Towards Online and Transactional Relational Schema Transformations

Lesley Wevers, Matthijs Hofstra, Menno Tammens, Marieke Huisman, Maurice van Keulen

University of Twente

ABSTRACT

In this paper, we want to draw the attention of the database community to the *problem of online schema changes*: changing the schema of a database without blocking concurrent transactions. We have identified important classes of relational schema transformations that we want to perform online, and we have identified general requirements for the mechanisms that execute these transformations. Using these requirements, we have developed an experiment based on the standard TPC-C benchmark to assess the behaviour of existing systems. We look at PostgreSQL, which does not support online schema changes; MySQL, which supports basic online schema changes; and pt-online-schema-change, which is a tool for MySQL that uses triggers to implement online schema changes. We found that none of the existing systems fulfill our requirements. In particular, existing non-blocking solutions can not maintain the ACID guarantees when composing schema transformations. This leads to intermediate states being exposed to database programs, which are non-trivial to handle correctly. As a solution direction, we propose *lazy schema transformations*, which can naturally be composed into complex schema transformations that properly guarantee the ACID properties, and which have minimal impact on concurrent transactions.

1. INTRODUCTION

Software is in constant need of maintenance, adaptation and extension. For applications storing and maintaining data in a database, a software change often involves restructuring of data, i.e., a schema change with an accompanying conversion (or migration) of the data. However, current relational database systems are ill-equipped for changing the structure of the data while the database is in use, causing downtime for these applications. This is a real problem in systems that need 24/7 availability [6], such as telecommunication systems, payment systems and control systems. Unavailability can lead to inconveniences ranging from an e-mail system being down, 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 can not be retrieved. Not only is this a problem in its own right, evolution of software is also slowed down as developers tend to avoid making changes because of the downtime consequences.

Problem Statement. The SQL standard, which is supported by most relational database systems, allows the execution of schema changing commands such as `ALTER TABLE` concurrently with other transactions. However, current relational database management systems can not really execute these transformations in parallel with other transactions. For example, PostgreSQL, one of the most advanced open source databases, takes a global table lock when changing the schema of a table. The effect is that concurrent transactions completely halt until the (possibly long) execution of the schema change has finished. The urgency to solve this problem is evident by a multitude of tools developed in industry, such as pt-online-schema-change[1], oak-online-alter-table[2], and the online-schema-change tool used at Facebook[3]. Recently, also MySQL announced limited support for online schema changes[4], and Google has added support for online schema changes to their F1 database [8]. In this paper, our goal is to show the extent of the problem, and to draw the attention of the database community to the problem of online schema changes.

Approach. To show the extent of the problem we experimentally investigates how existing systems actually cope with online schema changes. As representatives we chose the two most popular open-source database systems: PostgreSQL, which does not support online schema changes, and MySQL, which has limited support for online schema changes. In addition, we have investigated pt-online-schema-change, which is a tool for MySQL that uses triggers to implement online schema changes, and which serves as a representative for similar tools that are based on the same principles. We look at simple operations such as adding and removing columns, but also complex operations like changing the cardinality of a relationship, or changing the primary key of a table. Our results show that current solutions are insufficient for complex schema changes, as they do not allow non-blocking schema transformations to be composed while

[1] `http://www.percona.com/doc/percona-toolkit/2.1/pt-online-schema-change.html`

[2] `http://openarkkit.googlecode.com/svn/trunk/openarkkit/doc/html/oak-online-alter-table.html`

[3] `https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932`

[4] `http://dev.mysql.com/doc/refman/5.6/en/innodb-online-ddl.html`

guaranteeing correctness. As a solution to this problem, we propose *lazy schema transformations*, which can provide this kind of composition in a natural way.

Requirements. We have identified important classes of relational schema transformations that we believe should be supported by any online schema transformation mechanism. These classes serve as a basis for our experiment, where we experimentally investigate a representative transformation for every class. We discuss these classes in Section 2. We do not only investigate simple schema transformations, but also complex schema transformations such as splitting and joining tables. To identify important schema transformations, we consider database schemas that are implementations of entity-relationship models. Important transformations in this setting are logical transformations corresponding to changes in the entity-relationship model, and physical transformations that allow changes to the implementation of an entity-relationship model as a relational database.

Furthermore, we have identified general requirements that an online schema transformation mechanism should satisfy, which we use to evaluate the behaviour of existing systems. These requirements are discussed in Section 3. We assert that, ideally, schema transformations should be performed *transactionally*, i.e., satisfying the ACID properties, and with *minimal interference* to concurrent transactions. Moreover, we require that non-blocking schema transformations are composable, to allow complex schema transformations. In Section 4 we discuss the state of the art in strategies for performing online schema transformations, and evaluate these strategies based on our requirements.

Experiment. Based on these requirements, we have set up an experiment to evaluate the behaviour of existing systems when performing schema changes, which we discuss in Section 5. Our experiment is based on the industry standard TPC-C benchmark, which models a complete order placement system. The TPC-C schema shows sufficient diversity to implement interesting schema changes, while it is small enough to implement these schema transformations for a large number of cases. For each of the relational schema transformation classes described in Section 2, we pick a representative schema transformation in the TPC-C schema, we perform this schema transformation while the TPC-C benchmark is running, and we measure the impact on the TPC-C transaction throughput.

Results. Section 6 presents the results of our experiment. Our main conclusion is that existing solutions are insufficient for complex transformations, as they do not allow non-blocking schema transformations to be composed using transactions. As a solution direction, we discuss *lazy schema transformations*, which allow instantaneous access to data in the new schema by transforming data on demand. This is beneficial for complex transformations, as lazy transformations can naturally be composed. In Section 7 we discuss the idea in more detail, where we also discuss challenges in the implementation of this approach, and our ongoing work.

Contributions. The contributions of this paper are:

- Drawing attention to the problem that database systems cannot perform schema changes sufficiently concurrent with other transactions by experimentally investigating the extent and severity of this problem..
- Criteria for evaluating online schema transformation mechanisms in general, and for the relational data model in particular.
- An experimental investigation of the behaviour of existing systems: PostgreSQL, which does not support online schema changes, MySQL, which supports basic online schema changes, and the pt-online-schema-change tool for MySQL, as a representative for tools that implement online schema changes on existing systems using triggers.
- A solution direction for complex non-blocking transformations based on lazy schema transformations.

2. RELATIONAL TRANSFORMATIONS

In this section we identify important classes of relational schema transformations that could be required in practice, and which we use in our experiments. We do not make any a priori assumptions about the difficulty of schema transformations or the capabilities of existing systems. As such, the transformations that we consider range from very simple single-relational transformations to complex multi-relational transformations. To identify important schema transformations, we consider databases that are implementations of Entity-Relationship (ER) models. We consider logical transformations that correspond to changes in the ER model, and physical transformations that correspond to changes in the implementation decisions of these ER models.

ER Models and Implementations. ER modelling is a standard method for high-level modelling of databases [1]. 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. For our experiments, we consider logical transformations on a relational database that correspond to the following transformations on ER models:

- 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 transformations that needs no database schema change, but only a data transformation. Such schema transformations correspond to normal bulk 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 system generated 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 a important subset of schema changes with which we can experimentally compare the behaviour of different online schema change mechanisms.

3. GENERAL CRITERIA

In this section we discuss general criteria on the mechanisms provided by a system to support non-blocking schema transformations. We use these criteria as the basis for our experiment to compare and evaluate existing systems, as discussed in Section 5. We make no assumptions about the specific strategies used to perform online schema transformations. Moreover, these criteria also apply to data models other than the relational model. We specify criteria both on the functionality of schema transformations and on their performance. For every criterion, we discuss the *ideal* behaviour of the schema transformation mechanism, and we also discuss what behaviour we would still consider *acceptable* for systems that must be online 24/7.

Functional Criteria. First, we specify criteria on 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, the subset of transformations that we have discussed in the previous section allows transformations between any relational schema that is an implementation of an ER model.

2. **Transformation of Data**. After executing a schema change on a database, 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. In particular, we expect the following properties for schema transformation transactions:

   (a) **Serializability**. The execution of transformations must have serializable semantics. If this is not the case, concurrent transactions may observe partially transformed states, i.e., where data is available in two different schemas.

   (b) **Failure Atomicity**. Schema changes must be robust in the event of 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.

   (c) **Composability**. Systems must allow transactional composition of basic schema transformations into more complex transformations, while maintaining transactional guarantees. If this is not possible, intermediate schemas are visible to other transactions, which then have to handle these intermediate states. Moreover, system failure may leave the system in an intermediate state after recovery. For some applications these issues may be acceptable, but handling them comes at the cost of additional development effort.

   (d) **Adapting Programs**. There must be a mechanism to ensure that database programs can adapt to the new schema, and continue to operate correctly during and after the schema transformation. This can for example be achieved by updating stored procedures as part of a schema transformation. Alternatively, the system can translate existing stored procedures automatically to the new schema, or translate operations on the old schema to operations on the new schema.

Performance Criteria. Second, we specify criteria on 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** Just like any other heavy transaction, executing a schema change concurrently with other transactions will have an impact on those transactions. We distinguish several kinds of impact and discuss what may be acceptable or not:

   (a) **Blocking**. Transactions should always be able to make progress independent of the progress of concurrent schema transformations. Ideally, a schema transformation should never block the execution of concurrent transactions, with the exception of possibly very short periods of time. Similarly, 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, slowdown is independent from the complexity of transformations, i.e., slowdown has a predictable upper bound, and more complex transformations can be performed without causing additional slowdown. Slowdown that depends on the complexity of the transformation is acceptable to a certain extent, however, this may impose limitations on the complexity or number of transformations that can be performed concurrently.

   (c) **Aborts**. Ideally, schema transformations never cause transactions to abort. Especially for long running transactions such aborts are problematic. Due to their nature, it is acceptable to abort optimistically executed transactions, however, ideally the database system can transparently and successfully retry those transactions without any severe consequences for the application programs who issued these transactions.

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

   (a) **Time to Commit**. The time to commit defines how long it takes for the results of a schema transformation to be visible to other transactions. For very large databases, the time to commit can become very long, which can be unacceptable in time critical situations. 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. For mechanisms that commit only after all data has been transformed, the time to commit is about the same as the transformation time, which should obviously be minimal. However, we also want to consider lazy transformation of data, i.e., transformation of data *after* the transformation has been committed. In this case the time to commit is generally much shorter than the actual transformation time, and as long as concurrent transactions are not impacted, the actual 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 immediately after starting the transformation 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**. Ideally, a schema transformation requires only a constant amount of additional memory. However, it could be acceptable if more memory is needed, as long as this is within acceptable limits.

Discussion. This list of criteria is new, but we are not the first to define requirements for online schema transformations. In a survey on online reorganization of databases, Sockut also discusses requirements for strategies that perform online reorganizations [10]. 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. 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 transactional guarantees and satisfying the performance requirements. In addition, we also require that database applications can be changed as part of a schema transformations. 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 database, which may choose to make progress if this reduces impact on running transactions.

4. STATE OF THE ART

In this section we briefly discuss the state of the art in techniques for online schema transformations. First, we discuss Ronströms method [9], which is used by tools such as pt-online-schema-change. Next, we discuss the log redo method by Løland and Hvasshovd [4], which improves on Ronströms method. We also briefly discuss recent work on online schema changes in the Google F1 database [8], and work on lazy schema transformations by Neamtiu et al. [5].

On-line Schema Update for a Telecom Database. Ronström describes a method for online changing of columns, adding indices, and horizontally and vertical splitting and merging of tables by using database triggers [9]. The idea behind this method is as follows. First, new tables that match the desired schema are created. Next, triggers are created on the original tables that ensure that any update on the old tables are reflected on the new tables. Next, data is copied from the old tables to the new tables in small batches, while performing the desired schema transformation on the data. Finally, after copying the data, the old tables are removed and the new tables are used instead.

An interesting benefit of Ronströms method is that it is easy to integrate with existing database systems. For instance, pt-online-schema-change is based on the techniques by Ronström, though it does not implement horizontal and vertical splitting of tables. Another benefit is that constructing the new schema next to the old schema allows programs to be tested on the new schema while the old schema is still in use. Moreover, long running transactions can continue executing on the old schema, while the new schema is in use. However, a drawback of this method is that memory is required to maintain two tables, and that the use of database triggers decreases the performance for all database updates. Moreover, copying the database slows down other transactions, and the new schema can not be used until copying is complete.

Ronström's method only provides ACID guarantees for individual transformations. For complex transformations that consist of multiple schema transformations, Ronström proposes the use of SAGA's. The idea of SAGA's 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 [3]. While SAGA's provide failure atomicity for composed transactions, they do not provide serializable semantics. This means that database programs must be able to handle intermediate states.

Online, Non-blocking Relational Schema Changes. Løland and Hvasshovd improve on Ronström's method by introducing *log redo* as an alternative to using triggers [4]. Similar to Ronström's method, a transformed copy of a source table is made without locking the source table, but their method differs from that of Ronström in that they do not use triggers to synchronize the source and target tables. In Løland and Hvasshovd's method, all transactions on the source table are logged, and after copying is done, inconsistencies in the copied table are repaired by propagating the logged changes to the copied table. Although no direct comparison has been made with Ronströms method, experiments by the authors show that their method has minimal impact on the performance of concurrent transactions.

Compared to Ronström's method, a drawback of log redo is that it has to be implemented at the database level, and to our knowledge, no implementation is available for mainstream database systems. Another difference from Ronströms method is that the data in the old schema is not synchronized with the data in the new schema after the transformation has completed. Thus losing the ability to perform tests on the new schema, and long running transactions cannot continue execution on the old schema.

Online Schema Change in F1. Rae et al. investigate online schema change in the Google F1 database [8], 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 nodes can lead to incorrect behaviour. Their solution is to split a complex schema change into a sequence of simpler schema changes, of which each guarantees not to cause conflicting operations on the distributed state. For example, deleting a column can lead to inconsistencies if some nodes are still writing new data. However, if the column is first made delete-only, and is deleted later, this can not occur. Their work essentially extends the idea of SAGA's to allow consistent operations in a distributed setting. This means that complex schema transformations still expose intermediate states to database programs.

On-the-fly, March-Forward Schema Updates. Neamtiu et al. propose on-the-fly (lazy) updates of databases [6]. 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 SQLLite, and have performed experiments, showing very short time to commit and low overhead.

The main benefit of their approach is that schema transformations have very low time to commit, as the work is performed lazily. 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.

Discussion. To our knowledge, none of the existing techniques for non-blocking schema transformations allow for composition while maintaining ACID guarantees. The use of SAGA's, which do not provide serializability, is the state of the art for composed transformations. However, as we discuss in Section 7, we believe that lazy transformations are a promising approach to solve this problem.

5. EXPERIMENTAL SETUP

We use the TPC-C benchmark specification as a basis for an experiment to investigate to what degree schema transformations can be performed online in existing database systems. To investigate the behaviour of existing systems, we perform a schema transformations while concurrently running the TPC-C benchmark on the database that is being transformed. More concretely, we define two extensions to the TPC-C benchmark. First, we define concrete schema transformations for the TPC-C schema that are representative for the transformations identified in Section 2. Second, we describe a benchmark process that specifies how our experiments should be performed, and we discuss the interpretation of the results with regard to the non-functional requirements as discussed in Section 3 are satisfied.

The TPC-C Benchmark. The TPC-C benchmark is an industry standard OLTP benchmark that simulates an order-entry environment, where users can perform transactions to enter and deliver orders, record payments, check the status of orders, and monitor the level of stock at warehouses[5]. 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

[5]`http://www.tpc.org/tpcc/spec/tpcc_current.pdf`

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-a	*Copy ORDER-LINE to ORDER-LINE-B.* Drop ORDER-LINE. Change the stored procedures to use ORDER-LINE-B instead of ORDER-LINE.
remove-relation-b	*Copy ORDER-LINE to ORDER-LINE-B.* Drop ORDER-LINE-B.
Column Transformations	
remove-column-a	*Copy OL_AMOUNT to OL_AMOUNT_B.* Drop OL_AMOUNT_B.
remove-column-b	*Copy OL_AMOUNT to OL_AMOUNT_B.* Drop OL_AMOUNT. Change the stored procedures to use OL_AMOUNT_B instead of OL_AMOUNT.
add-column-a	Create OL_TAX as NULLABLE of the same type as OL_AMOUNT.
add-column-b	Create OL_TAX as NULLABLE of the same type as OL_AMOUNT. Change the stored procedures to set OL_TAX to OL_AMOUNT upon insertion.
add-column-default-a	Create OL_TAX as NOT NULL of the same type as OL_AMOUNT.
add-column-default-b	Create OL_TAX as NOT NULL of the same type as OL_AMOUNT. Change the stored procedures to set OL_TAX to OL_AMOUNT upon insertion.
add-column-derived-a	Create OL_TAX as NOT NULL and initial value OL_AMOUNT × 0.21.
add-column-derived-b	Create OL_TAX as NOT NULL and initial value OL_AMOUNT × 0.21. Change the stored procedures to set OL_TAX to OL_AMOUNT upon insertion.
rename-column-a	Rename column OL_AMOUNT to OL_AMOUNT2. Change the stored procedures to use OL_AMOUNT2 instead of OL_AMOUNT.
rename-column-b	*Copy column OL_AMOUNT to OL_AMOUNT2* Rename column OL_AMOUNT2 to OL_AMOUNT3.
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 and Uniqueness-Constraint Transformations	
create-index-a	Create an index on OL_DELIVERY_D.
create-index-b	Create an index on OL_I_ID.
remove-index-a	*Execute create-index-a.* Drop the index created by create-index-a.
remove-index-b	*Execute create-index-b.* Drop the index created by create-index-b.
create-unique-a	*Create a column OL_U, and fill this with unique values.* Add a uniqueness constraint on OL_U.
create-unique-b	*Create a column OL_U, and fill this with unique values.* Add a uniqueness constraint on OL_I_ID and OL_U.
remove-unique-a	*execute create-unique-a.* Drop the uniqueness constraints created by create-unique-a.
remove-unique-b	*Execute create-unique-b.* Drop the uniqueness constraints created by create-unique-b.
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-a
Multi-Relational Transformations	
change-primary	Add an auto increment column O_GUID. Add a column OL_O_GUID, and set its value to the O_GUID of the corresponding order. 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 >= (ST_O_ID - 20).
split-relation	Create a table ORDER-ORDER-LINE with columns OOL_O_ID, OOL_D_ID, OOL_W_ID, OOL_OL_ID and OOL_NUMBER. Create auto increment column OL_ID 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.
DATA TRANSFORMATIONS	
change-data	Set OL_AMOUNT to OL_AMOUNT × 2.

Table 1: Experiment specification.

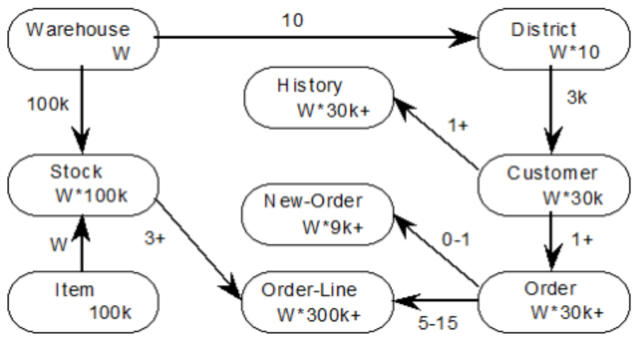


Figure 1: TPC-C Benchmark Schema.

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.

For our experiment, we use the TPC-C benchmark essentially unmodified. This means that existing TPC-C benchmark implementations can be used. However, we assume that the TPC-C transactions are implemented as stored procedures that can be changed as part of a schema transformation.

Experiment Definitions. In this section we describe our experiments at a high level. For the full specification of the experiments, we refer the reader to the technical report that accompanies this paper [15]. Our specifications are database independent, only describing the semantic transformation at the relational level. We do not prescribe a specific implementation of the transformations, as we do not want to preclude the use of features provided by the database system. For each experiment specification, if possible, we have implemented versions for PostgreSQL, MySQL and pt-online-schema-change, which can be accessed from our website[6].

We define most schema transformations on the ORDER-LINE table, which is the largest table in populated TPC-C databases. Transformations affecting multiple tables are mostly performed on the ORDERS and ORDER-LINE tables. Some transformations have a preparation part, where the TPC-C schema is transformed prior to the actual transformation that we want to evaluate. We use this to experiment with inverse transformations of many multi-relational transformations. For instance, we have an experiment 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.

Many transformations change the stored procedures, so that the TPC-C benchmark can continue running on the transformed schema. For many transformations, we have implemented two versions. First, a version where the transformation does not affect the stored procedures of the TPC-C benchmark, such as adding a column that is not used by the TPC-C stored procedures. Second, a version where the stored procedures are affected, such as removing a column used by the TPC-C benchmark while changing the stored procedures to use another column.

[6]`http://wwwhome.cs.utwente.nl/~weversl2/?page=ost`

Based on the requirements as defined in Section 2, we have defined schema transformations in the following categories:

- **Relations** Creating, deleting and renaming relations.
- **Columns** Adding columns without and without default values, adding columns with values derived from other columns, renaming columns, removing columns and changing the types of columns.
- **Indices** Creating and removing indices and uniqueness constraints.
- **Constraints** Creating and removing constraints.
- **Data** Changing data in bulk transactions.
- **Multi-relation transformations** We perform the following operations in both directions:
  - **change-primary**: Changing the primary key of ORDER-LINE from a composite key to a serial integer surrogate key.
  - **split-relation**: Encode the relation between ORDER and ORDER-LINE in a separate table ORDER-ORDER-LINE by splitting ORDER-LINE.
  - **defactorize**: Moving CARRIER_ID in ORDER to ORDER-LINE, such that each order line can have a different carrier.
  - **defactorize-boolean**: Encoding entries in NEW-ORDER, which models a set of new orders, as a boolean field IS_NEW in ORDER.
  - **precompute-aggregate**: Precomputing the total amount of an order, instead of computing it upon every request.

Table 1 shows the detailed specification of our experiments. We prefix all column names with the abbreviated table name, to avoid having to write the full table name. For example, OL_NAME is a column in order line. and O_NAME is a column in ORDERS. The preparation part of experiments is presented in *italics*. For some experiments, the preparation part is the result of the transformation performed by another experiment.

Process. For our experiments we use a standard TPC-C benchmark load. The load consists of multiple threads continuously executing randomly chosen TPC-C stored procedures on the database. After each transaction attempt, the type of transaction, its starting time and its end time are logged. Failed transactions are logged with type `ERROR`.

The execution of an experiment is done in multiple phases: setup of the database, preparation, intro, transformation, and outro. First, we create a TPC-C database. Then, if necessary, we perform a preparation transformation. Then, we start the TPC-C benchmark load, where we log the executed transactions. Next, in the intro phase we wait for 10 minutes before starting the transformation, while measuring the baseline TPC-C performance. After the 10 minutes have passed, we start the execution of the actual transformation. We log the transformation begin and end time. Finally, we wait for another 10 minutes to measure the performance of TPC-C in the outro phase.

To analyse the behaviour of the online schema change mechanism, we present the result of an experiment as a histogram that plots the TPC-C transaction execution rate in time intervals of fixed length. In the histogram, we mark the start and commit time of the transformation with vertical lines. If a schema transformation is non-blocking, we expect to see either no effect on the TPC-C execution rate or a reduction in performance during the transformation. If a schema transformation is blocking, we see a zero TPC-C execution rate for extended periods of time.

6. EXPERIMENTAL RESULTS

In this section we present the results of performing our experiments on three systems. First, we look at PostgreSQL, which is one of the most advanced open source database systems. PostgreSQL does not support online schema transformations, except for the creation of indices, but it does allow certain operations to be performed instantaneously. Next, we look at MySQL, which is interesting as it has recently added support for online schema transformations, and which is one of the first to do so. Finally, we look at pt-online-schema-change, which uses Ronströms method as discussed in Section 4, to perform online schema transformations on MySQL databases using triggers. These kinds of tools are interesting, as they can add support for non-blocking schema changes to any database system that supports triggers.

We use the TPC-C implementation HammerDB[7] to provide scripts for generating the schema and to populate the database, and to provide stored procedures. HammerDB supports a number of database implementations, including both PostgreSQL and MySQL. We use HammerDB to generate one database per database implementation, which we backup once, and which we 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 database system 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. For all experiments, we generate a database of 30 warehouses, and we use 64 threads of load on the database. We do not spawn new threads to start other transactions while a thread is blocked.

For the experiments we have used a quad-core Intel i7 machine with 16GB of RAM and a solid-state drive. For the software we used Ubuntu Linux kernel 3.20.0, PostgreSQL version 9.1.14, MySQL version 5.6.20, pt-online-schema-change version 2.2.11 and HammerDB version 2.14.

In this section we only discuss the most interesting experimental results. For the results of all implemented cases see Appendix A.

6.1 PostgreSQL

First, we look at PostgreSQL, which is one of the most advanced open source databases. PostgreSQL supports transactional schema transformations, but except for online creation of indices, it does not have support for online schema transformations. However, many transformations in PostgreSQL can be performed instantaneously. We discuss the results of our experiments in the paragraphs below.

Relations. As would be expected, creating, deleting and renaming relations are instantaneous operations that do not affect running transactions.

Columns. Removing and renaming columns are instantaneous operations, as well as adding columns without a default value. These operations take about a second to execute, but they are blocking. In practice this means that if these transformations are composed with long running transformations, the composed transformation is blocking as well. Interestingly, adding a column with a default value blocks access to the database for a much longer time, about 73 seconds, as shown in Figure 2a. Adding a derived column is a more complex operation, as it requires an `ALTER TABLE` to add a column, and an `UPDATE` to fill the column. As the `ALTER TABLE` takes a table lock, the whole operation is blocking, as shown in Figure 2b. Finally, changing the type of a column is a blocking operation, as shown in Figure 2c.

Indices and Uniqueness Constraints. PostgreSQL allows indices to be created online using the `CREATE INDEX CONCURRENTLY` statement. Figure 2d shows the behaviour of PostgreSQL when creating an index, which took about 60 minutes to complete. Figure 2e shows the same situation, but omitting the `CONCURRENT` keyword, so that PostgreSQL creates the index while blocking access to the table. Now, creating the index only takes about 35 seconds. I.e. in this case, creating the index online is about 100 times slower than creating the index offline. However, we should note that this can be explained by the system being under constant heavy load. Removing indices and uniqueness constraints are instantaneous operations.

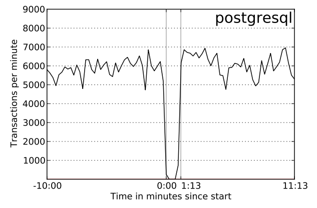
Interestingly, PostgreSQL allows uniqueness constraints to be added online. An index is build up concurrently, and all rows that are inserted are checked against the index that has been built so far. If the constraint is violated, building the index is aborted. Figure 2f and Figure 2g show the behaviour of PostgreSQL when adding uniqueness constraints concurrently. Note that the execution time of create-unique-a is quite long, about 145 minutes, but at no point do the TPC-C transactions block.

Constraints. Adding a `CHECK` constraint in PostgreSQL is a blocking operation. However, PostgreSQL has a facility to add constraints without checking validity, and allows validating constraints later. However, in our version of PostgreSQL this only works for foreign key constraints, and not for check constraints.

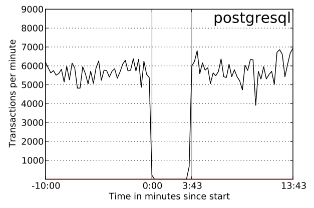
Data Transformations. For data-only schema transformations, we perform an `UPDATE` statement under `SERIALIZABLE` isolation level. In our experiments, the `UPDATE` invariably fails due to contention from the TPC-C transactions. To avoid this, we had to explicitly get an exclusive lock on the table, which leads to a blocking transformation, as shown in Figure 2h.

Multi-Relation Transformations. Any complex multi-relational transformations can not be performed online in PostgreSQL, due to the fact that `ALTER TABLE` takes a table lock which it holds for the entire transaction. For example, see Figure 2i, Figure 2j or Figure 2k. An interesting case is defactorize-boolean, as shown in Figure 2l, which does not block running transactions. The schema transformation does not update the existing table, but creates a new table from the existing table. While we could not verify

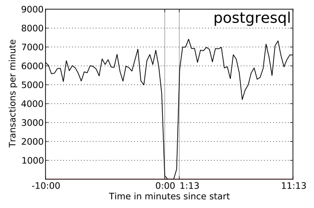
[7] `http://hammerora.sourceforge.net/`



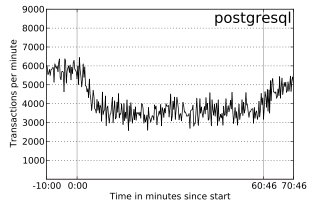
(a) add-column-default-a



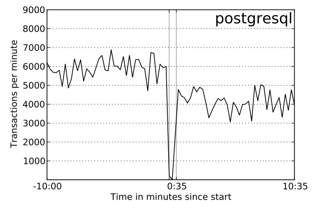
(b) add-column-derived-a



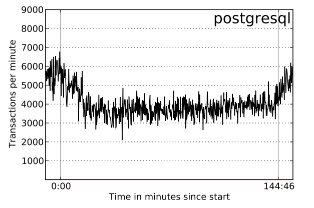
(c) change-type-a



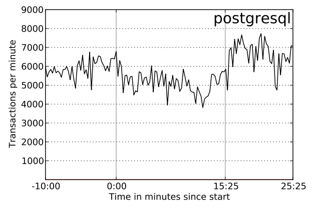
(d) create-index-b concurrent



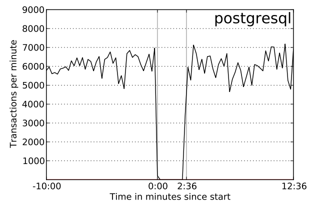
(e) create-index-b
non-concurrent



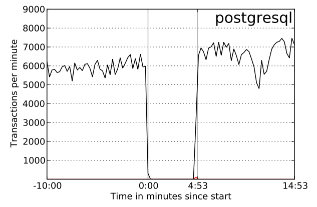
(f) create-unique-a



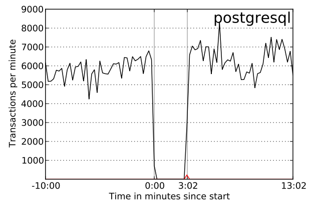
(g) create-unique-b



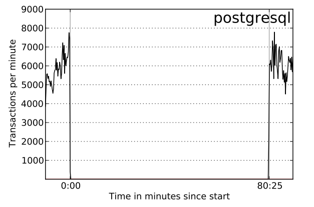
(h) change-data



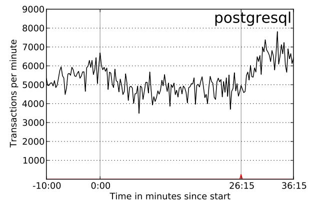
(i) defactorize



(j) split-relation



(k) join-relation



(l) defactorize-boolean

Figure 2: Experimental results for PostgreSQL. Successful transactions are plotted in black, erroneous transactions are plotted in red.

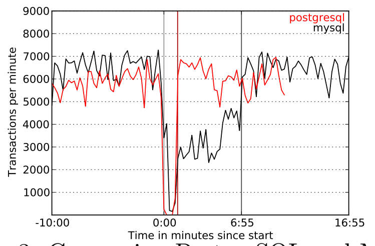


Figure 3: Comparing PostgreSQL and MySQL: add-column-default-a

this, we expect that this transformation is incorrect because the TPC-C transactions are not executed in `SERIALIZABLE` mode. As the schema transformation runs in `SERIALIZABLE` mode, it is likely that the new table is created from a snapshot, and any updates on the original table during the transformation are not reflected in the new table.

Summary. PostgreSQL shows the following behaviour with respect to our criteria:

- Schema transformations generally execute correctly due to transactional schema transformations.
- Indices can be created without blocking. All other single-statement transformations are blocking, but some can be executed instantaneously.
- Complex multi-statement transformations are blocking due to table locks.
- Explicit locking is required when using `UPDATE` statements to avoid aborts due to concurrency conflicts.

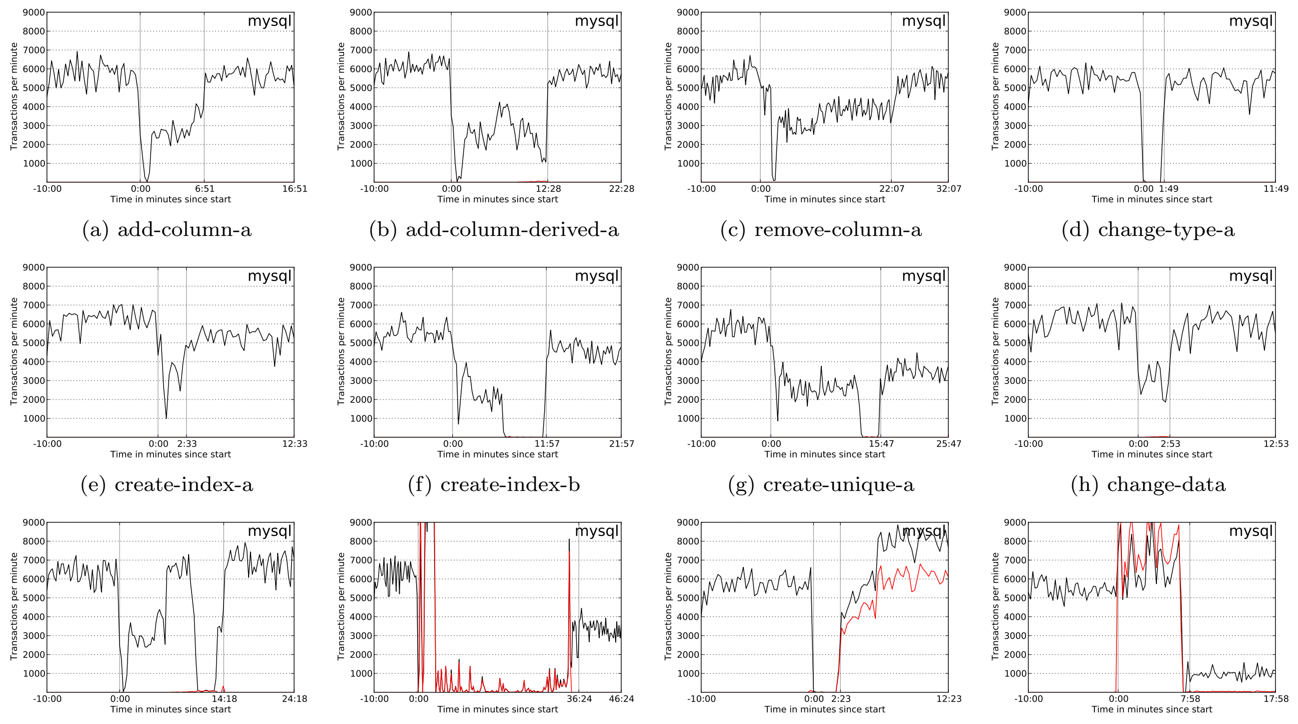
6.2 MySQL

Since MySQL version 5.6, the InnoDB storage engine features support for online schema transformations in the form of online DDL[8], which allows certain `ALTER TABLE` operations to be performed online. Some operations still require table locks, and there are brief periods where a transformations takes an exclusive lock[9]. In contrast to PostgreSQL, MySQL implicitly commits every schema transformation statement immediately after completing its execution[10]. Moreover, locks are released after a schema transformation statement is executed, so it is not possible to enforce table locks over multiple operations. This means that MySQL provides no mechanism that can guarantee correctness of multi-statement schema transformations, which has several implications:

- Intermediate states of complex schema transformations are visible to concurrent transactions.
- System failure during a schema transformation can leave a database in an intermediate state.
- Schema transformations can not be rolled back.

Moreover, we cannot atomically change a set of a stored procedures as part of a schema transformation. For experiments where we have to change stored procedures, we can do this either before or after the schema transformation. If we change the stored procedures before the transformation, stored procedures may perform incorrect operation on the old schema. Similarly, if we change the stored procedures

[8]http://dev.mysql.com/doc/refman/5.6/en/innodb-online-ddl.html
[9]http://dev.mysql.com/doc/refman/5.6/en/innodb-create-index-limitations.html
[10]http://dev.mysql.com/doc/refman/5.6/en/implicit-commit.html



(a) add-column-a (b) add-column-derived-a (c) remove-column-a (d) change-type-a

(e) create-index-a (f) create-index-b (g) create-unique-a (h) change-data

(i) defactorize (j) change-primary-a (k) split-relation (l) join-relation

Figure 4: Experimental results for MySQL. Successful transactions are plotted in black, erroneous transactions are plotted in red.

after the transformation, the old procedures may perform incorrect operations on the new schema. Alternatively, stored procedures can be dropped before a schema transformation, and re-created after the transformation to simulate locking. However, as this effectively causes blocking, we chose not to do this, and instead observe the behaviour of MySQL if we leave the stored procedures in place.

Relations. Like PostgreSQL, creating, renaming and removing relations are instantaneous operations, and do not affect running transactions.

Columns. Like PostgreSQL, renaming a column in MySQL is an instantaneous operation. Figure 4a shows the behaviour of MySQL when adding a column without a default value. In contrast to PostgreSQL, this is not an instantaneous operation. We do see the online schema change feature doing its work, as during the transformation the TPC-C benchmark can continue executing transactions. However, we also see that for a short time, no TPC-C transactions can be performed at all. In Figure 3 we compare the behaviour of MySQL and PostgreSQL when adding a column with a default value. Interestingly, we see that the initial drop in performance of MySQL has about the same duration as the complete schema transformation in PostgreSQL. This means that MySQL's online transformations can still block concurrent transactions for a significant amount of time.

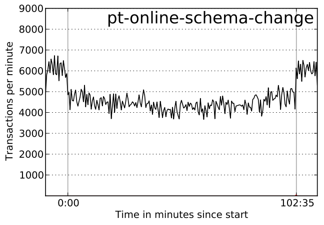
Figure 4b shows the behaviour of MySQL when adding a column whose values are derived from that of another column. This case is implemented by adding a column using `ALTER TABLE`, and then filling the column using `UPDATE` under `SERIALIZABLE` isolation. Concurrent transactions can continue executing while adding the new column, but during the `UPDATE`, we see spurious errors in the TPC-C transactions. Moreover, we doubt that the `UPDATE` is performed correctly, because MySQL does not implement true serializability.

The behaviour of MySQL when removing a column is shown in Figure 4c. Again, we see a significant drop in performance for a short period of time. Moreover, considering that PostgreSQL can do this in 1.77 seconds, MySQL is very slow at 22 minutes. However, we should note that it is likely that the high load on the system is the cause of this.

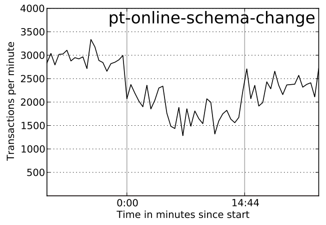
Finally, changing the type of a column is a blocking operation, as shown in Figure 4d. However, running transactions are not affected when changing the type of a column that is not touched by the TPC-C transactions.

Indices and Uniqueness Constraints. Figure 4e shows the behaviour of MySQL when creating an index concurrently on a column which is never read or updated. We see a drop in performance, but no significant slowdown. Surprisingly, when creating an index concurrently on a column that is also read and updated, the TPC-C transactions are blocked, as shown in Figure 4f. Considering the online schema change capabilities of MySQL, this is not what we expected, also because PostgreSQL is able to handle this case concurrently. When creating a uniqueness constraint, we also see that MySQL is blocking, as shown in Figure 4g.

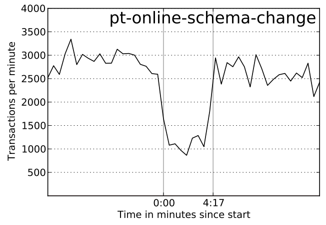
Data Transformations. Like PostgreSQL, we execute a data transformation under SERIALIZABLE isolation level in MySQL. Figure 4h shows the behaviour of MySQL when updating all values in a table. In contrast to PostgreSQL,



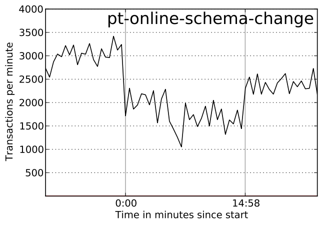
(a) add-column-a
Chunk size 1000, load 64



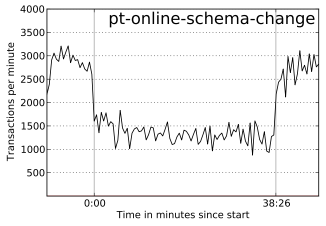
(b) add-column-a
Chunk size 1000, load 4



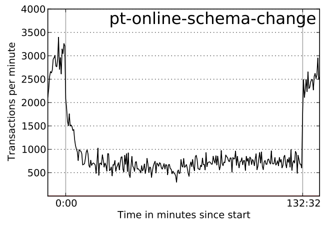
(c) add-column-a
Chunk size 10000, load 4



(d) change-type-a
Chunk size 1000, load 4



(e) create-index-b
Chunk size 1000, load 4



(f) create-unique-a
Chunk size 1000, load 4

Figure 5: Experimental results for pt-online-schema-change.

which required an exclusive lock to avoid deadlocks under serializable isolation, MySQL was able to complete the transformation without an exclusive lock. However, we do see TPC-C transaction errors during the transformation. Moreover, MySQL does not implement true serializability, but only snapshot isolation. I.e., the transformation is incorrect, as updates by TPC-C transactions during the transformation are not transformed.

Multi-Relation Transformations. It is not possible to perform online multi-relation transformations correctly in MySQL, due to the fact that MySQL performs a commit after each `ALTER TABLE` statement. Figure 4i and Figure 4j show typical behaviour during a complex transformation. Either the stored procedures perform incorrect operations on intermediate states, as is the case for defactorize, or the stored procedures cannot execute due to errors, as is the case for change-primary-a. Moreover, we were unable to successfully complete the execution of split-relation and join-relation due to `UPDATE` statements coming to a deadlock with the TPC-C transactions. This problem persisted even when an explicit `LOCK TABLES` was used on the table being updated. Figure 4k and Figure 4l respectively shows the behaviour of MySQL during the split-relation and join-relation transformations. We see many errors during the transformation, as well as errors after the transformation, due to the transformation being unable to complete.

Summary. MySQL shows the following behaviour:

- Many single `ALTER TABLE` statements can be performed transactionally.
- Some *online* `ALTER TABLE` statements can be performed without blocking, but many still block for a significant amount of time, and do no better than PostgreSQL offline schema changes. Other `ALTER TABLE` statements can not be performed online, and are thus blocking.
- Multiple-statement transformations can not be performed transactionally, including updating of stored procedures. Moreover, there is no facility to acquire explicit locks to guarantee correctness of multiple statement transformations. Consequently concurrent transactions can perform incorrect operations which can destroy data integrity, and the database can be left in an intermediate state if there is a system failure during a transformation.
- Transformations that involve `UPDATE` statements may not able to complete successfully due to deadlocks.

6.3 pt-online-schema-change

The pt-online-schema-change tool from the Percona Toolkit[11] can perform online schema changes on MySQL databases. The tool accepts an `ALTER TABLE` statement, and executes it transactionally. It is based on the technique described by Ronström [9], as we have discussed in Section 4. However, it does not support splitting and merging of tables.

A transformation in pt-online-schema-change can consist only of a single `ALTER TABLE` statement. This means that we can only modify one table at a time, and we can not update stored procedures atomically. Within the context of our experiment, pt-online-schema-change allows the following transformations:

- Adding, removing and renaming columns, and changing the type of columns.
- Adding and removing indices.
- Adding and removing uniqueness constraints.

The pt-online-schema-change tool creates a new table with the new schema, and copies the rows from the source table to this new table. Copying is done in chunks of a certain size, which can be configured using two strategies. First, a fixed chunk size can be specified, and second, a fixed time per chunk can be specified. While a chunk is being copied, the copied rows are locked for writing. A larger chunk size impacts concurrent transactions more due to locking, while a shorter chunk size slows down the schema transformation.

[11]`http://www.percona.com/doc/percona-toolkit/2.1/pt-online-schema-change.html`

Effect of Chunk Size and Load. The chunk size and the TPC-C benchmark load have a large effect on the performance of pt-online-schema-change. This is because pt-online-schema-change executes transactions in `LOW PRIORITY` mode, to minimize slowdown for concurrent transactions. Figure 5a shows the behaviour of MySQL when pt-online-schema-change is used to add a column with a TPC-C load of 64 threads and chunk size 1,000. The time to commit is very long, about 102 minutes, much longer than the 6:51 used by MySQL to perform the same operation. If we lower the TPC-C load from 64 threads to only 4 threads, and keep the chunk size at 1,000, pt-online-schema-change commits in only 14:44, as shown in Figure 5b. If we increase the chunk size to 10,000, pt-online-schema-change completes in 4:17, as shown in Figure 5c, however, we also see a reduction in TPC-C performance.

Results. Figure 5 also shows the results of the other experiments, where we used a chunk size of 1,000, and a load of 4 threads. Interestingly, pt-online-schema-change can perform some operations online that MySQL can not, such as changing the type of a column as shown in Figure 5d. We also see that pt-online-schema-change does not exhibit the initial drop in performance from which MySQL suffers. Furthermore, pt-online-schema can create indices concurrently in cases where MySQL fails, as shown in Figure 5e. It can also create unique indices online as shown in Figure 5f, however, the time to commit for this transaction was quite high at about 132 minutes, but which is still faster than PostgreSQL. In general, due to the way pt-online-schema-change works, most operations take a similar time to complete, except for creating (unique) indices.

Summary. The pt-online-schema-change tool shows the following behaviour with respect to our criteria:

- Any individual `ALTER TABLE` statement can be performed transactionally without blocking.
- There is a trade-off between fast time to commit, or minimal impact on running transactions. The impact on transactions is significant for a faster commit time, and the commit time can be much higher than MySQL's equivalent operation when choosing for lower impact on concurrent transactions.
- There is no support for transformations consisting of multiple `ALTER TABLE` statements, there is no support for `UPDATE` statements, and there is no facility to update stored procedures.

6.4 Conclusion

PostgreSQL and MySQL both show mixed results for single statement transformations, but pt-online-schema-change shows good results. The pt-online-schema-change tool has the limitation that only `ALTER TABLE` is supported, and there is no facility to update stored procedures. Moreover, none of the tested systems allow for correct complex transformations without blocking. PostgreSQL can perform complex operations correctly, but blocks. MySQL can not perform complex operations correctly, but allows transactions to continue executing. However, during online transformations MySQL still blocks for some periods of time, which can accumulate to the same time as a blocking transformation in PostgreSQL. Finally, while pt-online-schema-change works well for single `ALTER TABLE` statements, it has no support for other transformations, or complex transformations.

7. SOLUTION DIRECTION

Our survey of the state of the art and our experiments show that current database systems can (partially) handle simple schema transformations, but they can not handle complex multi-relational transformations at all. The main problem of existing techniques is that composition of non-blocking schema transformations is not possible. We believe that *lazy schema transformations* provide a promising solution to this problem, which we discuss below.

Lazy Schema Transformations. A lazy schema transformation is a schema transformation where data is transformed after the transaction has been committed. A lazy schema transformation can essentially be viewed as a query on the existing database that returns a database in the new schema. Instead of eagerly transforming data to the new schema, as current database systems do, in lazy schema transformations, individual tuples or values are transformed upon request. Data can be materialized in the new schema as a side effect of requesting data from the transformed schema, which can be sped up by transforming data during idle time.

How this better meets the requirements. Lazy schema transformations can be implemented to satisfy the ACID properties. One method is to access data through a wrapper that transforms data to the new schema upon access [5]. Another method is to implement transformations as suspended computations [11, 7]. Both methods provide the same guarantees as performing a transformation guarded by a lock, but have the advantage that data can be accessed on demand. Furthermore, instead of journaling modified values to guarantee durability, only the schema transformation definition has to be journaled. As a consequence, lazy schema transformations can be committed before they are executed, and their results are visible immediately. An interesting benefit of lazy transformations is that they can naturally be composed without causing blocking; the output of one transformation can feed directly into the next transformation. This allows complex transformations to be defined in a natural way.

Challenges. Lazy transformations have already been studied for some time in object-oriented databases [2]. Moreover, Neamtiu et al. recently investigated lazy schema transformations for relational databases [5], as discussed in Section 4. However, complex relational schema transformations are still a challenge. Some issues that still need to be solved are the following:

- Not all transformations can be performed lazily without blocking, for example, checking constraints, or creating a full index. Current database systems already have solutions to these problems, however, integrating these with lazy schema transformations without losing the aspect of lazy composition is still a challenge.
- Some operations are inherently sequential, such as generating incremental values for surrogate keys, and can not be executed lazily. We need techniques to perform similar operations in a lazy setting.

- In non-lazy updates, indices can be build before a table is in use. When transforming tables lazily, this is not possible. An interesting challenge is to see whether we can transform indices lazily.
- Compared to how database systems are traditionally implemented, lazy schema transformations require a significantly change. A big question is if this system can match the performance and scalability of existing techniques for traditional OLTP transactions.

Functional Databases. We are working on the implementation of a database based on a purely functional core, where the implementation of lazy schema transformations is one of our goals [13, 14]. Functional languages are not far from the declarative query languages provided by current systems. For instance, XQuery is a successful purely functional query language for XML databases [12]. Functional languages provide many of the same opportunities for optimization as currently performed on relational queries, making them suitable for use in the context of databases. Moreover, they also provide lazy evaluation, and the ability to define arbitrary functions in database queries.

In contrast to functional query languages, we are investigating the use of functional languages to optimize the *concurrency* of transactions that update a database. In our system, a transaction is a function $S \to S' \times R$ that takes the current state of the database S, and which produces the next state of the database S', together with an observable result R. Based on early work by Trinder [11] and Nikhil [7], we are investigating the use of lazy evaluation to achieve concurrent execution of functional transactions, which makes lazy schema transformations the natural solution to schema transformations. Additionally, we plan to investigate rewriting of functional expressions to further optimize concurrency. Moreover, as our language allows arbitrary functions to be defined in transactions, we can commit transactions that contain user defined logic. This allows more types of transactions to be performed lazily than a database system that provides a fixed set of functions.

8. CONCLUSION

Our main aim is to draw the attention of the database community to the problem of online schema transformations. We show the extend of the problem by experimentally investigating existing solutions. For our experiments, we have identified an important subset of relational schema transformations that we want to be able to perform online. Moreover, we have defined general requirements for online schema transformation mechanisms, where we assert that schema changes should be performed transactionally and with minimal interruption of concurrent transactions. We have used the standard TPC-C benchmark, augmented with schema transformations, to evaluate the behaviour of PostgreSQL, MySQL and the pt-online-schema-change tool for MySQL, which serve as representatives for the state of the art in database system implementations. Our experiments showed the following results:

- PostgreSQL can create indices without blocking concurrent transactions, and it can perform many operations instantaneously. However, for more complex transformations, PostgreSQL blocks concurrent transactions due to exclusive table locks.
- While MySQL supports online schema changes, these can still show periods of blocking, which can be as long as the blocking schema changes of PostgreSQL. Moreover, MySQL does not support transactional composition of schema transformations, which resulted in incorrect transformations in our experiment.
- The pt-online-schema-change tool shows good results for simple transformations, but it can only perform a single `ALTER TABLE` statement transactionally.

A general conclusion is that composition of non-blocking schema transformations is not possible in existing systems. The literature describes a number of methods for non-blocking schema transformations, which also includes complex transformations such as joining and splitting of tables. However, these techniques do not allow transactional composition of non-blocking schema transformations. Moreover, many of the existing solutions also have the drawback that there is a long delay between starting and committing a schema transformation. We believe that lazy schema transformations are a promising solution direction, as lazy transformations can be composed in a natural way while maintaining non-blocking behaviour. Working out the details of this idea, and implementing a working system, is still a challenge that we are working on.

9. REFERENCES

[1] P. P.-S. Chen. The Entity-relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, pages 9–36, 1976.

[2] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *VLDB '94*, pages 261–272, 1994.

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

[4] J. Løland and S.-O. Hvasshovd. Online, Non-blocking Relational Schema Changes. In *Advances in Database Technology - EDBT*, pages 405–422. Springer, 2006.

[5] I. Neamtiu, J. Bardin, M. R. Uddin, D.-Y. Lin, and P. Bhattacharya. Improving Cloud Availability with On-the-fly Schema Updates. 2013.

[6] I. Neamtiu and T. Dumitras. Cloud software upgrades: Challenges and opportunities. In *MESOCA*, pages 1–10. IEEE, 2011.

[7] R. S. Nikhil. Functional Databases, Functional Languages. In *Data Types and Persistence*, Topics in Information Systems, pages 51–67. Springer, 1988.

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

[9] M. Ronström. On-Line Schema Update for a Telecom Database. In *ICDE*, pages 329–338, 2000.

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

[11] P. Trinder. *A functional database.* PhD thesis, University of Oxford, 1989.

[12] P. Wadler. XQuery: A Typed Functional Language for Querying XML. In *Advanced Functional Programming*, LNCS, pages 188–212. Springer, 2003.

[13] L. Wevers. A persistent functional language for concurrent transaction processing. Master's thesis, 2012.

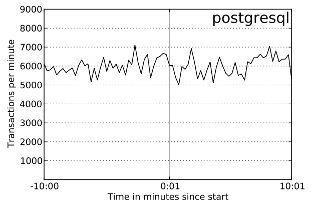
[14] L. Wevers. Persistent functional languages: toward functional relational databases. In *SIGMOD PhD symposium*, pages 21–25. ACM, 2014.

[15] L. Wevers, M. Hofstra, M. Tammens, M. Huisman, and M. van Keulen. Towards Online and Transactional Relational Schema Transformations. Technical Report TR-CTIT-14-10, 2014.

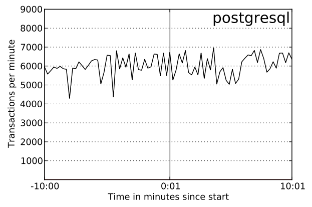
APPENDIX

A. ALL EXPERIMENTAL RESULTS

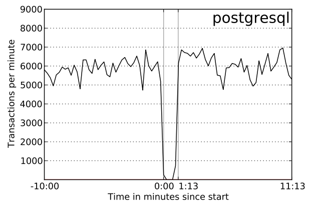
Below, we show the experimental results of all implemented cases. Figure 6 and Figure 7 show the results for PostgreSQL. Figure 8 and Figure 9 show the results for MySQL, and Figure 10 shows the results for pt-online-schema-change.



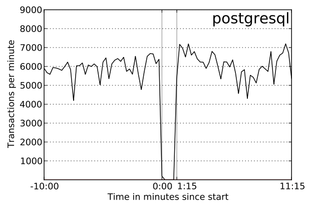
(a) add-column-a



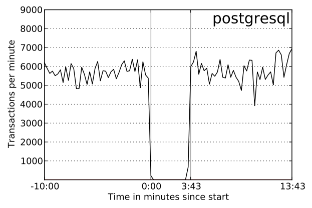
(b) add-column-b



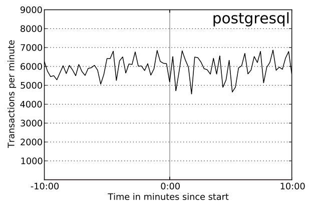
(c) add-column-default-a



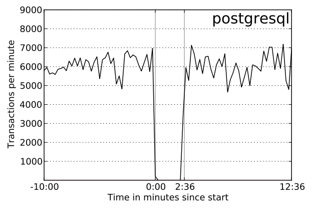
(d) add-column-default-b



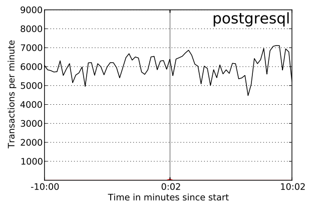
(e) add-column-derived-a



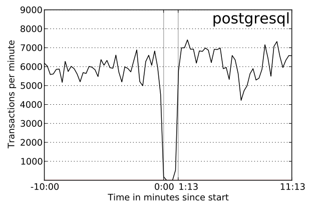
(f) add-column-derived-b



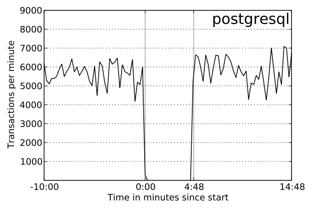
(g) change-data



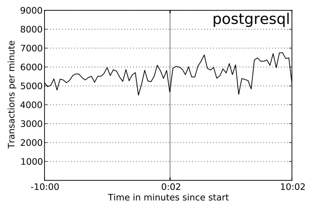
(h) change-primary-a



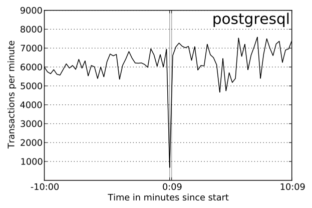
(i) change-type-a



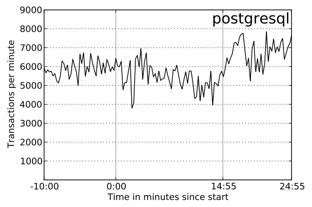
(j) change-type-b



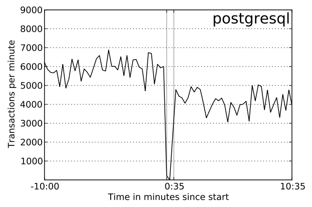
(k) create-constraint-a



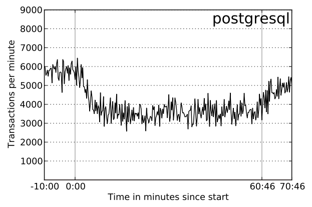
(l) create-index-a-nc



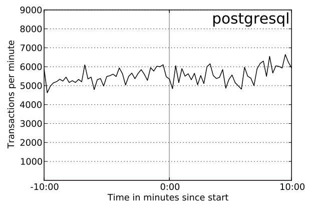
(m) create-index-a



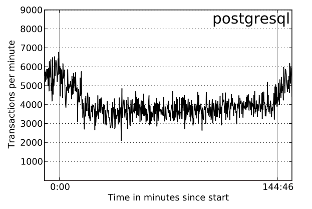
(n) create-index-b-nc



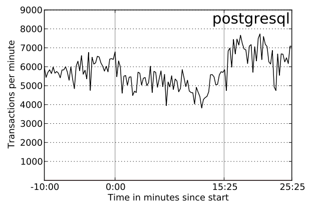
(o) create-index-b



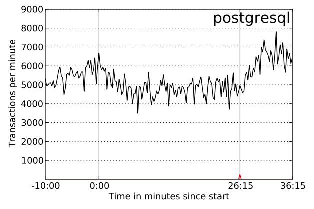
(p) create-relation



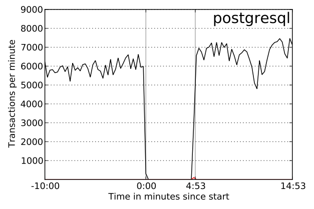
(q) create-unique-a



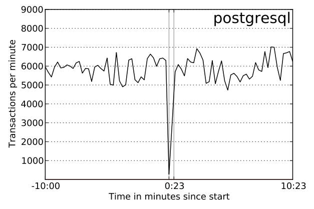
(r) create-unique-b



(s) defactorize-boolean

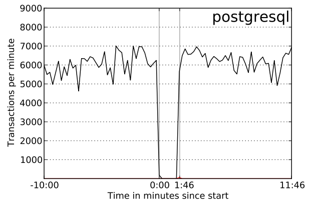


(t) defactorize

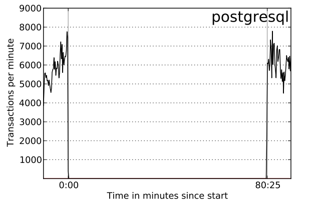


(u) factorize-boolean

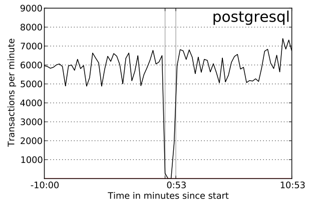
Figure 6: All PostgreSQL results, part 1 of 2.



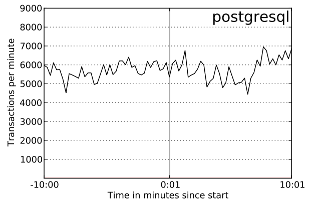
(a) factorize



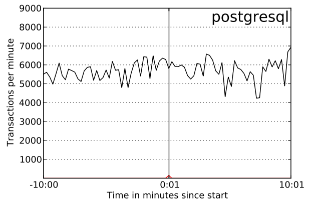
(b) join-relation



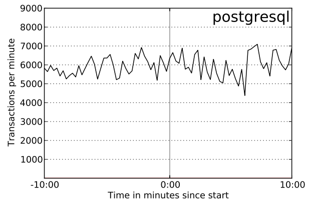
(c) precompute-aggregate



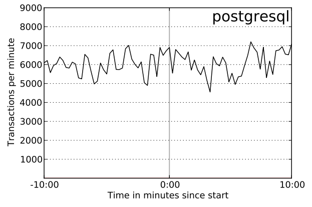
(d) remove-column-a



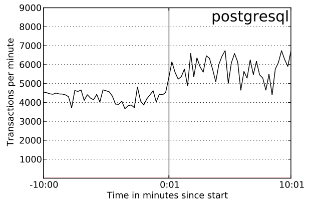
(e) remove-column-b



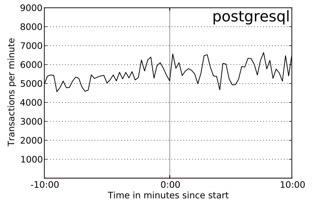
(f) remove-constraint-a



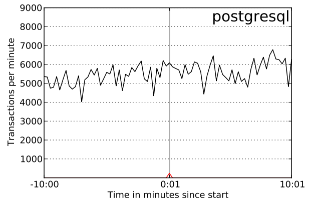
(g) remove-index-a-nc



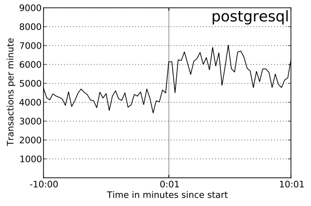
(h) remove-index-b-nc



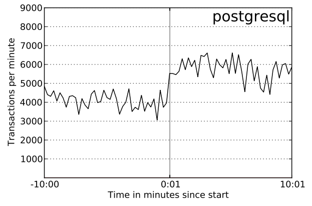
(i) remove-relation-a



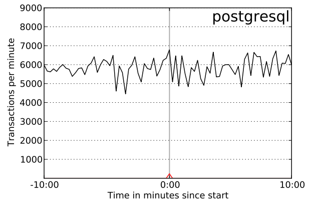
(j) remove-relation-b



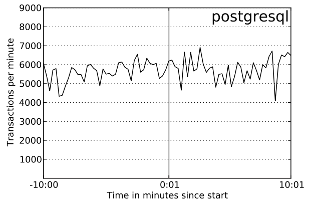
(k) remove-unique-a



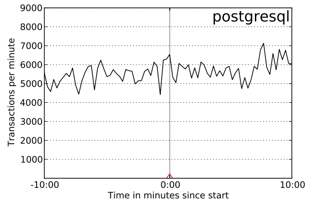
(l) remove-unique-b



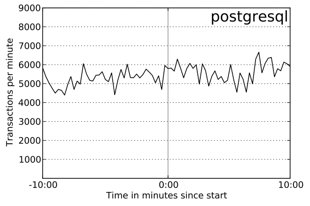
(m) rename-column-a



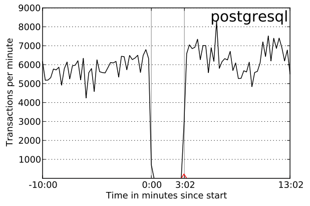
(n) rename-column-b



(o) rename-relation-a

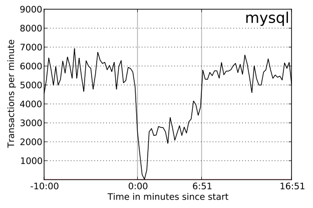


(p) rename-relation-b

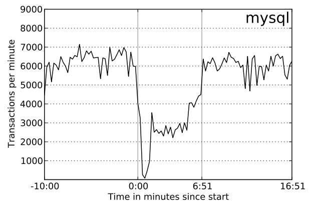


(q) split-relation

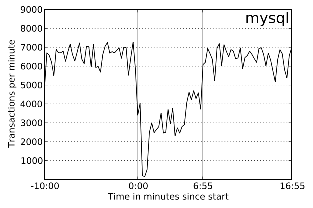
Figure 7: All PostgreSQL results, part 2 of 2.



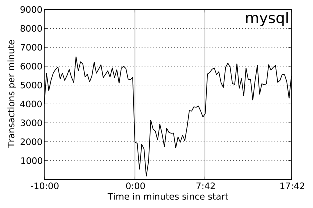
(a) add-column-a



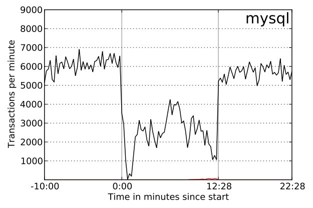
(b) add-column-b



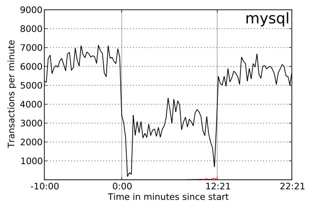
(c) add-column-default-a



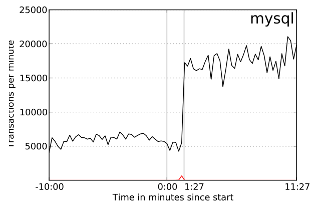
(d) add-column-default-b



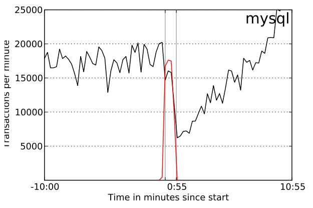
(e) add-column-derived-a



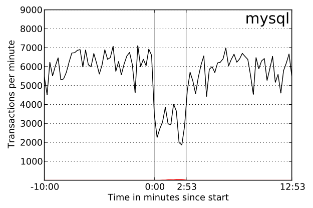
(f) add-column-derived-b



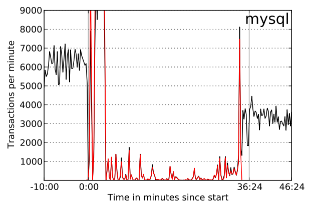
(g) change-cardinality-b



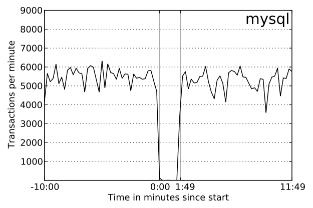
(h) change-cardinality-c



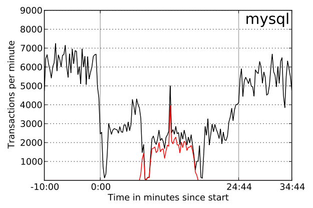
(i) change-data



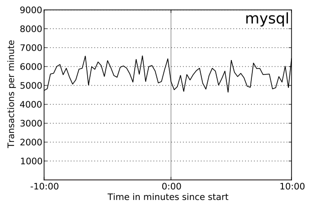
(j) change-primary-a



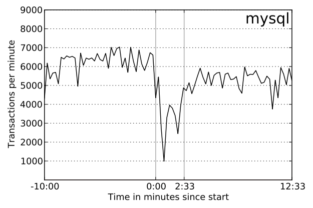
(k) change-type-a



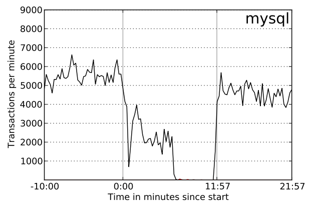
(l) change-type-b



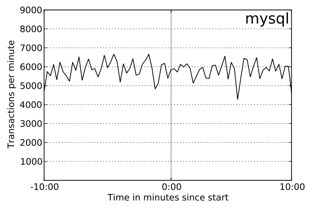
(m) control



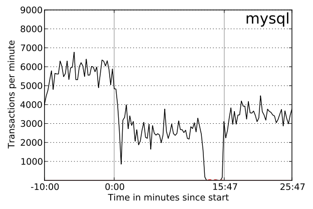
(n) create-index-a



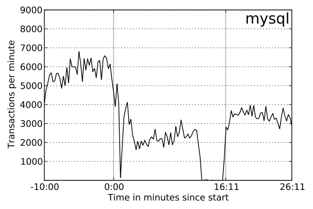
(o) create-index-b



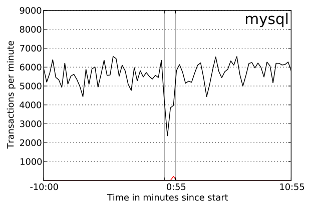
(p) create-relation



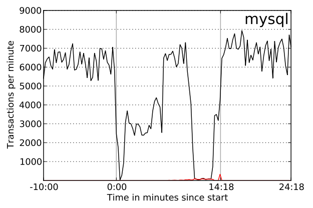
(q) create-unique-a



(r) create-unique-b

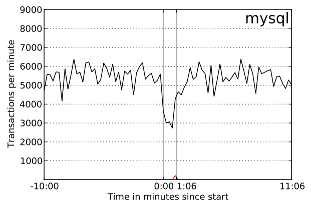


(s) defactorize-boolean

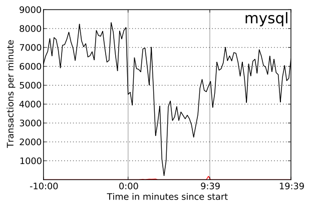


(t) defactorize

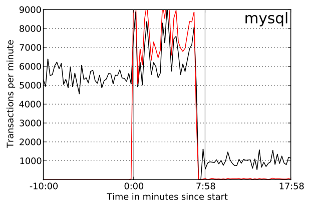
Figure 8: All MySQL results, part 1 of 2.



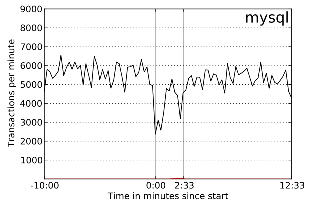
(a) factorize-boolean



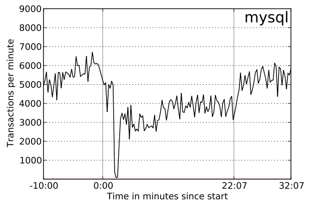
(b) factorize



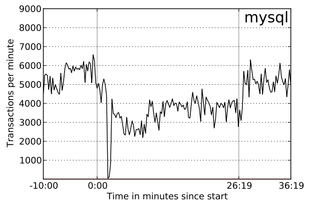
(c) join-relation



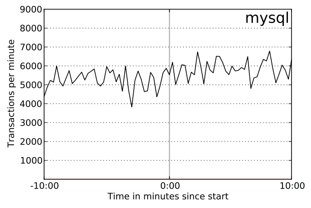
(d) precompute-aggregate



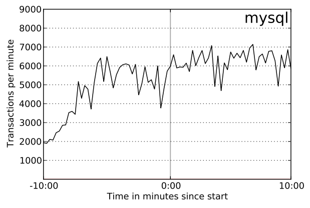
(e) remove-column-a



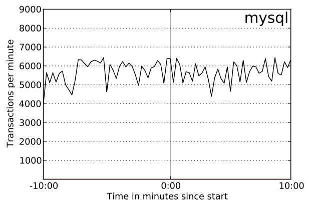
(f) remove-column-b



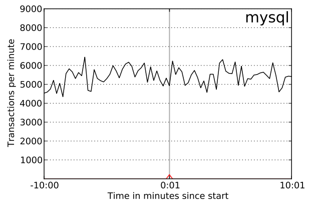
(g) remove-index-a



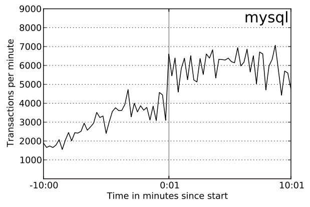
(h) remove-index-b



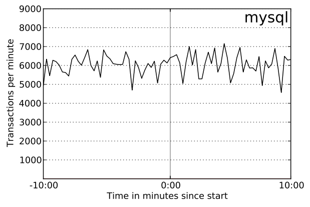
(i) remove-relation-a



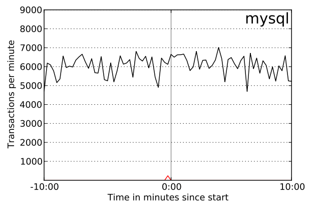
(j) remove-relation-b



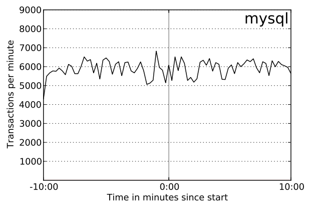
(k) remove-unique-a



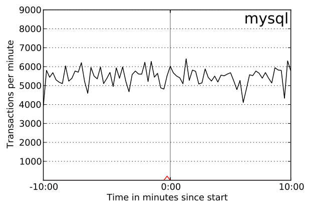
(l) remove-unique-b



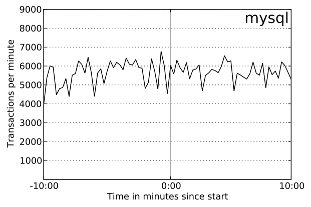
(m) rename-column-a



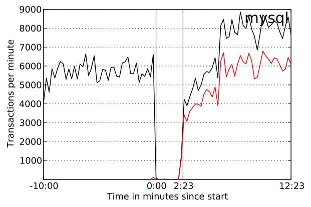
(n) rename-column-b



(o) rename-relation-a

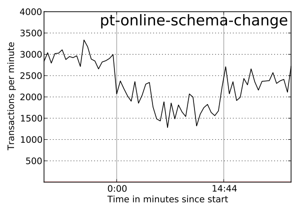


(p) rename-relation-b

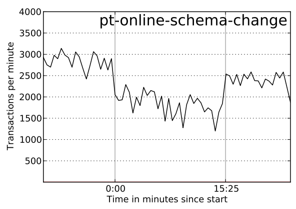


(q) split-relation

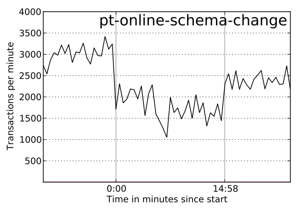
Figure 9: All MySQL results, part 2 of 2.



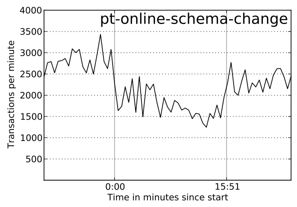
(a) add-column-a



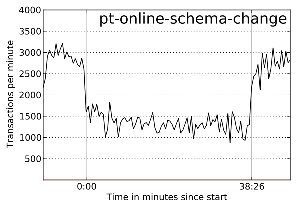
(b) add-column-default-a



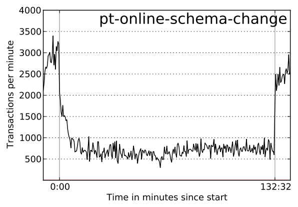
(c) change-type-a



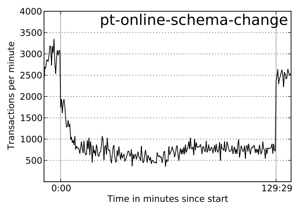
(d) create-index-a



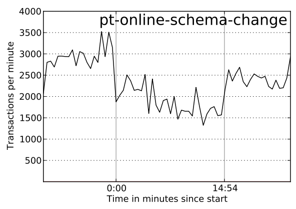
(e) create-index-b



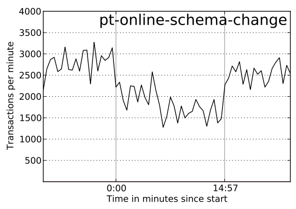
(f) create-unique-a



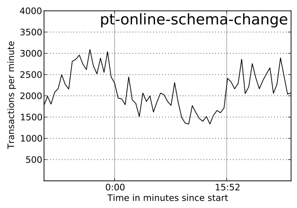
(g) create-unique-b



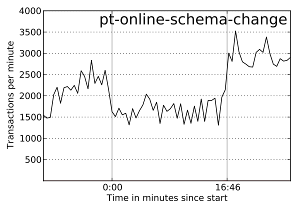
(h) remove-column-a



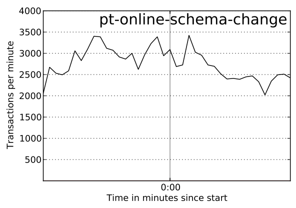
(i) remove-index-a



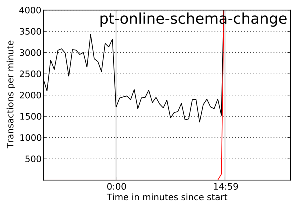
(j) remove-index-b



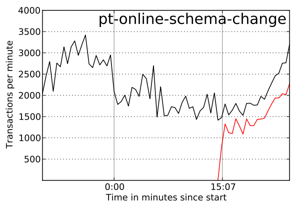
(k) remove-unique-a



(l) remove-unique-b



(m) rename-column-a



(n) rename-column-b

Figure 10: All pt-online-schema-change results.