema transformations to a limited degree, as an update operation can consist of multiple operations on multiple tables. However, they only support relatively simple operations, such as adding and removing relations and columns. Complex operations such as splitting and joining relations are mentioned as future work.

Online Schema Change in F1 Rae et al. investigate online schema change in the Google F1 database (Rae et al., 2013), which is a relational DBMS built on top of a distributed key-value store. The distributed setting introduces additional complications, as inconsistencies between schemas in different compute nodes can lead to incorrect behavior. Their solution is to split a complex schema change into a sequence of simpler schema changes that are guaranteed not to perform conflicting operations on the distributed state. Each schema change operation is fully executed before the next one starts, so that the composed operation is guaranteed to be correct. However, while this approach allows for some complex transformations, many types of transformations are not covered. Moreover, similar to the use of SAGAs in Ronströms method, intermediate states of transformations are visible to database applications, which may have to be adapted to handle these states.

Rewriting Approaches Curino et al. investigate automatic rewriting of queries, updates and integrity constraints based on schema transformation specifications (Curino et al., 2010). This provides an alternative approach to schema transformations by rewriting operations on the new schema to operations on the old schema. A similar approach is to use updatable views, where the new schema is represented as an updatable view over the old schema. However, these approaches do not provide a complete solution to online schema transformations, as the schema may need to be extended to store new types of data, or data may need to be physically rearranged to improve performance.

Related Topics Apart from physical transformation of data without blocking, another challenge in schema evolution is handling application migration. For instance, there is work on schema versioning, which investigates how multiple schema versions can be used concurrently through versioned interfaces (Roddick, 1995). Work in this field can benefit from the ability to perform non-blocking schema transformations.

Also of note is the problem of keeping materialized views up to date, as many of the same issues of performing non-blocking physical schema transformations are encountered (Gupta and Mumick, 1995).

Discussion Each of the discussed approaches to physical transformation of data has their own strengths and limitations. In particular, existing methods have different characteristics in terms of time to commit and the impact on concurrent transactions. For instance, on-the-fly transformations commit before the physical transformation of data starts, while Ronströms method commits after the physical transformation has completed. However, not all transformations can be performed on the fly, such as creating of indices or checking of constraints. Another important observation is that none of the existing approaches provides a satisfactory solution to composition of schema transformations. Current solutions generally expose database programs to intermediate states of the transformation, which breaks the isolation property of the ACID guarantees.

3 CRITERIA FOR ONLINE TRANSFORMATIONS

In this section we define criteria for non-blocking schema transformations, which we use as the basis for our benchmarks. Our criteria are based on the characteristics of the state of the art approaches to non-blocking schema transformations as discussed in the previous section. We address the limitations of existing approaches, while also considering their strengths. We specify criteria both for the functionality of schema transformations and for their performance. For every criterion, we discuss the *ideal* behavior of the schema transformation mechanism, and we also discuss what behavior we would still consider *acceptable* for systems that must be online 24/7. Note that our criteria make no assumptions about the specific strategies used to perform schema transformations, and also apply to data models other than the relational model.

Functional Criteria First, we specify criteria for the functionality provided by a schema transformation mechanism:

1. **Expressivity**. Ideally, an online schema transformation mechanism can perform any conceivable schema transformation online. For example, in the relational model this would be all schema transformations that can be expressed in SQL. However, in practice, it could be sufficient if we can only perform a subset of all possible transformations online. For instance, we discuss an important set of relational transformations in Section 4, which could be sufficient in practice.
2. **Transformation of Data**. After a schema change commits, all existing data must be available in the new schema. This is the main challenge in performing schema transformations online, and any lesser guarantees are generally unacceptable.
3. **Transactional Guarantees**. For the correctness of database programs, and to ensure database integrity, it is important that schema transformations satisfy the ACID properties, as is currently the norm for OLTP transactions:
   (a) **Atomicity**. Systems must allow transactional composition of basic schema transformations into more complex transformations, while maintaining transactional guarantees. If this is not possible, intermediate schemas of the basic schema transformations are visible to other transactions. For some applications this may be acceptable, but handling them comes at the cost of additional development effort.
   (b) **Consistency**. Ideally, all defined constraints hold for the new schema. However, deferring the checking of constraint to a later time may be acceptable for certain applications.
   (c) **Isolation**. The execution of transformations must have serializable semantics, i.e., all concurrent transactions must either see data in the old schema, or see data in the new schema. Partially transformed states where one part of the data is available in the old schema, and the other part is available in the new schema must not be visible to concurrent transactions.
   (d) **Durability** Once a schema transformation has been committed, its effect must be persistent, even in the event of a system failure, i.e. all transactions that are serialized after the transformation must see the result of the transformation. After a system failure, it may never be the case that data is lost, that a database is left in a partially transformed state, or that a database is left in an intermediate state of a transformation.
4. **Application Migration** There must be a mechanism to ensure that database programs can continue to operate correctly during and after the schema transformation. Ideally, the system provides schema versioning or automatic rewriting of queries, updates and integrity constraints. However, it may be sufficient if stored procedures can be updated as part of a schema transformation.
5. **Declarativity**. Ideally, schema transformations are specified declaratively. The user should not have to be concerned with the execution details of a schema transformation. For instance, the SQL data definition language can be considered to be declarative. Manual implementation of schema transformations, e.g., using Ronströms method, can be expensive to develop and can be prone to mistakes that can damage data integrity, but could be acceptable in certain applications.

Performance Criteria Second, we specify criteria for the performance of online schema change mechanisms, where we assume that OLTP transactions and schema transformation transactions are executing concurrently. We first discuss the impact of schema transformations on OLTP transactions, and then we describe performance criteria for the schema transformations themselves.

1. **Impact of a schema transformation on concurrent transactions** Executing a schema change concurrently with other transactions will have an impact on those transactions. We distinguish several kinds of impact:
   (a) **Blocking**. Transactions should always be able to make progress independent of the progress of concurrent schema transformations. Ideally, a schema transformation never blocks the execution of concurrent transactions. However, depending on the application, blocking for short periods of time, e.g., up to a few seconds, could be acceptable. Additionally, schema transformation mechanisms must never prevent new transactions from starting while a schema transformation is in progress.
   (b) **Slowdown**. A general slowdown in throughput and latency of transactions is acceptable to a certain degree, depending on the application. Ideally, more complex transformations can be performed without causing additional slowdown. However, slowdown that depends on the complexity of the transformation could be acceptable, but may impose limitations on the complexity of online transformations.

(c) **Aborts**. Ideally, schema transformations allow transactions that are already running to continue executing without aborting them. For instance, by means of snapshot isolation, and by translating updates on the old schema to the new schema. This is especially important for long-running transactions. However, it is acceptable to abort short-running transactions, as long as they do not suffer from starvation during the execution of the transformation.

2. **Performance criteria for online schema transformations** For online schema transformations, we have identified different requirements:

(a) **Time to Commit**. The time to commit is defined by how long it takes for the results of a schema transformation to be visible to other transactions. A very long time to commit can be unacceptable in time critical situations. To cater for on-the-fly transformation of data, i.e., transformation of data *after* the transformation has been committed (Neamtiu et al., 2013), we want to distinguish time to commit from the *transformation time*, which is the total amount of time needed to transform all data in the database. As long as concurrent transactions are not impacted, the transformation time does not matter.

(b) **Aborts and Recovery**. Due to their long running time, it is generally not acceptable to abort schema transformations due to concurrency issues. As an exception, aborting a transformation before it is executed is acceptable, for instance, in case a conflicting schema transformations is already in progress. Moreover, aborts due to semantic reasons, such as constraint violations, cannot be avoided. In case of a system failure during a transformation, ideally the system can recover and continue execution of the transformation. However, due to the rarity of system failures, aborting the transaction could be acceptable. It is important that recovery from an abort also minimizes impact on concurrent transactions. Finally, a request for schema transformation should only be rejected if there is a conflicting uncommitted schema transformation; processing of any schema transformation that has already been committed should not delay the start of another schema transformation.

(c) **Memory Usage**. Apart from timing behavior, a schema transformation should not consume large amounts of memory. Ideally, a transformation should be performed in-place, and not construct copies of the data. However, depending on available hardware resources, it could be acceptable if additional memory is needed to perform the transformation.

Discussion Sockut and Iyer also discuss requirements for strategies that perform online reorganizations (Sockut and Iyer, 2009). They consider not only logical and physical reorganizations, but they also have a strong focus on maintenance reorganizations, i.e., changing the physical arrangement of data without changing the schema. Their main requirements are correctness of reorganizations and user activities; tolerable degradation of user activity performance during reorganizations; eventual completion of reorganizations; and, in case of errors, data must be recoverable and transformations must be restartable.

Our main addition to these requirements is that basic transformations should be composable into complex transformations using transactions, while maintaining ACID guarantees and satisfying the performance requirements. A difference in our requirements is that instead of requiring eventual completion of transformations, we only consider the time to commit, and leave the matter of progress as an implementation detail to the DBMS, which may choose to make progress if this reduces impact on running transactions.

4 RELATIONAL TRANSFORMATIONS

In this section we identify important classes of relational schema transformations that could be required in practice. In our benchmark, we select representative transformations for these classes on the TPC-C schema. We do not make any a priori assumptions about the difficulty of schema transformations or the capabilities of existing systems. To identify important schema transformations, we consider databases that are implementations of Entity-Relationship (ER) models.

ER Models and Implementations ER modelling is a standard method for high-level modelling of databases (Chen, 1976). In the ER model, a domain is modelled using entities and relationships between these entities, where entities have named attributes, and relationships have a cardinality of either 1-to-1, 1-to-n or n-to-m. An ER model can be translated to a relational database schema in a straightforward manner: Entities are represented as relations, attributes are

mapped to columns and relationships are encoded using foreign keys. Several implementation decisions are made in this translation, for example, which types to use for the attributes and which indices to create. Based on this, we can identify two kinds of schema transformations: logical transformations that correspond to *changes in the ER model*, and physical transformations that correspond to *changes in the implementation decisions*.

ER Model Transformations We consider logical transformations on relational databases that correspond to the following ER-model transformations:

- Creating, renaming and deleting entities and attributes.
- Changing constraints on attributes, such as uniqueness, nullability and check constraints.
- Creating and deleting relationships, and changing the cardinality of relationships.
- Merging two entities through a relationship into a single entity, and the reverse, splitting a single entity into two entities with a relationship between them.
- Moving attributes from an entity to another entity through a relationship.

Note that certain changes in the ER model do not result in an actual change of the database schema, but only in a data transformation. For instance, changing the currency of a price attribute is an example of an ER-model transformation that needs no database schema change, but only a data transformation. Such schema transformations correspond to normal bulk data updates.

Implementation Transformations Furthermore, we also consider physical transformations on relational databases that correspond to changes in the implementation decisions:

- Changing the names and types of columns that represent an attribute.
- Changing a (composite) primary key over attributes to a surrogate key, such as a sequential number, and vice versa.
- Adding and removing indices.
- Changing the implementation of relationships to either store tuples of related primary keys in a separate table, store relationships in an entity table (for 1-to-*n* and 1-to-1 relationships), or merging of entity tables that have a 1-to-1 relationship.
- Changing between computing aggregations on-the-fly, or storing precomputed values of aggregations.

The above set of transformations is by no means complete, but it provides an important subset of schema changes that are used in practice, and it is sufficient for the benchmark to showcase the limitations of schema transformation mechanisms in existing database systems.

5 BENCHMARK SPECIFICATION

In this section, we describe a benchmark to investigate the behavior of online schema transformations in database systems. We shortly discuss the TPC-C benchmark, we specify how our benchmark should be performed, we discuss the interpretation of the results, and we define concrete schema transformations for the TPC-C schema. Finally, we discuss the completeness of our benchmark.

The TPC-C Benchmark The TPC-C benchmark (TPC, 2010) is an industry standard OLTP benchmark that simulates an order-entry environment. Figure 1 shows a high-level overview of the TPC-C benchmark schema. TPC-C specifies the generation of databases of arbitrary sizes by varying the number of warehouses W. The workload consists of a number of concurrent terminals executing transactions of five types: New Order, Payment, Order Status, Delivery and Stock Level. The transaction type is selected at random, following a distribution as specified by TPC-C. The TPC-C benchmark measures the number of New Order transactions per minute. Additionally, TPC-C also specifies response time constraints for all transaction types.

Our benchmark extends the TPC-C benchmark with schema transformations, but does not require modifications, i.e., we use a standard TPC-C database and workload. This means that existing TPC-C benchmark implementations can be used. However,

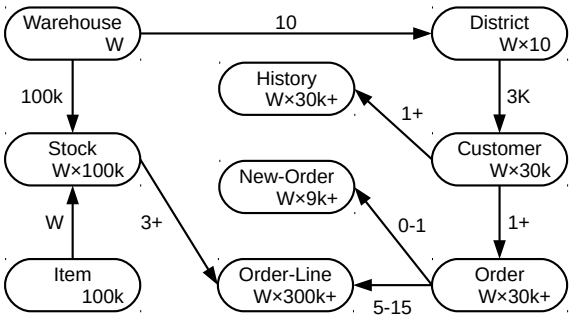

Figure 1: TPC-C schema overview (TPC, 2010).

we assume that the TPC-C transactions are implemented as stored procedures that can be changed as part of a schema transformation. The TPC-C scaling factor should be chosen such that the impact of a schema transformation are measurable.

Benchmarking Process The execution of a benchmark case is done in multiple phases:

1. **Setup** Create a TPC-C database.
2. **Preparation** If specified by the benchmark case, perform the preparation transformation.
3. **Intro** Start the TPC-C benchmark load, while logging the executed transactions. Wait for 10 minutes before starting the transformation, while measuring the baseline TPC-C performance.
4. **Transformation** Start the execution of the actual transformation, and wait for it to complete. Log the transformation begin and end time.
5. **Outro** Wait for another 10 minutes while measuring the TPC-C performance in the outro phase. Note that on-the-fly schema changes may require more than 10 minutes to complete. In this case, additional time can be added to the outro phase.

After each transaction attempt, the type of transaction, its starting time and its end time are logged. Failed transactions are logged with type `ERROR`.

Result Analysis As shown in Figure 2, we present the result of a benchmark as a line graph that plots the TPC-C transaction execution rate in time intervals of fixed length. In the graph, we mark the start and commit time of the transformation with vertical lines. Additionally, we plot `ERROR` transactions in red to detect aborted and failed transactions.

We use these graphs to evaluate the impact of schema transformations. First, we can see the time to commit for the transformation. Second, we can see the effect of the schema transformation on the transaction throughput both during the transformation phase, and during the outro phase. The latter can be used to investigate the performance of on-the-fly schema transformations. Note that we are not interested in the absolute transaction throughput, but only in the relative throughput during the transformation compared to the baseline performance in the intro phase. If a schema change is blocking, we see a throughput of zero. We can determine the total blocking time by computing the cumulative time of the intervals where the throughput is below a certain threshold. Additionally, we can compute the average throughput during time periods to determine the slowdown caused by the schema transformation.

Transformations Table 1 and Table 2 show the transformations that we have selected for our benchmark. These were chosen as representatives for the transformations identified in Section 4. Like the TPC-C specification, we do not prescribe a specific implementation for the transformations, as we do not want to preclude the use of features provided by the DBMS. We define most schema transformations on the ORDER-LINE table, which is the largest table in a populated TPC-C database. Transformations affecting multiple tables are performed on the ORDER and ORDER-LINE tables. Note that we prefix all column names with the abbreviated table name. For example, OL_NAME is a column in ORDER-LINE. and O_NAME is a column in ORDER. Some transformations have a preparation part (presented in *italics*) where the TPC-C schema is transformed prior to the actual transformation that we want to evaluate. For some benchmarks, we use the resulting schema of another benchmark as a starting point. For instance, there is a benchmark where a table is split, which we also use as the preparation phase for a transformation where we rejoin the tables to obtain the original TPC-C schema. Also note that many transformations change the stored procedures, so that the TPC-C benchmark can continue running on the transformed schema. We have divided the benchmark into basic and complex transformations:

- **Basic Transformations** Table 1 shows basic benchmark cases that correspond to the operations provided by the SQL standard, categorized by different types of transformations. These cases are relatively synthetic in nature, and can mainly be used to compare the capabilities of different schema transformation strategies. For most benchmark cases, there are two versions, one that does not affect running transactions, and one that does. The latter have names ending in `-sp`, and require changing the stored procedures. Note that while creating indices is not technically a schema transformation, it is an important DDL operation.
- **Complex Transformations** Table 2 shows complex benchmark cases, which generally require multiple DDL operations, and involve *UPDATE* statements. If a DBMS can perform these complex cases, it is likely that the basic cases are also covered, as most complex cases overlap with the basic cases. The following list provides a high-level overview of the transformations:
  1. **add-column-derived** Instead of storing only the total amount of the order, we also want to separately store the sales tax paid on the order. We assume that all previous and new orders have a 21% sales tax.

Table 1: Basic benchmark cases.

Relation Transformations	
create-relation	Create a new relation TEST.
rename-relation	Rename ORDER-LINE to ORDER-LINE-B. Change the stored procedures to use ORDER-LINE-B instead of ORDER-LINE.
remove-relation	*Copy ORDER-LINE to ORDER-LINE-B.* Drop ORDER-LINE-B.
remove-relation-sp	*Copy ORDER-LINE to ORDER-LINE-B.* Drop ORDER-LINE. Change the stored procedures to use ORDER-LINE-B instead of ORDER-LINE.
Column Transformations	
add-column	Create OL_TAX as NULLABLE of the same type as OL_AMOUNT.
add-column-sp	Create OL_TAX as NULLABLE of the same type as OL_AMOUNT. Change the stored procedures to set OL_TAX to OL_AMOUNT $\times$ 0.21 upon insertion.
add-column-default	Create OL_TAX as NOT NULL with default value 0 of the same type as OL_AMOUNT.
add-column-default-sp	Create OL_TAX as NOT NULL with default value 0 of the same type as OL_AMOUNT. Change the stored procedures to set OL_TAX to OL_AMOUNT $\times$ 0.21 upon insertion.
rename-column	*Copy column OL_AMOUNT to OL_AMOUNT_B.* Rename column OL_AMOUNT_B to OL_AMOUNT_C.
rename-column-sp	Rename column OL_AMOUNT to OL_AMOUNT_B. Change the stored procedures to use OL_AMOUNT_B instead of OL_AMOUNT.
remove-column	*Copy OL_AMOUNT to OL_AMOUNT_B.* Drop OL_AMOUNT_B.
remove-column-sp	*Copy OL_AMOUNT to OL_AMOUNT_B.* Drop OL_AMOUNT. Change the stored procedures to use OL_AMOUNT_B instead of OL_AMOUNT.
change-type-a	Change OL_NUMBER to use a greater range of integers.
change-type-b	Split OL_DIST_INFO into two columns OL_DIST_INFO_A and OL_DIST_INFO_B. Change the stored procedures to split the value for OL_DIST_INFO into two parts upon insertion, and to concatenate the values upon retrieval.
Index Transformations	
create-index	Create an index on OL_I_ID.
remove-index	*Execute create-index-a.* Drop the index created by create-index.
Constraint Transformations	
create-constraint	Create a constraint to validate that $1 \leq$ OL_NUMBER $\leq$ O_OL_CNT.
remove-constraint	*Execute create-constraint-a.* Drop the constraint created by create-constraint.
create-unique	*Create a column OL_U, and fill this with unique values.* Add a uniqueness constraint on OL_U.
remove-unique	*Execute create-unique-a.* Drop the uniqueness constraints created by create-unique.
Data Transformations	
change-data	Set OL_AMOUNT to OL_AMOUNT $\times$ 2.

Table 2: Complex benchmark cases.

Complex Transformations	
add-column-derived	Create OL_TAX as NOT NULL and initial value OL_AMOUNT $\times$ 0.21. Change the stored procedures to set OL_TAX to OL_AMOUNT $\times$ 0.21 upon insertion.
change-primary	Add a column O_GUID with unique values. Add a column OL_O_GUID, and set its value to the O_GUID of the order corresponding to this order line. Set (OL_O_GUID, OL_O_NUMBER) as the primary key. Drop OL_O_ID, OL_D_ID and OL_W_ID. Add a column NO_O_GUID, and set its value to the O_GUID of the corresponding order. Drop NO_O_ID, NO_D_ID and NO_W_ID. Set NO_O_GUID as the primary key. Drop O_ID. Update the stored procedures to use the new structure, change STOCK_LEVEL to select the top 20 rows ordered by O_GUID instead of the condition OL_O_ID $\geq$ (ST_O_ID - 20).
split-relation	Create ORDER-ORDER-LINE with columns OOL_O_ID, OOL_D_ID, OOL_W_ID, OOL_OL_ID and OOL_NUMBER. Create a column OL_ID with unique values as primary key. Insert all tuples (OL_O_ID, OL_D_ID, OL_W_ID, OL_ID, OL_NUMBER) into ORDER_ORDER_LINE. Drop columns OL_O_ID, OL_D_ID, OL_W_ID, OL_ID and OL_NUMBER. Update the stored procedures to use the new structure.
join-relation	*Execute split-relation.* Add columns OL_O_ID, OL_D_ID, OL_W_ID and OL_NUMBER and set their values to the corresponding values in ORDER-ORDER-LINE. Drop OL_ID, and set primary key (OL_O_ID, OL_D_ID, OL_W_ID, OL_NUMBER). Drop relation ORDER-ORDER-LINE. Update the stored procedures to use the original stored procedures.
defactorize	Add column OL_CARRIER_ID, and set its value to O_CARRIER_ID of the corresponding order. Drop column O_CARRIER_ID. Update the stored procedures to use the new structure.
factorize	*Execute defactorize.* Add column O_CARRIER_ID, and set its value to OL_CARRIER_ID for the corresponding order line where OL_NUMBER = 1. Drop column OL_CARRIER_ID. Update the stored procedures to use the original stored procedures.
factorize-boolean	Add boolean column O_IS_NEW and set its value to true if NEW-ORDER contains the corresponding order, otherwise set it to false. Drop relation NEW-ORDER. Update the stored procedures to use the new structure.
defactorize-boolean	*Execute factorize-boolean.* Create table NEW-ORDER as original. Insert the primary key of all orders into NEW-ORDER where O_IS_NEW = true. Drop column O_IS_NEW. Update the stored procedures to use the original stored procedures.
precompute-aggregate	Add column O_TOTAL_AMOUNT and set its value to the sum of OL_AMOUNT of the corresponding order lines. Update the stored procedures to update O_TOTAL_AMOUNT when inserting order lines, and to use O_TOTAL_AMOUNT instead of computing the aggregate.

2. **change-primary** In the original TPC-C benchmark, each warehouse maintains a separate order counter, so that orders from different warehouses share the same key space. In this benchmark, we change orders to have a globally unique order identifier. We also change all order lines to refer to this new identifier.
3. **split-relation** We want to store the relation between ORDER and ORDER-LINE in a separate table ORDER-ORDER-LINE, and use a surrogate key for ORDER-LINES.
4. **join-relation** We perform split-relation in reverse, i.e., we take the resulting schema of split-relation, and transform this back to the original TPC-C schema.
5. **defactorize** Instead of allowing only a single carrier for an order, we want the ability to have multiple carriers. To do this, we store the carrier per order line instead of per order.
6. **factorize** We perform the inverse operation of defactorize. Instead of a carrier per order line, we want only a single carrier for every order. We take the carrier of the first order line as the carrier for the order.
7. **factorize-boolean** Instead of using the NEW-ORDER table to mark orders as being new, we store this using a boolean field in ORDER.
8. **defactorize-boolean** We perform the reverse transformation of factorize-boolean.
9. **precompute-aggregate** Instead of computing the total amount of an order dynamically, we want to precompute this value and store it in the ORDER table.

Discussion We now briefly discuss the completeness of our benchmark with regard to our criteria, and we discuss the correctness and accuracy of results obtained using our benchmark.

With regard to our criteria on functionality, our benchmark evaluates the expressivity of schema transformation mechanisms in the sense that it covers basic DDL operations, and the complex transformations as discussed in Section 4. The ability to perform application migration is only covered in the sense that stored procedures can be updated. More complex migration approaches such as schema versioning are not covered. Our criteria specify that transformations should be specified declaratively, but an implementation of our benchmark does not have to adhere to this. It is up to the implementor to evaluate this aspect. For the correctness of benchmark results, we assume that all transformations are implemented correctly, that the data is transformed correctly, and that the ACID guarantees are satisfied.

With regard to our performance criteria, our benchmark measures throughput, blocking and aborts of the TPC-C transactions. Moreover, we measure the time to commit of schema transformations, and we can detect whether schema transformations abort. Our benchmark does not cover the impact of recovery in case of system failure, and it does not cover memory consumption of schema transformations. However, our benchmark could be extended to cover these cases. Finally, note that for accurate results, a sufficiently large TPC-C instance should be used to show the impact of the schema transformations. Moreover, the database should be given sufficient time to warm up, and no background tasks should be running.

6 IMPLEMENTATION AND RESULTS

As the main topic of this paper is the presentation of the benchmark, an in depth analysis of our experimental results is outside the scope of this paper. However, in this section we briefly discuss our implementation of the benchmark, and provide example results. A detailed analysis of our experimental results can be found in our technical report (Wevers et al., 2014).

We have implemented benchmark scripts for PostgreSQL and MySQL, which can be accessed from our website[5]. We use the TPC-C implementation HammerDB[6] to create the TPC-C database and to provide stored procedures. We generate one TPC-C database for each DBMS, which we backup once, and then restore in the setup phase of every experiment. Before starting the introduction phase of the experiment, we let the TPC-C benchmark run for ten seconds, as to give the DBMS some time to warm up. Finally, to generate load on the system, and to measure the TPC-C performance, HammerDB provides a driver script. However, as this script does not perform logging of transactions, we have ported the script to Java and we have added logging facilities.

Figure 2 shows typical results for PostgreSQL and MySQL for the complex transformation cases. While PostgreSQL supports transactional DDL and can perform many DDL operations instantaneously, the DDL operations do still take a full table lock, which persist during the transaction. Moreover, bulk UPDATE statements also take a full table lock, and can take a long time to complete. The effect is that the TPC-C workload is completely blocked. In contrast to PostgreSQL, MySQL cannot perform DDL operations in-

[5] wwwhome.ewi.utwente.nl/~weversl2/?page=ost
[6] hammerora.sourceforge.net

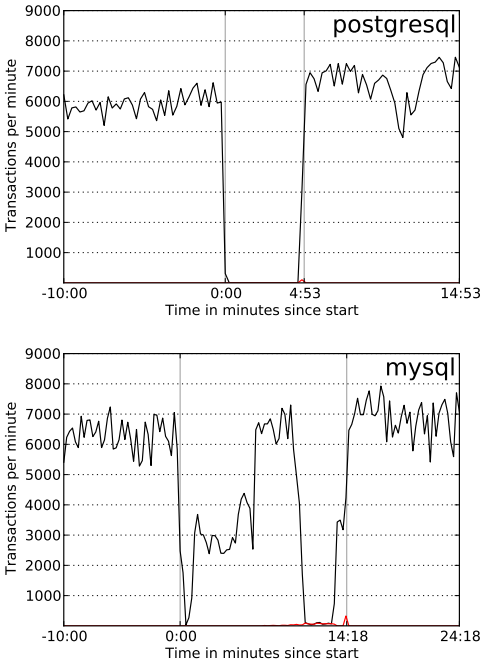

Figure 2: Benchmark results for PostgreSQL and MySQL for the `defactorize` case.

stantaneously, but it can perform some of them online. However, MySQL commits every DDL operation immediately after it is executed, which means that stored procedures may need to be updated after every transformation step. In Figure 2, MySQL shows a short period of blocking at the beginning of the transformation when adding a column, and we see reduced throughput while the column is being added. The last phase of the transformation involves an UPDATE, where we see that MySQL blocks the TPC-C workload.

7 CONCLUSIONS

Current DBMSs have poor support for non-blocking online schema transformations beyond basic transformations. The literature describes a number of methods for non-blocking schema transformations. However, these techniques do not cover all cases, and generally do not compose without exposing intermediate states of the transformation. While complex transformations are generally possible by adapting programs to work on intermediate states, these transformations are non-declarative, and require significant development effort to implement.

While we do not provide solutions to solve this problem, we have provided criteria that clarify the problem, and we have specified characteristics of an ideal solution. In particular, we would like to see that DBMSs provide a mechanism for declarative and composable non-blocking schema transformations that satisfy the ACID properties. We have provided a benchmarking methodology together with a concrete benchmark to evaluate schema change mechanisms with regard to our criteria. With this benchmark, we challenge the database community to find solutions to allow transactional non-blocking schema transformations.

REFERENCES

Chen, P. P.-S. (1976). The Entity-relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36.

Curino, C. A., Moon, H. J., Deutsch, A., and Zaniolo, C. (2010). Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 4(2):117–128.

Curino, C. A., Tanca, L., Moon, H. J., and Zaniolo, C. (2008). Schema evolution in wikipedia: toward a web information system benchmark. In *ICEIS*, pages 323–332.

Garcia-Molina, H. and Salem, K. (1987). Sagas. In *SIGMOD '87*, pages 249–259. ACM.

Gupta, A. and Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18.

Løland, J. and Hvasshovd, S.-O. (2006). Online, Non-blocking Relational Schema Changes. In *EDBT '06*, pages 405–422, Berlin, Heidelberg. Springer-Verlag.

Neamtiu, I., Bardin, J., Uddin, M. R., Lin, D.-Y., and Bhattacharya, P. (2013). Improving Cloud Availability with On-the-fly Schema Updates. COMAD '13, pages 24–34. Computer Society of India.

Neamtiu, I. and Dumitras, T. (2011). Cloud software upgrades: Challenges and opportunities. In *MESOCA '11*, pages 1–10. IEEE.

Rae, I., Rollins, E., Shute, J., Sodhi, S., and Vingralek, R. (2013). Online, Asynchronous Schema Change in F1. In *VLDB '13*, pages 1045–1056.

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37:383–393.

Ronström, M. (2000). On-Line Schema Update for a Telecom Database. In *ICDE '00*, pages 329–338. IEEE.

Sockut, G. H. and Iyer, B. R. (2009). Online Reorganization of Databases. *ACM Computing Surveys*, pages 14:1–14:136.

TPC (2010). TPC Benchmark C Standard Specification. `www.tpc.org/tpcc/spec/tpcc_current.pdf`. Accessed 19 may 2015.

Wevers, L., Hofstra, M., Tammens, M., Huisman, M., and van Keulen, M. (2014). Towards Online and Transactional Relational Schema Transformations. Technical Report TR-CTIT-14-10, University of Twente.