# Towards Automated Online Schema Evolution

*Yu Zhu*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-218
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-218.html

December 14, 2017

# Towards Automated Online Schema Evolution

by

Yu Zhu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Eric Brewer, Chair
Professor Joseph Hellestein
Professor Joshua Blumenstock

Fall 2017

**Towards Automated Online Schema Evolution**

# Abstract

Towards Automated Online Schema Evolution

by

Yu Zhu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

Schema evolution studies the issue of moving a database from one version of its schema to a new updated schema. Traditionally, database administrators perform these tasks offline and they involve large amounts of manual labor and custom scripting. In today's world where databases power many 24/7 online services, schema evolution can no longer be an offline process. Furthermore, application requirements change much more rapidly today, causing more frequent changes in database schemas. Because of these trends, it is critical for database administrators to have automated tools to evolve database schemas in an online fashion that does not disrupt the foreground services.

This thesis attempts to explore ways an administrator might automate this process and provide some insight into building tools to help make this process easier, faster and more reliable. The thesis makes the following contributions. First, it provides a complete system implementation, Ratchet, that a database administrator can use to perform efficient schema evolution on supported platforms (PostgreSQL). The system uses various techniques such as improved fine-grained locking and a delayed-copy strategy to improve its schema evolution performance. Second, it analyzes the characteristics of schema evolution for a five-year period for Wikimedia, one of the most widely used websites. Third, using Ratchet, the thesis recreates five years of schema evolution automatically. Finally, the thesis provides a mechanism of rollback in schema evolution.

To my family

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I would like to thank my advisor Eric Brewer for all his help throughout my graduate career.

# Chapter 1

# Introduction

A database schema describes a structure of the database in a formal language. Like the real world it models, database schema can change over time as the needs of applications change. The database community has been interested in the problem of how to change schemas in a structured, scalable and safe manner for many years. They coined the term *schema evolution* to describe this process of constantly changing database schemas.

Schema evolution is a challenging problem for several reasons. First, schema changes vary greatly in complexity, so consequently the solution must be general enough to handle these different cases. In addition, large amounts of data often need to move from table to table as a result of a schema change, which causes stress on the system and can be error prone. Moreover, applications need to understand this schema change and they often need to be updated to adapt to the new schema.

In the past, online services often have maintenance windows, where they shut down to perform maintenance tasks such as schema changes. The benefit of this kind of approach is its safety and reliability, since no data change other than those related to the schema change can occur while the service is offline. When the database change is complete, the service comes back online with both the client and the database updated, so any new application associated with the new schema is in place. However, the obvious drawback of taking down a service in today's online world is simply unacceptable to many organizations and their customers. These services need to not only be online, but maintain a reasonable level of service quality while the schema evolution is taking place.

As online services become more prevalent, schema changes are also more frequent and more complex. For example, the software powering Wikipedia MediaWiki has undergone *several hundreds of schema changes* since its inception[17]. TripAdvisor regularly has *half a dozen schema changes per week*[27]. These frequent changes demand the schema upgrade process to be efficient and automated and in case of failure, easy to recover. We need effective tool support for database administrators to maintain the availability of the online services.

For this purpose, we created a tool designed to help with the process of online schema evolution, called Ratchet. In this work, we specifically focus on the problem of online schema evolution, where the administrators upgrade the schema while keeping both the database and the applications running. We will be using PostgreSQL as our database platform. While the majority of our tool is deployed outside of the database, we will be making a few important modifications to PostgreSQL. We have a number of objectives of the project:

- The database must be online and be sufficiently available to ensure a certain level of service quality for the applications using the database.
- The database must provide tools for the administrator to manage the upgrade process. In the unlikely event of failure, the administrator should be able to revert back to previous versions of the database schema:
- The upgrade process must be general enough to handle different schema upgrade scenarios from simple to complex multi-table joins and merges.
- The upgrade process for the application and the database need to be decoupled, as different organizations are likely managing the database and the applications.

With these four goals in mind, we will take a deeper look at the problem of schema evolution. We will examine the kind of work involved in upgrading a schema for a database.

## 1.1 Challenges in Schema Update

In this section, we look at a few reasons why schema update itself is challenging. One of the top reasons is the complexity of schema change. Administrators often change schemas for different reasons, and they change schema in different ways. There are broadly two reasons an administrator may want to change the database schema.

1. The schema changes because the underlying system it tries to model is constantly changing. For example, when a new feature is added to an application, the database that supports the application often needs a new column or even a new table to model the additional objects and their attributes.

| Employee ID | Salary | Dept id |
|---|---|---|
| 1 | 1000 | 1 |
| 2 | 2000 | 1 |
| 3 | 1000 | 2 |

Table 1.1: Employee Table

| Dept id | Dept Name |
|---|---|
| 1 | Sales |
| 2 | Engineering |
| 3 | Accounting |

Table 1.2: Department Table

| Employee ID | Salary | Dept id | Dept Name |
|---|---|---|---|
| 1 | 1000 | 1 | Sales |
| 2 | 2000 | 1 | Sales |
| 3 | 1000 | 2 | Engineering |

Table 1.3: Employee Table

2. Schema changes for performance. In this case, there is often no visible feature change. Performance is the primary reason for the schema change. For example, through profiling, we might have discovered that two tables are often joined together in most queries the system answers, and they are rarely updated independently. It would improve the performance of the queries if these two tables are represented as one table.

Not only do schema changes vary in the purpose they serve, they also differ in the complexity. Our solution needs to be general enough to address all kinds of complex schema changes. Next we will go through a number of examples of schema change, showing the work necessary to complete these schema changes, and why they might be challenging to complete online.

Let's take the following simple tables as an example. Table 1.1 and Table 1.2 are two tables describing two entities Employee and Department, and the relationship between them.

In the example case, the database administrator would like to add a column to the end of a table. The most obvious approach is to use `ALTER TABLE` SQL command to achieve this. For example, the following SQL command would add a column `address` to a table named `Employee`.

ALTER TABLE Employee ADD COLUMN address character varying(50);

Since this query only changes the metadata of the table, but does not touch the data in the table, changes can be done very quickly and efficiently. In fact, efficient support for this kind of operation is included in recent versions of DB2, and do not require locking the table. However, being able to update the table while altering a table structure in general remains an unsolved problem.

If the change causes data change in addition to metadata change, the size of the table will usually determine how long the change takes. Larger tables will cause more conflicts between updating each row as part of schema evolution and serving incoming requests from other sources such as web servers.

```
ALTER TABLE Employee ADD COLUMN address character varying(50)
DEFAULT 'Berkeley';
```

For example, in the above case, if we require the new field to be initialized to a default value, most databases would need to go through the table to change each row to reflect this requirement. The above SQL command executes this change. This operation can incur a large cost in read and write performance when the table is very large. Moreover, the modifications made to the tables can potentially acquire locks to resources and prevent other requests' timely access to the database. This causes performance degradation and unpredictable performance.

What we described above can be classified as an *in-place update strategy*, and it is most commonly used strategy today. In this strategy, the background task modifies the tables directly to achieve the desired resulting table structure. This strategy has the benefit of being efficient and is directly supported by simple SQL DML statements. However, the drawback is also significant. In the general case, it can cause interference with the foreground process that is also accessing the table. The two main reasons of intereference are locking and IO contention. Modification of each row in the table requires an exclusive lock on the table, which prevents others from using the table. Writing large amounts of data to the table also requires a lot of IO operation, which affects other concurrent IO operations.

Alternatively, in certain complex cases, it might be preferable to choose a *copy-based strategy.* In this case, the schema modification process does not change the original table. It simply creates a new copy with the desired table structure, and starts to copy data over to the new table. This approach also introduces background copying tasks. However, since it does not modify the original table, it does not need to exclusively lock any resources such as rows or indexes on that table. Instead of random access and writes on the original table, it writes out sequentially entire new tables. The drawback of this approach is its complexity. It often takes several steps and many SQL commands to achieve desired effect. Since it works above the database level, other applications are protected from seeing the intermediate results of this process. However, changes to the original table need to be propagated to the copy, otherwise inconsistencies will occur.

Next, we look at a case where the change in schema is necessary for performance reasons. In this case, the administrator may notice that the salary of an employee always changes with the department, and they are always accessed together anyways. She/he may choose to denormalize the tables, and convert the two tables into a single one, as shown in Table 1.3. This change is more complicated than adding a column, and can not be performed using a single command such as `ALTER`. The administrator normally needs to create the new table, and start issuing SQL commands to migrate data from the old table into the new tables. However, while the data migration is happening, the original table could be changed if it is not locked as we did in `ALTER TABLE`. So, the administrators will need to continue to monitor what has changed in the original tables and propagate them to the new table. Moreover, any application relying on these tables needs to be updated to query the correct table, depending on the stages of the evolution process. When the evolution process started, the application was reading the Employee and Department Table. By the end of the evolution process, it should be reading and writing the combined Employee Table. These steps can quickly grow in complexity and without the proper support, this process can be error prone.

In addition, because different groups are typically responsible for management of databases and applications, they may not be updated at the same time. Hence the database needs to support potentially an older version of the application operating on its data. This separation of update processes creates additional difficulties in evolving database schema online effectively.

## Challenges in Online Schema Evolution

There are several reasons why the background workload such as schema upgrade operation may interfere with foreground operation such as query answering. First, the database has internal locking mechanisms to ensure correct transaction semantics, and two different workloads could be accessing the same resource protected by the same lock, and therefore the schema upgrade operation could completely stop the foreground query answering. Second, database operations ultimately require hardware resources such as CPU and disk access. Workloads must share these resources. Contention on these resources is another reason why background operation may slow down query answering. The following sections dive into these two topics and explore why existing solutions may run into problems.

### Locking causes Blocking

Unlike contention on disk and CPU resources, which merely slows down a particular query, locking can completely block a query from executing. PostgreSQL, like many other databases, supports transactions and ACID semantics [6]. Internally, these features are implemented using a hierarchy of locks. In fact, most PostgreSQL commands will automatically acquire appropriate locks in order to ensure data consistency. In PostgreSQL, there are two main levels of locking, table-level locking and row-level locking. These locks, when acquired, will prevent other queries or commands from making progress.

Schema evolution steps move a large amount of data around, and therefore can cause locking conflicts with the foreground queries and stall them. In Section 2.5, we will go into more detail on locking in PostgreSQL and how we avoid blocking caused by locks.

### Contention in Hardware Resources

Besides the aforementioned locking, contention in hardware resources can also lead to performance degradation for the live traffic we are serving while we carry out background maintenance tasks such as schema evolution. Previous work has proposed many techniques to address this issue. Some of these techniques are client-based, some are server-based.

Client-based techniques include dividing up the workload and rate-limiting. If we can divide up one large background task into many small pieces, we can limit the rate we issue these requests to the database server. This allows database enough resources to process the more important foreground tasks in a timely manner. In schema upgrade processing, dividing up the work is not always easy, often it comes at the cost of increased complexity. Previous work has used this technique to create copies of a table (as mentioned in the previous section). If the schema upgrade operation involves multiple tables, this technique is less practical simply because of complexity. The advantage of these client-based approaches are the servers can remain unmodified.

Server-based techniques mainly center around priority-based scheduling. Because in our scenario, we have a clear preference for performing the foreground queries well, and we can tolerate slowdowns in the background schema changes, we can use this technique quite effectively. Unfortunately, PostgreSQL only supports CPU-based priority through the prioritize extension module. In schema upgrade, we are more likely bottlenecked by I/O performance. Ringer wrote a blog post that provides some workarounds in using linux ionice facility to simulate priority [22]. McWherter et al.

provided a detailed analysis of different workloads [16] and how they implemented lock-queue priority and CPU priority for PostgreSQL.

Our work does not focus on addressing hardware resource contentions. All of the techniques mentioned above can be applied to our system, and further improve the foreground query performance.

## 1.2 Previous work and Contribution

Before our attempt, there has been several efforts to address the issue of online upgrades of schema by others, both in industry and in academia.

Oracle 10g [19] supports online schema evolution by using an edition-based schema system. Each schema has a version number associated with it. Database clients can specify which edition the current connection is using. Additionally, the database provides a mechanism called cross-view triggers that can take changes in one version of the schema and apply them to tables in another version of the schema. There are several steps to a schema evolution process in this setting. First, the database administrator creates a new schema version and design the desired schema with the new version. In the meanwhile, the older version of the database continues to serve live traffic. Using cross-version triggers, the database can propagate any changes to the older version of the database to the newer version schema, and therefore keeping the newer version of the database consistent.

Facebook has used a similar copy-based strategy and applied it to MySQL database. They created a suite of php scripts and released it in a blog post[5]. It converts *ALTER TABLE* SQL commands into non-blocking operations by creating a copy of the table being changed. It uses database triggers to copy over the changes to the original table.

Compared to Oracle's and Facebook's solutions, our solution is different in the following ways. First, our solution uses a logical operation unit SMO that is higher level than *ALTER TABLE* commands. A single SMO often contains both how the table is changed and the data movement associated with that change. Because of this, SMOs serve as a convenient unit for rollback and commit operations. Second, it utilizes some very well supported database facilities in views and materialized views. This allows those DBMS that support incremental update to benefit even more from these new features. It also allows us to easily port our solution to other DBMS in the future because views are so widely supported. Third, it adds the ability to convert from materialized views into tables. This allows post-upgrade views to break away from the constraint of views and be truly independent from the tables on which they are based. Fourth, it breaks large, difficult-to-reverse schema upgrade operations into more manageable Schema Modification Operators. These operators can always rollback, and in most cases the rollback operation keeps all the data consistent as well. In the cases where we do not support rollback, we warn the administrator before committing the changes. For a detailed discussion of rollback and reversibility of SMOs, please see Section 4.4. Fifth, it combines simple DDL-based changes with more complex view-based changes and uses a common interface in schema operator. Consequently, we can add additional implementation strategies based on specific database support without changing the interface the administrator or the application uses.

## 1.3 Outline

The rest of the dissertation is organized as follows. In Chapter 2, we will introduce some database facilities such as views and materialized views that are fundamental to our approach to schema evolution. In Chapter 3, we introduce the overall architecture of Ratchet, explaining the major components of the system and how they interact with each other. In Chapter 4, we follow how a single schema modification operator travels through our system and how the database upgrades its schema in response to a modification statement. In Chapter 5, we evaluate our system both qualitatively and quantitatively. Qualitatively, we evaluate to see how it handles a diverse set of commands and real-life upgrade scenarios. Quantitatively, we generate different workloads to examine the quality of service of foreground queries in the presence of schema upgrade operations. We continue to examine other related work in Chapter 6, and we conclude in Chapter 7.

# Chapter 2

# Background

This section introduces some background on the different aspects of database systems we chose to build on when we implemented Ratchet. We will introduce them here, with the reasons why we chose to use them.

## 2.1 Materialized views and their roles in schema upgrades

The database "view", as a concept, has been around for a very long time[6, 26]. In Ingres, a view was defined as a virtual relation defined in terms of relations that physically exist. As implemented today, it often represents an object in a database that is defined by the result set of a stored query. Sometimes it is referred to as a virtual table, because it does not contain data but rather relies on the database's query engine to supply it with data on demand. When the system tries to answer a query involving one or more views, it first translates the query into a more complex query using the definition of the view.

A "materialized view", on the other hand, builds on this concept of view. Instead of having the database engine repeatedly executing the query to generate data for this virtual table, it stores the result of these queries, so they are pre-computed when needed. This can lead to more efficient query execution and avoid repeated work in the database.

However, because this materialization represents a different view of the data from the original table and it contains cached results, it is possible these results are outdated. For this reason, databases provide many different mechanisms to update this cached data to reflect the latest changes. This is called "view maintenance".

The simplest and the safest way to refresh a materialized view is to re-execute the stored query. This ensures that the resulting materialized view is consistent with the latest version of the data in the original tables. This method can be quite expensive in terms of computational cost though. Three factors ultimately determine how expensive this process is, 1) refresh frequency, 2)the size of the tables referenced in the query and 3) the complexity of the query. Refresh frequency is usually determined by the application requirements and tolerance for stale information. If the application can tolerate very old data being part of the query result, then refresh via re-execute is perfectly fine. The size of the tables and the complexity of the query determines the running time of the query.

Incremental maintenance of views has been studied extensively by the database community[12, 14]. Instead of re-executing the stored query to obtain the latest result for the materialized view, incremental maintenance seeks to algorithmically derive the changes that need to be applied to the last version of the materialized view, based on the changes applied to the tables referred in the query. This often has the desired effect of lower refresh time and lower burden on the database server that needs to maintain these materialized views. We do not go into details on the algorithms of incremental maintenance. However, we show that the same algorithms can be very useful in keeping the data in an updated schema consistent with data in the older schema.

Now that we have introduce views and materialized views, let us take a look how they relate to schema evolution. In schema evolution, there are two versions of the schema, the initial version V1, and the post-upgrade version V2. They both represent the same set of data. Each table in V2 can be expressed as the result of a query referencing tables in V1. Because of this, we can conveniently use all the facilities databases have implemented using views and any support for incremental update if they exist. Conceptually, it also makes it easier to reason about these schema upgrades, and should any error occur during the upgrade process, we simply go back to a previous version.

## 2.2 Database Triggers

A database trigger is a user defined function that runs whenever a certain event or trigger occurs. The most common types of event that can trigger such a function to run include addition, modification or deletion from a certain table. For example, when a new record is added to a table that has a trigger defined on the row addition event, this particular function will be called to perform some maintenance function for the record that was just inserted. Triggers are often used for audit and logging.

PostgreSQL started supporting database triggers in 1997 and we use them extensively in our schema update process.

## 2.3 PostgreSQL's Materialized View Refresh

In this section, we discuss how the DBMS we chose to implement our system handles materialized views and how they are refreshed.

In PostgresSQL, materialized views are implemented very much like tables. This gives us the additional ability to convert a materialized view that no longer needs refreshing into an actual table object. Details of this can be found in Chapter 4. When the user calls `REFRESH MATERIALIZED VIEW` command on a view object, it builds an entire temporary table from the query result. This process is very expensive and will take a long time if the resulting table is very large. In addition, this command acquires an exclusive lock on the entire materialized view while it is refreshing it, so it blocks any other queries that need access to the materialized view.

Recent versions of PostgreSQL have added an option to allow queries to read from the old versions of the materialized view while it is being refreshed. This is accomplished by creating a new copy of the table and atomically switching the two versions when it is done. However, the method which it employs to refresh the materialized view is still re-executing the query, not using any of the incremental view maintenance techniques. We are not using this concurrent update option in Ratchet, instead we use triggers to simulate incremental update behaviors for the schema operations that we support. Details of this can be found in Chapter 3.

## 2.4 Database Locking

ACID properties in database refer to a set of properties that a database transaction must have to ensure the correctness of database in the event of failure and errors. The acronym ACID stands for Atomicity, Consistency, Isolation and Durability. To implement these properties for transactions, databases often employ different kinds of locking mechanisms. Gray et al. described in his seminal paper[11] the importance of database locking and how to efficiently implement locks so that they do not impede concurrent processes' progress. One of the paper's central ideas is that different access patterns in the database require locks of different granularity to be efficient. Locking at record level allows concurrent transactions to proceed without interfering with each other. However, the frequent locking/unlocking operation may become the main overhead if the transactions involve many records. Hence, taking a lock at the table level might be more appropriate for those transactions.

Next section, we will discuss how locking mechanisms work in PostgreSQL, and how these locking mechanisms are used by Ratchet to achieve good concurrency for the schema evolution process.

## 2.5 Locking mechanism in PostgreSQL

The first mechanism we consider is table-level locking. There are eight locking modes at the table level. The following summary are from the PostgreSQL manual and details which SQL command would acquire which lock level for each table it references.

`ACCESS SHARE`

The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.

`ROW SHARE`

The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

`ROW EXCLUSIVE`

The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that modifies data in a table.

`SHARE UPDATE EXCLUSIVE`

Acquired by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, and some forms of ALTER TABLE.

`SHARE`

Acquired by CREATE INDEX (without CONCURRENTLY).

`SHARE ROW EXCLUSIVE`

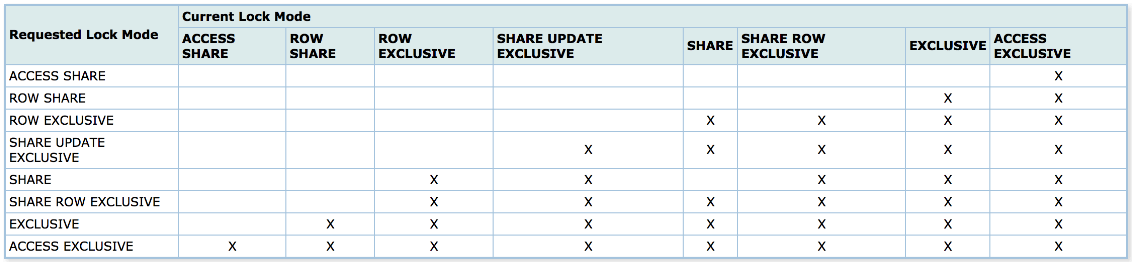
This lock mode is not automatically acquired by any PostgreSQLQL command.

`EXCLUSIVE`

This lock mode is not automatically acquired on tables by any PostgreSQLQL command.

`ACCESS EXCLUSIVE`

Acquired by the ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL commands. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.



| Requested Lock Mode | Current Lock Mode | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |
| ACCESS SHARE | | | | | | | | X |
| ROW SHARE | | | | | | | X | X |
| ROW EXCLUSIVE | | | | | X | X | X | X |
| SHARE UPDATE EXCLUSIVE | | | | X | X | X | X | X |
| SHARE | | | X | X | | X | X | X |
| SHARE ROW EXCLUSIVE | | | X | X | X | X | X | X |
| EXCLUSIVE | | X | X | X | X | X | X | X |
| ACCESS EXCLUSIVE | X | X | X | X | X | X | X | X |

Figure 2.1: Row Locking Modes

Table 2.1 shows how each of these locking mode may conflict with each other. Notice a number of commands often used to modify schema such as ALTER TABLE, DROP TABLE acquire the most strict of the locks `ACCESS EXCLUSIVE`. This means while these commands are executing, none of the other commands referencing the same table can make any progress. This is also why simply using DDL commands to change database schema would likely result in unacceptable performance for any foreground queries, because any queries referencing those tables would be blocked. Thus, we must find alternative ways to ensure the access to the original table is still allowed while the schema modification is going on.

The strategy we employ is to modify a copy of the table. All the locking would be done on this copy, while foreground queries continue to operate on the original table, which has no locks acquired on it. Once we are done with schema modification, we atomically switch the role of the two copies. Operating on a copy is one of the central ideas we build Ratchet around. This is discussed in detail in Chapter 4.

Whenever we have copies of data, how to keep the copies consistent becomes the most important task. Here we use a combination of two techniques, trigger-based updates and delayed updates to address this concern. These will be discussed in detail in Chapter 4.

# Chapter 3

# Architecture

In this chapter, we will explain our key design goals and requirements. Later, we will explain the high-level architecture of the proxy-based schema evolution system Ratchet.

## 3.1 Design Goals and Requirements

In this section, we will try to summarize several design goals and requirements of the system.

First, for this to be considered an online system, it must remain responsive when the schema evolution is under way. The system must continue to maintain a certain level of service quality. We will measure the system to ensure the background task does not unduly interfere with the foreground traffic, and when possible, gives priority to the serving of foreground requests.

Second, the system must be general enough to support all the schema changes that are possible in the offline scenario. Previous work in column-oriented databases has shown that certain schema changes, such as adding a column can be achieved much more efficiently by using a different way of storing data[15]. However, this does not work for the general problem of online schema change. More complex schema upgrades may require several statements to achieve a desired change. In these cases, it may be more efficient to simply create another table and copy data in as needed. The solution must be able to take advantage of both efficient *ALTER* implementations when possible and be general enough to support complex joins and merges.

Third, the system should support an undo operation in case of a failed upgrade. The system should able to return itself to a stable and self-consistent state. Updates during the transition period should be preserved in some form, which can be later extracted automatically or manually.

Fourth, the system should support independent upgrade processes for the database tables and the database clients. Upgrading the database should not stop the client from working completely. There should be a window where older versions of the client and the new version of the client can co-exist. They will operate on the database and produce results consistent with the schema upgrade process. In cases where this is not possible, the older client should retire and the new version of the client can take over without shutting the database down or causing disruption in the database services.

Before we describe each system component in detail, and how they fulfill these requirements, we first introduce the notion of a schema modification operator, as it is a key concept used throughout our system.

## 3.2 Schema Modification Operator

Schema Modification Operators (SMOs) are a set of operations originally proposed by Curino et al. In this work, they proposed a set of operators that describe a possible set of schema changes. In their work, they also validated this set of SMOs by describing a list of historical changes to MediaWiki using SMOs only.

Because we want to support rolling back of schema changes, we want to break down the schema changes into small and manageable components. This aspect of SMOs are related to the work of Sagas proposed by Garcia-Molina and Salem [10]. Sagas were used to address long-running transactions holding on to database resources for too long. Instead they are a series of short-running transactions that can be interleaved with other database transactions. However, if this series of transactions encounters an error in the middle of execution, the database executes additional *compensating* transactions to logically bring the database to a consistent state.

Similarly, we use and implement SMOs in this fashion. A transformation is composed of several SMOs, and indeed other changes and transactions can happen between these SMOs. However, with the addition of rollback, what we are really doing is executing compensating SMOs to negate the effect earlier partial execution of SMOs in the schema evolution. The majority of SMOs support reverse operations that can be used as undo operations. Hence, we use SMOs in our system as the unit of schema change.

Here, we give some background on these operators. A Schema Modification Operator is defined as a function that receives as input a relational schema and the underlying database and produces as output a (modified) version of the input schema and a migrated version of the database.

Table 3.1 lists a complete set of operations that we support. These operators vary in complexity, information preservation, uniqueness of the inverse, and redundancy. Here, we focus on how these operators may be implemented in an online fashion. One feature that greatly expands the expressibility of SMOs is custom functions. We see examples of that in the `ADD COLUMN` operator. In Section 5.2, we use some examples from MediaWiki to explain why custom functions are important and the caveat in using them.

## 3.3 Overall Architecture

There are three main components to Ratchet: the client side, the proxy itself and the various modifications we make on the server side. These components could be co-located on the same machines or distributed across several machines, depending on the deployment scenario.

On the client side, we provide the database clients with a JDBC-compatible driver. It is a type III JDBC driver. This means it will connect to another JDBC driver that interacts directly with the database server. Alternatively, we could have modified a JDBC driver for a particular database to fit out needs, which likely would have resulted in better performance. Instead, we chose this approach because we do want to keep the option to support a variety of different database servers.

In this approach, clients can interact with the database using our JDBC driver. It provides all the familiar interfaces such as resultset, query interface, and the ability to iterate through the resultset. However, in this case, the client does not directly connect to the database, but rather it connects to the proxy that interposes between the client and the server.

The proxy acts as the orchestrator of the schema upgrade process. It forwards the requests from the client to the database, and more importantly coordinates schema change operations between different clients. During the schema change operation, the proxy may rewrite certain queries to handle updates.

Finally, the database server, in this case PostgreSQL, is modified to support features such as an updatable views and converting views to tables.

Figure 3.1 is an architectural diagram of the system. Applications connect through a client-access library, which is a modified JDBC driver, to the proxy, which then connects to the database. When an administrator would like to make a schema change, it issues a schema change command that is passed through to the proxy. The proxy determines the exact SQL commands that should be executed on behalf

| Schema Modfication Operator | Description |
|---|---|
| CREATE TABLE R(a,b,c) | Introduces a new, empty table R into the database |
| DROP TABLE R | Removes an existing table R |
| RENAME TABLE R INTO T | Changes a table's name from R to T |
| COPY TABLE R INTO T | Creates a duplicate of table R as T |
| MERGE TABLE R, S INTO T | Takes two tables with the same schema R and S and creates a table T storing their union |
| PARTITION TABLE R INTO S WITH cond, T | Takes source table R and distribute into two tables S and T according to specified condition |
| DECOMPOSE TABLE R INTO S(a,b), T(a,c) | Creates two tables S and T with a subset of columns from the original table R |
| JOIN TABLE R,S INTO T WHERE cond | Joins two tables R and S according to a specified condition and store it in T |
| ADD COLUMN d [AS const— func(a, b, c)] INTO R | Introduces a new column into the specified table. The new column is filled with values generated by a user-defined constant or function |
| DROP COLUMN c FROM R | Removes an existing column c from table R |
| RENAME COLUMN b IN R TO d | Changes the name of a column in table R from b to d. |

Table 3.1: Schema Modification Operator Description

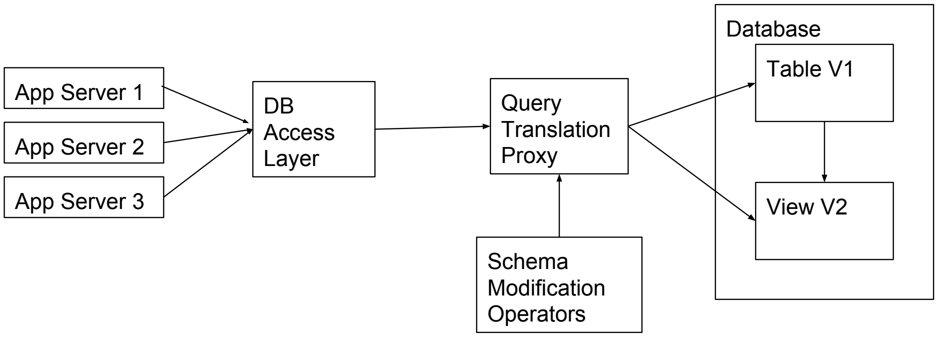


Figure 3.1: System Architecture

of the client. The server side is modified to support certain view-based operations that are not available in standard PostgreSQL.

## 3.4 Proxy

At the heart of Ratchet is a coordinator that mediates accesses to the database, and guides the schema upgrade process. It sits between clients and the database, and functions as an RPC service for the client. It uses the GRPC protocol[13] and protocol buffers to service these RPCs. Functionally, this component is responsible for coordinating the schema upgrade process, and to ensure the right clients reach the right tables in period of transition.

| Schema Modfication Operator | Schema Upgrade Strategy |
|---|---|
| CREATE TABLE R(a,b,c) | Direct |
| DROP TABLE R | Direct |
| RENAME TABLE R INTO T | Direct |
| COPY TABLE R INTO T | Direct |
| MERGE TABLE R, S INTO T | Copy-based |
| PARTITION TABLE R INTO S WITH cond, T | Copy-based |
| DECOMPOSE TABLE R INTO S(a,b), T(a,c) | Copy-based |
| JOIN TABLE R,S INTO T WHERE cond | Copy-based |
| ADD COLUMN d [AS const— func(a, b, c)] INTO R | Direct |
| DROP COLUMN c FROM R | Direct |
| RENAME COLUMN b IN R TO d | Direct |

Table 3.2: Schema Modification Operator Strategy

```
rpc getConn (ConnRequest) returns (ConnReply) {}
rpc execQuery (QueryRequest) returns (QueryReply) {}
rpc resultSetLoad (RSRequest) returns (RSReply) {}
rpc readRow(RowRequest) returns (RowReply) {}
rpc execSMO(SMORequest) returns (SMOReply) {}
rpc execUpdate(UpdateRequest) returns (UpdateReply) {}
rpc execSMOString(SMOStringRequest) returns (SMOReply) {}
```

Listing 3.1: RPC Inteface for OSEProxy

List 3.1 is a listing of the RPC interfaces provided by this component. *getConn* is responsible for obtaining a connection to the database. *execQuery*, *resultSetLoad*, *readRow* are used to execute queries and navigate the result set. *execSMO* and *execSMOString* are the key interfaces to send parsed and string-based SMO commands.

Once the SMO is issued by the client and reaches the proxy, the proxy interprets the request and sends appropriate commands to the database to implement the schema change. These changes could be a simple *ALTER* command to update in-place or a series of commands to create a new table and copy data over (i.e. copy-based strategy) to avoid locking the original data table. Table 3.2 shows the strategy we currently employ for each of the SMOs listed. Of course, depending on the underlying database, this can be adapted if there are more efficient implementation for certain operations.

## 3.5 Client Access Library

The Client Access Library is a relatively thin JDBC-compatible library that allows clients to connect to the database. The client has the option to specify which proxy to connect through. Under normal operation, the client behaves exactly like a JDBC client. It can issue SQL queries, and explore the returned result set using standard SQL commands. The JDBC functionalities are implemented using the RPC calls listed in 3.1.

Additionally, given the right credentials, it has the capability of issuing SMO calls to the proxy. It can issue the SMO in one of two formats, as a string or as a list of keywords that are already parsed. The exact steps of transition will be covered in later chapters.

## 3.6 Server Side Modification

### Ideal Database Server

Before we talk about the server-side modifications, we first discuss what an ideal database server would support. We believe support for the following features would make online schema update easy to implement in a safe and efficient fashion. Later, we will evaluate the current database systems and bridge the gap between the ideal support and what is currently available.

As mentioned above, we use both in-place update and copy-based schema updates. In this section, we specifically discuss copy-based schema updates, as they are the more complex approach and require more support from the underlying database beyond standard SQL support. In-place updates are well supported by standard SQL, and only serve as an optimization. So we can always fall back on copy-based schema updates if the database does not support in-place updates well.

- The underlying database must certainly support the creation of views and in particular materialized views.

  Unlike previous copy-based strategies where brand new tables are used, and a lot of manual copying is required between the tables, our copy-based strategy relies heavily on the use of views. There are two reasons for this. First, most databases already support views well. Second, a view of a table is essentially a different way of representing the same underlying data, which is exactly what we need to do in the process of upgrading a database schema. Hence, the database we select should have efficient support for views and materialized views.

- Materialized views should support incremental update.

  Incremental update is a well-studied topic in Database Management. It allows views to update themselves without repopulating the entire view, but rather only change those parts that are affected by the updates in the original table. In our case, for majority of our SMO operations, having this feature allows us to efficiently propagate changes from the original table to a new table without reading through the entire original table again. During our search for a database platform, we found that most databases lack this feature, but we believe it is an important feature to add, especially if it facilitates efficient online schema upgrade. The next section talks about a few changes we made to address the lack of this feature in PostgreSQL.

- A Materialized view should be convertable into a normal table, and support all the operations a table would normally support.

  Materialized Views are essentially tables with a source table that dictates how the view should be updated when the source table is updated. Because we use a materialized view to create our new table, when we decide to commit the change of schema, ideally we would like to operate on that materialized view as a normal table from that point on. Most systems we surveyed have similar implementations for materialized views and tables. In the case of PostgreSQL, we had to modify around a hundred lines of code to be able to convert a materialized view into a table.

## Modifying PostgreSQL

We chose PostgreSQL as the database server to support, for a number of reasons. First, it is open source and easy to modify/enhance to fit our needs. During our search, it was clear to us that none of the existing database systems supported all the criteria that we needed. So we needed to ensure whatever we chose was easily modifiable and adaptable to fit our needs. Second, it has some support for views, materialized views and view updates, all of which are important in our implementation of the copy-based strategy. However, we needed to make a few modifications to PostgreSQL 9.6 source code in order to fully support the features we need.

First, we added a command *CONVERT* in the PostgreSQL client console to convert a materialized view into a regular table. This was done so that we can use the view maintenance facility of the PostgreSQL database to propagate changes from the original table to the view that we create. After we commit the change, we would like to operate on a regular table instead of a materialized view, so this command is created to achieve that.

To achieve this, we changed the parser to recognize the *CONVERT* keyword and correctly identify the materialized view that needs to be converted. We also updated the database system catalog to change the appropriate row to have the class id matching that of a table instead of a materialized view. In addition, we modified the dependency graph in the system so that the materialized view no longer depends on the original tables. This allows independent modification to the original table and the newly converted table. This is necessary because in many instances, we choose to drop the original table after the schema update is completed. Having this dependency link would prevent PostgreSQL from dropping the original table.

Second, because PostgreSQL does not support incremental update of materialized views, we chose to simulate this functionality using triggers and updatable views.

In normal operations, when a user calls to refresh a materialized view, PostgreSQL takes the original table and completely regenerates the view when the refresh command is called. This is the only way to update a materialized view from a user's point of view. We instead allow users to directly manipulate the data that is already in a materialized view. This is done by changing a boolean flag to indicate to the database that user is in a context where updates of materialized view are allowed.

We enable triggers on materialized views in a similar fashion. PostgreSQL normally disables triggers on materialized views because users normally do not have the ability to update the materialized views. Since we enabled users to update materialized views, it makes sense to allow them to create triggers for materialized views as well.

In total, the modification to PostgreSQL was about two hundred lines of code. It allowed us to simulate incremental update of materialized views and use them in our schema update operation, specifically for copy-based strategies.

# Chapter 4

# Life of a Schema Evolution Operation

Last chapter, we gave an overall view of the different components of the system. In this chapter, we will follow the life cycle of a schema evolution operator and describe in detail how it is handled by each part of the system. Specifically, we will discuss parsing, view and trigger creation, incremental update, rolling back of operators, and how the updates are committed in the system.

## 4.1 SMO Class Structure

In Ratchet, administrators input schema change operators through a command line console. These commands are parsed and converted into our internal representations, which we discuss in this section. We used Antlr parser generator[2] to define the grammar for these input commands. For the detailed listing of the grammar, please see the attached grammar description in Appendix A. We create an SMO object for each command. This object encapsulates a command's parameters and its associated execution state, and is passed from the client to the proxy via RPC calls.

Figure 4.1 shows a class inheritance diagram of the related classes. All SMO objects implement the lSMOCommand interface, but can be subclasses of AbstractSimpleCommand or AbstractComplexCommand. SMOCommand defines the basic steps for an SMO to execute, including, connect, execute, rollback and whether or not it is reversible. Subclasses of AbstractSimpleCommand can be executed with a few lines of SQL command that do not require a large amount of execution time, and do not block other queries from accessing the database. Because of this, we can issue queries to modify the table in-place rather than making a copy and modifying the copy.

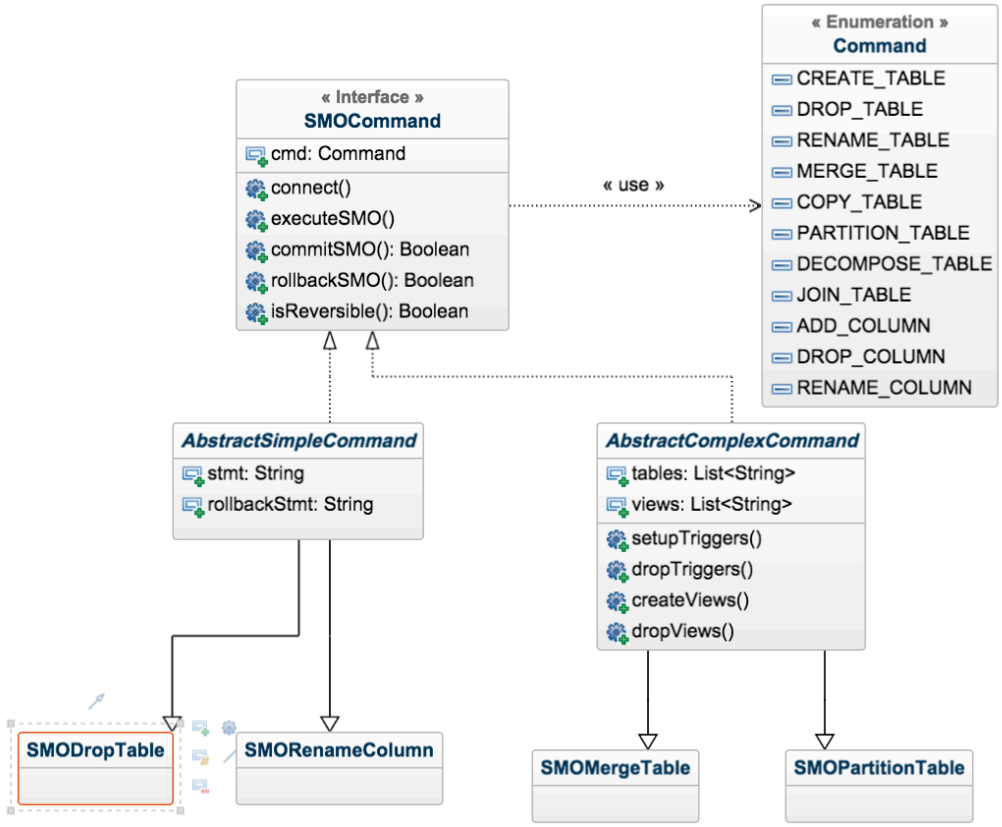


Figure 4.1: Class Diagram

These simple SMOs are discussed in Section 4.2. Figure 4.1 lists DropTable and RenameTable as examples of this.

The abstract class `AbstractComplexCommand` includes skeleton code that creates views and triggers to propagate data to keep old and new tables consistent, and is discussed in more detail in Section 4.3. Examples of its subclasses include `SMOMergeTable` and `SMOPartitionTable`.

Most commands are similar to SQL Data Definition Language (DDL) commands, and it is fairly easy to convert between them. However, DDL statements only change the schema and do not change or move the data in the tables. A single schema change operator can change both at the same time. This creates a link between the schema change and the intended use of the change shown through data movement.

This is important because SMOs create a link between data movement with schema change, which helps with rolling back changes. For example, we have a

COPY_COLUMN operator that copies a column from one table to another table. Using DDLs, this would be accomplished by an ALTER COLUMN statement and an UPDATE command. During rollback, these two statements are considered independent, and individually an UPDATE statement is difficult to undo because of its expressiveness. Coupled with the insertion of a column, it becomes rather easy to undo, we simply remove the new column that was added. A difficult data movement statement is now easier to rollback because it is coupled with a simpler schema change statement.

## 4.2 Life of a Simple SMO

Roughly half of the schema modification operators fall into this category of simple SMOs. Many of these schema changes, such as RENAME COLUMN and RENAME TABLE only modify table metadata, so they are rather quick to execute, even if they acquire some small locks in the process. Here we discuss a few SMOs that do modify data, but are implemented using Simple SMOs. Note these choices assume the underlying database uses Multi-Version Concurrency Control(MVCC), which is what PostgreSQL uses by default. COPY_TABLE creates an additional copy of a table, and can take some time to complete. However, other queries can still access the original table while the copying is taking place. Hence, we can still consider it a simple SMO. ADD_COLUMN also changes the data in the original table. However, as long as it does not need to initialize the column, the process is quite efficient and does not take very long. This is because PostgreSQL stores a null bitmap for each row of the table to record which of the column is null. Any newly added column is going to be defaulted to null in this null bitmap. Thus adding a nullable column defaulted to null is very efficient in PostgreSQL and does not require changing the tuples. For this reason, we use simple SMO to implement ADD_COLUMN. The benefit of using SMO is that we can change the implementation strategy depending on the underlying database implementation. For a different platform, we might have chosen a different strategy.

For these simple SMOs, we extend the class SMOSimpleCommand, which itself is an implementation of SMOCommand Interface, and override the implementation of methods such as commitSMO, executeSMO and rollbackSMO. This class structure allows us to have some shared implementation between these commands such as maintaining the database connection.

We use COPY_TABLE as an example to explain the operations in more detail. When executeSMO is called, it issues SQL commands such as Create Table to create a new table and Select to copy old data from the old table to the newly created table. rollbackSMO reverses these changes, so it would issue Drop Table command to drop the newly created table. commitSMO means the user is satisfied with the change and

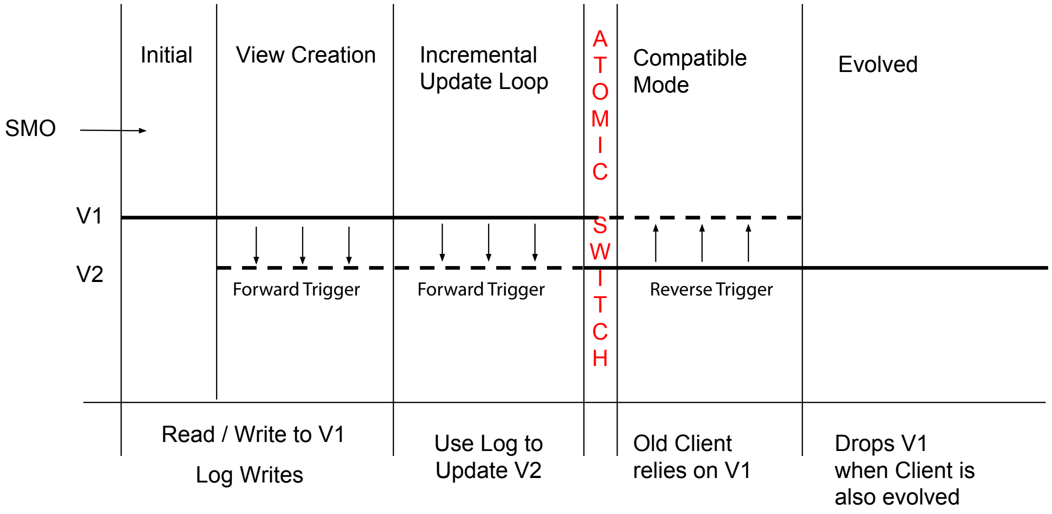


Figure 4.2: Timeline for Complex SMOs

will not rollback. This call is more meaningful and will be explained in more detail in Section 4.3.

## 4.3 Life of a Complex SMO

Similar to the aforementioned simple SMO, complex SMOs also implement the same set of interfaces, such as `commitSMO`, `executeSMO` and `rollbackSMO`. However, the internals are quite different and much more complex. Complex SMOs often create new tables that are views of the original tables, and operate on these views. In addition, complex SMOs use triggers to make sure any changes to the original tables are propagated to the views we created. In some cases, reverse triggers are also created to propagate changes from views back to the original tables.

To support these common operations, we created an abstract base class from which all complex SMOs extend. This base class provides operations such as setting up triggers, tearing down triggers, and setting up and removing of views. The entire process of copy-based schema change can be divided into four stages. Figure 4.2 shows the complete timeline for a complex SMO. The process starts by creating views and triggers, and continues to receive more updates from the foreground operations, while migrating data in the background. After several iterations of data migration, the process undergoes an atomic switch where the two copies of the data switch their roles. The DBMS continues to maintain the consistency of the copies by using reverse triggers. The following sections walk through these stages in more detail.

## View and Trigger Creation

Complex SMOs all produce a table or multiple tables as an end result. In our system, we use views to simulate the tables and convert the views into tables after we are satisfied with the result. The first stage is view and trigger creation. Using our running example, the MERGE operator has the following syntax:

```
MERGE_TABLE ID COMMA ID INTO ID
```

It merges two tables with the same schema into one table. Thus, the very first step is to create the view that will hold the result of the query. The query depends on the SMO. Here, it is a select query of the following form:

```
CREATE MATERIALIZED VIEW table3 AS
select *, 1 as srctable FROM table1 UNION ALL select *, 2 as srctable FROM table2
```

Here, we are essentially creating a second view of the same data that was originally stored in table 1 and table 2. Note that we generated an additional column to record whether the row came from table 1 or table 2 originally. This column will be useful when we propagate changes between the source tables and the view. We are using the copy and modify idea to avoid lock contention. As mentioned before, we need some mechanism to keep the data consistent between the original two tables and the new table we create. Normally we would be taking advantage of any incremental update features the database may support, since we are using a materialized view. However, since PostgreSQL does not support incremental updates of materialized views, we simulate incremental update by using triggers. When the original table is modified while the materialized view is being created, we need to propagate these changes to the merged materialized view. The ability to be able to insert into and update a materialized view becomes critical here, because without this capability we would not be able to propagate these changes individually, and would have to rely on the database to update the materialized view automatically.

There are two types of triggers, forward triggers and reverse triggers. As the name suggests, forward triggers propagate information from source tables to destination tables. Reverse triggers propagate information from destination tables back to the source tables. Some operations do not have reverse triggers; we call those irreversible operations. These will be covered in more detail in Section 4.4. Back to our example, we have three triggers in total. One forward trigger on each of the source tables, and one reverse trigger on the newly created materialized view. For each insertion on the source table, the forward trigger inserts the same row into the merged table. We also record which table caused this change in the srctable field. For each deletion on the source table, the forward trigger deletes the same row from the merged table if the srctable field also matches where the deletion is happening. For each update on the source table, the forward trigger updates all rows matching the criteria in the merged materialized view. Similarly, reverse trigger propagates changes back to the source tables. Table 4.1 details the propagation rules of all the forward triggers used to implement complex SMOs. Table 4.2 details the propagation rules of all the reverse triggers used to implement complex SMOs.

There are two methods we use to propagate changes from the original tables to the new materialized views, direct trigger propagation and indirect propagation.

### Direct Trigger Propagation

The most straightforward solution would be to install triggers on the original table so that for each insert, update, delete issued to the original tables, we will immediately modify the materialized view accordingly to reflect those changes.

However, this would not work while the database system is building the materialized view, because building of the materialized view acquires an exclusive lock on the entire materialized view. All these trigger actions would have to wait for the building of the materialized view to complete in order to proceed. This essentially stalls any updates to the original table, if these direct triggers were used.

We use direct trigger propagation in Ratchet once the materialized view has completed its initialization.

### Indirect Trigger Propagation

As its name suggests, indirect trigger propagation uses an intermediate table to store any trigger actions, and later apply them to the materialized view. In this case, we use a table called `history_log` to store this information. Its schema is detailed in Table 4.3.

Each history entry contains a trigger id, the source of the trigger stored as schema name and table name, the type of the trigger stored as insert, update or delete trigger, the old and the new data stored as hstore type in PostgreSQL, and the iteration, which records the number of times we have processed the history log.

HStore is a new datatype introduced in PostgreSQL 9.0 that is very similar to a key-value store. Here we use it to store a row of data in the database, with the

Table 4.1: Propagation Rules for Forward Triggers

| SMO | From base table to views |
|---|---|
| MERGE TABLE (R,S) INTO T | Assumes no duplicated entries in R and S<br>Insert R ⇒ Insert T,<br>Delete R ⇒ Delete T if originated from R<br>Update R⇒ Update T if originated from R |
| PARTITION TABLE R INTO S WITH cond, T | Insert R⇒ if cond, Insert S, else Insert T<br>Update R⇒ if cond, Update S, else Update T<br>Delete R⇒if cond, Delete S, else Delete T |
| DECOMPOSE TABLE R INTO S(a,b), T(a,c) | Insert R⇒Insert S(a,b) and Insert T(a,c)<br>Update R⇒ Update S and Update T<br>Delete R⇒Delete S and Delete T |
| JOIN TABLE R,S INTO T WHERE cond | Insert R⇒Insert T Record Join S where cond<br>Update R⇒Update T Record Join S where cond<br>Delete R⇒Delete T |

Table 4.2: Propagation Rules for Reverse Triggers

| SMO | From views to base tables |
|---|---|
| MERGE TABLE (R,S) INTO T | Insert T⇒Insert R, R<br>Update T⇒Update R and Update S<br>Delete T⇒Delete R and Delete S |
| PARTITION TABLE R INTO S WITH cond, T | Insert S⇒Insert R<br>Update S⇒Update R if cond is met<br>Delete S⇒Delete R if cond is met |
| DECOMPOSE TABLE R INTO S(a,b), T(a,c) | Insert S⇒Insert R with default values<br>Update S⇒Update R with default values<br>Delete S⇒Update R with null values |
| JOIN TABLE R,S INTO T WHERE cond | For joins where the condition is R.colA = S.colB<br>Insert T⇒Insert R if not exist and Insert S if not exist<br>Update T⇒if changed columns in R, then insert into R if not exist.<br>if changed columns in S, insert into S if not exist.<br>Delete entries from R, S if this is the last join entry.<br>Delete T ⇒ delete from R,S if this is the last join entry<br>For detailed discussion see Section 4.4 |

Table 4.3: History Log Schema

| Column | Type | Modifiers |
|---|---|---|
| op_id | integer | not null, serial |
| schema_name | text | not null |
| table_name | text | not null |
| action | text | not null |
| old_data | hstore | |
| new_data | hstore | |
| iteration | integer | |

column heading as the key and the actual data as the value. For insertion, only the new_data field is populated. For deletion, only the old_data field is populated. For update, both new_data and old_data fields are present.

As PostgreSQL builds the materialized view, any modification to the original table is stored in this history table, with an iteration number of 1. After the materialized view finishes building, we take these history entries and convert them into modifications to the materialized view. While this is happening, it is possible that there are more modifications to the original table, we again record these in the history table as iteration 2. This process continues until we ensure materialized view and the original table represent the same data, with no pending history entries to replay. If a schema update happens when the database is under heavy load, the materialized view may not be able to catch up with the update rate of the original table. Under this kind of conditions, the administrator may choose to throttle the update rate to the table. However, it is normally not advisable to perform schema update during peak hours.

This process of updating the materialized view is more complex than the direct trigger propagation approach, but it has the additional benefit that we do not need to update the materialized view while the system is holding an exclusive lock on the view. Hence, we never block any updates to the original table because of these triggers. In Section 5.3, we show that this is necessary to make progress while the query mix load concentrates on the tables that the materialized view depends on.

## Incremental Update

After the triggers are created and attached to the views, and while the materialization of the view is taking place, other applications can continue to modify the original table. However, because we have installed forward triggers on these tables, those changes are going to get propagated to views.

In this mode of operation, the proxy considers the original tables the primary copies to which the reads and writes will go. The database itself propagates these changes from the primary copies to the materialized views via triggers. As illustrated in Figure 4.2, consider a transformation of V1 to V2, at this stage of the process, V1 is the primary copy, and we forward information to V2 via triggers. This will change after the next step in the process.

## Atomic Switch

This is not a stage of operation, but rather a critical point in ensuring correctness of the system. Before this point, the primary copy of the data resides in the original tables. In case of sudden disruption of the evolution process, we revert back to the primary copy. However, after this point, the materialized views are considered to be the primary copy and the older tables are there for undo purposes and backwards compatibility.

This is also the point where the administrators can safely upgrade the client versions from V1 to V2. V2 clients can now safely update the materialized view and have the information propagated back to the original table via triggers. This is so that we have the ability to roll back the upgrade process should something unexpected happen.

As illustrated in Figure 4.2, the proxy ensures that the atomic switch happen for all clients accessing the relevant tables at the same time. This is especially important if the administrator deploys old and new versions of the clients simultaneously. This often occurs when a group of application servers or clients start a rolling upgrade process. In this case, we need a way to resolve any conflicts that may arise from the new version and older version of the application. A reasonable default policy lets the proxy designate the new version as the primary and the older version as a read-only copy after the atomic switch point. Our choice of proxy-based solution allows for such policies to be implemented easily.

## Compatibility Mode

In this stage, the system continues to process incoming queries under the assumption that both old and new versions of the client are both deployed and accessing the database. Clients will start to change the newly created views, and as a result of propagation, the older tables will be kept consistent with the new views. We call this the compatibility mode.

Table 4.4: Reversibility of Complex SMOs

| SMO | Reversibility |
|---|---|
| Merge | Yes |
| Partition | Yes |
| Decompose | Yes |
| Join | No |

During this period, the administrator can evaluate the effect of the changes to the overall system, and conduct any performance or load tests. The administrator also has the option to revert the change of schema if necessary. We will discuss rolling back of schema upgrades in detail in Section 4.4.

### Commit Evolution

This is the last stage of a complex SMO evolution. Through experiments and tests in earlier stages, the administrator has determined the system is behaving as desired, and thus the system commits the evolution.

As part of the process, the system purges all unnecessary tables that were kept for rollback purposes. It also deletes the triggers that propagate changes back and forth between new schema and old schema. From this point on, the old tables will cease to exist, and the old clients will no longer be able to access them. Any materialized views are converted into tables using the newly added `CONVERT` command, so that the link between the materialized view and the original tables are taken down.

The system is now ready for future schema evolution. Note that we could also remove the proxy from the path of the query system if we wish to at this point.

## 4.4 Rollback Process for Schema Evolution

In the unfortunate case where the schema upgrade is not successful, we can roll back the schema change without the application changing any of its behaviors. We will discuss how this works for simple SMOs, then complex SMOs, and finally how rollback and recovery works for a series of SMOs composed together.

Note that not all SMOs are reversible. As mentioned before, for complex SMOs that transition from V1 to V2, the system keeps them in sync by creating triggers on both the original table and newly created views. However, some complex SMOs do not have a way to map all the changes to V2 back to changes in V1. An example of an inreversible operation is the general case of the JOIN operation. Later in this section, we will discuss in more detail why JOINs are not reversible. For this set of SMOs, we can force the reverse operation but we would suffer some data loss and inconsistency. Hence, we made the decision to simply label these as irreversible operations, and allow database administrators to manually reverse these operations or avoid these operations entirely by choosing an alternative restructuring plan. Table 4.4 lists all the complex SMOs and whether they are reversible.

### Simple SMO Rollback

Since simple SMOs usually have a single or a few lines of SQL code associated with its action, it is relatively easy to reverse the effect of such actions. We implement `RollbackSMO` method in these SMO classes. To support the rollback operation, some SMOs such as `DROP_TABLE` and `DROP_COLUMN` do not actually drop the table or the column when first called, they merely rename them and render them hidden from the clients. They are actually removed from the database when the schema change is committed.

### Complex SMO Rollback

The steps involved in a rollback for a complex SMO are actually not very complex. This is mainly because the difficult part of keeping the older version and the newer version consistent is already done in other stages of the operation.

Still, complex SMOs transitioning from V1 to V2 have several tasks to perform when a rollback is initiated. To implement rollback, each complex SMO class has a rollback method where it would implement the following logic. First, it must stop any future updates to any tables in V2, in our system, this is done by removing the associated tables. Also, it must remove the triggers placed on V1 and V2 tables. After the rollback, the proxy also must ensure no more access to V2 is granted to clients, because such requests would result in errors.

### Cascading Rollback

In some cases, the administrator may wish to rollback to a certain point in a chain of upgrade processes. In Ratchet, we have a tree that keeps track of all historical operations and their dependencies. The administrator can specify that he/she wants to go back to a particular node in this dependency graph. The system will traverse the tree and undo any operation that happened after that particular operation.

However, as mentioned before, there are operations that can not be safely undone. When the system encounters this kind of operation it has three choices. It can forbid such operations. This limits the expressiveness of the SMO, but guarantees reversibility. The second choice is to automatically commits all the operations prior to any irreversible operation. A third alternative is to give the database administrators an opportunity to manually undo those operations that can not be undone automatically. The system can continue the rollback process when the manual undo is complete.

## Irreversible SMOs

Certain cases of `JOIN` operators do not have a suitable reverse operation. This is mainly due to the complexity and the flexibility of joins. Because in the general case, the join condition could be any arbitrary logic expression. This gives great expressiveness to the join operation, however it makes reversing the general join operation nearly impossible.

However, one of the most common uses of the `JOIN` operator is reversible. The reason for its reversibility relates to the notion of lossless join decomposition. There are two kinds of decompositions: Lossy decomposition and lossless join decomposition.

The decomposition of relation R into R1 and R2 is lossy when the join of R1 and R2 does not yield the same relation as in R. The join of R1 and R2 could miss entries in R or produce spurious entries not found in R. If the original tables were a lossy decomposition of the resulting join, writing a reverse propagation rule for our JOIN operation becomes impossible, because there simply isn't a corresponding R1 or R2 that would produce the JOIN result that we want.

However, when the original tables to be joined are the lossless join decomposition of the resulting joined table. It is possible to find such R1 or R2 such that R1 join R2 produces the modified view. To be a lossless join, the attributes involved in the join must be a candidate key for one of the original tables. That is also the condition for a JOIN, more specifically an equijoin to be reversible.

Example of such a join looks like this.

```
JOIN TABLE R,S INTO T WHERE R.colx = S.coly
```

In this case, either colx is a key for table R or coly is a key for table S. For any insert entry E into T, we can divide E into $E_R$ and $E_S$ to indicate the subset of columns coming from table R and table S. We insert $E_R$ into R if it does not already exist in R. Similarly we insert $E_S$ into S if it does not already exist in S.

For any delete entry E from T, we can divide E into $E_R$ and $E_S$ similarly. We only remove $E_R$ from R if there is no other entry in T that contains $E_R$. Also we remove $E_S$ from S if there is no other entry in T that contains $E_S$.

For an update on T, it is simply a deletion followed by an insertion. We just follow the reverse propagation rules of a deletion followed by an insertion.

# Chapter 5

# Evaluation

In this chapter, we evaluate Ratchet with respect to the goals we set out at the beginning of this dissertation. Specifically, we evaluate how general our approach to online schema update is, how it impacts foreground workload, and its rollback functionalities.

## 5.1 Validating correctness and reliability

We first would like to evaluate the effectiveness of our system in automatically handling various schema changes. For this, we are using a dataset that was published by Curino et al. [9]. This dataset contained a total of 170 schema changes and described them using SMOs. In our experiment, we are able to upgrade all 170 schema changes with our tool. We discovered two bugs in the dataset. Of the 170 changes, 168 changes can be performed automatically. Initially, two changes required some manual edits. This prompted us to add type information to SMOs. After the addition of type information, all upgrades were automated. In the following section, we first give some background on the dataset that we are using, and then describe our experience evolving these 170 schema changes using Ratchet.

### Background on the dataset: Mediawiki and Wikipedia

MediaWiki is the underlying software powering many wiki sites, most notably Wikipedia. Wikipedia has over 5 million articles in its English version alone and currently ranks as the fifth most visited website in the world according to Alexa Rankings. Mediawiki, as its underlying platform, has gone under many changes in its history. Because of its open-source nature, each version of its database schema is preserved in its version control history. During its first 4 years and 7 months of life, MediaWiki underwent 170 schema changes. Curino et al. extracted these changes and provided the basis for our further analysis and evaluation. Their study, and consequently our evaluation, consists of all the versions from MediaWiki's initial public version of 1.1 to the version 1.11 published in November 2007. For a detailed study on the version histories, please see their paper [8]. Here, we primarily focus on how our tool performed the 170 schema changes, the challenges and the lessons learned from handling these schema changes.

## Published dataset on schema changes

First, we take a look at the schema changes published in the previous study. The dataset includes MySQL initialization scripts that create the database schema for each of the 171 versions. This was extracted from the version history of the MediaWiki software. Along with these database commands, there are 170 sets of commands that describe how the schema changes from one version to the next. These schema changes are described using Schema Modification Operators, which our system is based on. Thus, Ratchet can directly consume these as commands and automatically evolve the schema versions. The following is an example of SMO representation from version 36 to version 37. It consists of three SMO changes, all of which are RENAME COLUMN changes.

```
Smo V(36,37) := {
      RENAME COLUMN ur_id IN user_rights TO ur_user;
      RENAME COLUMN ug_uid IN user_groups TO ug_user;
      RENAME COLUMN ug_gid IN user_groups TO ug_group;
};
```

There are broadly two types of schema changes. Some modify the actual table structure of the database schema, and therefore require changes to the queries. Some simply modify the DBMS engines, indexes, data types, and while they often impact performance of the queries, they do not require changes to the queries. Of the 170 schema changes, 94 or approximately 55% actually changed the table designs. The other 76 changes either changed database indexes only, or were bug fixes or documentation changes. We use NOP commands to represent these changes in our input to the system.

These schema changes also vary greatly in terms of complexity. As mentioned above, many schema version changes do not actually change the structure of the tables. We label these as having 0 SMOs in the schema change. Figure 5.1 shows

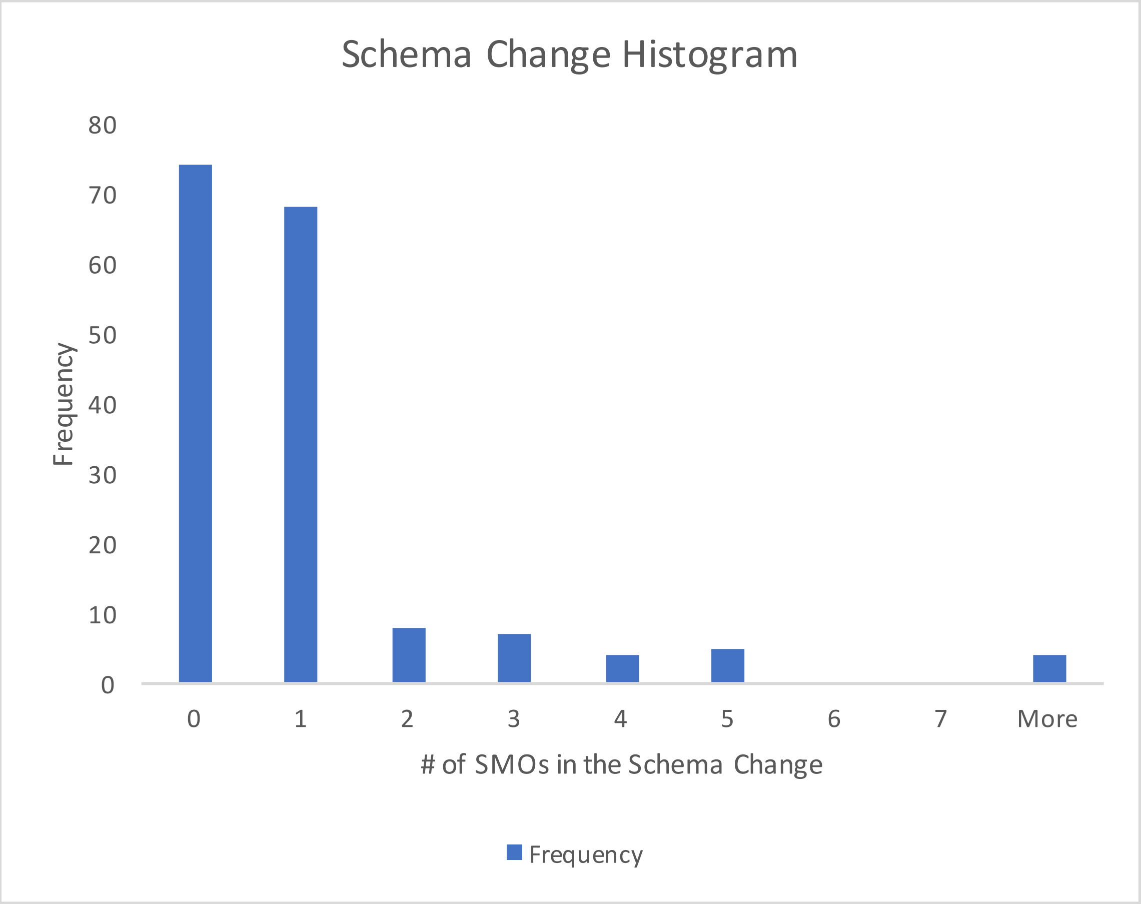


Figure 5.1: Histogram: Number of SMOs in the Schema Change, showing a bimodal pattern. Most of the schema changes are quite short, but a few can consist more than 20 SMOs

a histogram of all 170 schema changes categorized by the number of SMOs in each change. Most of them (83.5%) have zero or one SMOs. However, there are a number of complicated changes, one of them having 96 SMOs in a single version change. In the next section, we will see how Ratchet handles these different schema changes.

We further analyzed the 170 schema changes in terms of simple vs complex SMOs, and found that only 3 of 170 contained complex SMOs. This indicates that most of the changes are small changes or compositions of simple changes, as applications develop from version to version.

We also found that 100% of the SMOs are reversible SMOs. This allows for fully automated rollback should the administrator discover problems during the schema upgrade process.

## Performing schema changes using Ratchet

Since we are using PostgreSQL as our database, in order to initialize our database, we took the initial version of the mysql initialization script and converted it into a PostgreSQL compatible script. This translation involved converting Mysql-specific types to PostgreSQL-specific types, such as converting auto increment types to SERIAL types.

We also populated this database with some artificial data. Because we are using a schema that is the very first version of MediaWiki (April 2003), none of the current wikipedia dumps we can find fits this schema. Hence we are using generated data.

After the database is initialized, we feed the file containing all the SMOs to Ratchet. In our experiment, Ratchet was able to handle all but two changes automatically, and we were able to discover one error in the SMOs dataset. Both cases requiring manual intervention were merge operations. To merge two tables together, the tables must have exactly the same number of columns with the same types. However, the existing `ADD COLUMN` operator of the SMO operation does not contain any type information. This information is also not found in the dataset. So in addition to a merge operation, some explicit type casting is needed before the merge could proceed. That required a total of five SQL statements to be entered into the database interactive terminal.

To address this problem, we extended the Schema Modification Operator, specifically the `ADD COLUMN` operator and `CREATE TABLE` operator with optional type information. Adding this information allows all the columns to have the correct type information. Therefore merge becomes fully automatic. We searched through the initialization scripts and found type information for all the columns being created and added them using our extended SMOs. Thereafter, the merge operations were successful automatically. In practice, when a database administrator uses Ratchet to upgrade a schema, he or she would have access to that information, and be able to specify that information in the SMOs.

We also found one error in the SMO dataset. For one of the merge operations, one table has four columns and one table has three columns. By looking at the schema initialization script, we determined that the dataset was missing a `DROP COLUMN` operation. We fixed the error by adding the missing `DROP COLUMN` operation.

After these two changes, we were able to automatically evolve the database schema from version 1 to version 170 without taking the system offline. This validates our approach of using SMOs to automatically perform schema evolution.

## 5.2 Case study: a Major Change in MediaWiki Schema

In this section, we take a look at the largest schema changes in the dataset that we have for MediaWiki. This change was committed on Dec 19, 2004. The comment at the time of the commit was as follows.

```
Merge SCHEMA_WORK into HEAD. Lots of changes, some things are probably broken:
* Page moves/overwrites are a little iffy
* Compression might not be working right
* Profit!
```

From the code history, it was clear that the developer worked on this schema change on a separate branch of the code. Without a tool like Ratchet, the developer's only choice was to try it out and see if it works. In this section, we will dive deeper into this particular schema change and see what was done at the time and what could be done with a tool like Ratchet.

### Schema Change Details

The approach that the Mediawiki developers took was a very sensible one considering the impact of this schema change. There is always a version-controlled tables.sql file under the maintenance directory. It is responsible for creating and initializing each version of new mediawiki installations. So this file was modified to reflect this schema change. In addition, the developers created documentation on a wiki to show how the proposed change would take place. Thanks to the extensive documentation, we are able to recreate this change and understand the intentions of the developers at the time.

In the old version of the schema, each page of the wiki was represented using two tables, a current version in the cur table and any historical versions of the articles in the old table. The cur and old tables originally had the following schema.

```
+-------------------+---------------------+------+-----+---------+-----------------+
| Field             | Type                | Null | Key | Default | Extra          |
+-------------------+---------------------+------+-----+---------+----------------+
| cur_id            | int(8) unsigned     |      | PRI | 0       | auto_increment |
| cur_namespace     | tinyint(2) unsigned |      | MUL |         |                |
| cur_title         | varchar(255) binary |      | MUL |         |                |
| cur_text          | mediumtext          |      |     |         |                |
| cur_comment       | tinyblob            |      |     |         |                |
| cur_user          | int(5) unsigned     |      | MUL | 0       |                |
| cur_user_text     | varchar(255) binary |      | MUL |         |                |
| cur_timestamp     | varchar(14) binary  |      | MUL |         |                |
| cur_restrictions  | tinyblob            |      |     |         |                |
| cur_counter       | bigint(20) unsigned |      |     | 0       |                |
| cur_is_redirect   | tinyint(1) unsigned |      |     | 0       |                |
| cur_minor_edit    | tinyint(1) unsigned |      |     | 0       |                |
| cur_is_new        | tinyint(1) unsigned |      |     | 0       |                |
| cur_random        | double unsigned     |      | MUL | 0       |                |
| inverse_timestamp | varchar(14) binary  |      |     |         |                |
| cur_touched       | varchar(14) binary  |      |     |         |                |
+-------------------+---------------------+------+-----+---------+----------------+
+-------------------+---------------------+------+-----+---------+----------------+
| Field             | Type                | Null | Key | Default | Extra          |
+-------------------+---------------------+------+-----+---------+----------------+
| old_id            | int(8) unsigned     |      | PRI | NULL    | auto_increment |
| old_namespace     | tinyint(2) unsigned |      |     | 0       |                |
| old_title         | varchar(255) binary |      |     |         |                |
| old_text          | mediumtext          |      |     |         |                |
| old_comment       | tinyblob            |      |     |         |                |
| old_user          | int(5) unsigned     |      |     | 0       |                |
| old_user_text     | varchar(255) binary |      |     |         |                |
| old_timestamp     | varchar(14) binary  |      |     |         |                |
| old_minor_edit    | tinyint(1)          |      |     | 0       |                |
| old_flags         | tinyblob            |      |     |         |                |
| inverse_timestamp | varchar(14) binary  |      |     |         |                |
+-------------------+---------------------+------+-----+---------+----------------+
```

As shown here, these two tables actually contained a lot of redundant information. cur_id represents a unique index for a given article's current version. old_id represents an article's past revisions. Information was frequently moved from the cur table into the old table as pages were updated and the newer version of the page became the current version.

As of MediaWiki version 1.5, these two tables were restructured and merged into a single page table and revision table. The structure of these tables looks like the following.

```
+-------------------+---------------------+------+-----+---------+----------------+
| Field             | Type                | Null | Key | Default | Extra          |
+-------------------+---------------------+------+-----+---------+----------------+
| page_id           | int(8) unsigned     | NO   | PRI | NULL    | AUTO_INCREMENT |
| page_namespace    | int(11)             | NO   | MUL | NULL    |                |
| page_title        | varchar(255) binary | NO   |     | NULL    |                |
| page_restrictions | tinyblob            | NO   |     | NULL    |                |
| page_counter      | bigint(20) unsigned | NO   |     | 0       |                |
| page_is_redirect  | tinyint(1) unsigned | NO   |     | 0       |                |
| page_is_new       | tinyint(1) unsigned | NO   |     | 0       |                |
| page_random       | real unsigned       | NO   | MUL | NULL    |                |
| page_touched      | char(14) binary     | NO   |     | NULL    |                |
| page_latest       | int(8) unsigned     | NO   |     | NULL    |                |
| page_len          | int(8) unsigned     | NO   | MUL | NULL    |                |
+-------------------+---------------------+------+-----+---------+----------------+
+----------------+---------------------+------+-----+---------+----------------+
| Field          | Type                | Null | Key | Default | Extra          |
+----------------+---------------------+------+-----+---------+----------------+
| rev_id         | int(8) unsigned     | NO   | PRI | NULL    | AUTO_INCREMENT |
| rev_page       | int(8) unsigned     | NO   | PRI | NULL    |                |
| rev_text_id    | int(8) unsigned     | NO   |     | NULL    |                |
| rev_comment    | tinyblob            | NO   |     | NULL    |                |
| rev_user       | int(5) unsigned     | NO   | MUL | 0       |                |
| rev_user_text  | varchar(255) binary | NO   | MUL | NULL    |                |
| rev_timestamp  | char(14) binary     | NO   | MUL | NULL    |                |
| rev_minor_edit | tinyint(1) unsigned | NO   |     | 0       |                |
| rev_deleted    | tinyint(1) unsigned | NO   |     | 0       |                |
+----------------+---------------------+------+-----+---------+----------------+
```

Before the change, each revision of a page was either a row in cur table or a row in old table. Now instead, it is a row in the revision table. Every page is a row in the page table, and has a page_latest entry which is a foreign key into the revision table. So that the current or the latest change can be easily looked up. At the same time, each revision contains a back reference to the page via the `rev_page` entry.

In addition, the change creates a separate `text` table where all the text for the revision is actually stored. This `text` table has only three columns and is structured as follows. Revisions and text are linked together via `rev_text_id` which is a foreign key into the `text` table.

```
+-----------+-----------------+------+-----+---------+----------------+
| Field     | Type            | Null | Key | Default | Extra          |
+-----------+-----------------+------+-----+---------+----------------+
| old_id    | int(8) unsigned | NO   | PRI | NULL    | AUTO_INCREMENT |
| old_text  | mediumblob      | NO   |     | NULL    |                |
| old_flags | tinyblob        | NO   |     | NULL    |                |
+-----------+-----------------+------+-----+---------+----------------+
```

Because this was an important change to the core storage of the pages in mediawiki, we were able to find plenty of documentation on this. Figure 5.2 shows how the columns from the old version map to the new version of the schema.

From this figure, it is clear that many columns were direct mappings from the old schema to a new location in the new schema. However, their names were changed. We believe this kind of additional documentation was necessary because the transformation itself did not contain this piece of information. The schema transformation was documented with two different initialization scripts in the code base. Comparing these two initialization scripts yields some information, but not enough to reconstruct Figure 5.2. Figure 5.3 shows the differences between the two initialization scripts.

Additionally, MediaWiki developers provided a custom php script that will convert a database from the old version to the newer version. This particular script is 134 lines long and consists of mostly sql statements mixed with php code. This script served as an important reference to construct SMOs for our system, since it contained how data was transformed from one schema to the next.

In summary, when the developers carried out this particular schema upgrade, there was lots of documentation to support it, indicating that the developers were very careful and diligent about this schema upgrade. It is also a sign that these kind of sweeping changes could potentially lead to a lot of additional work for the developers without proper tooling support.

## Schema Change with Ratchet

In this section, we walk through how we used Ratchet to achieve this particularly complex schema upgrade.

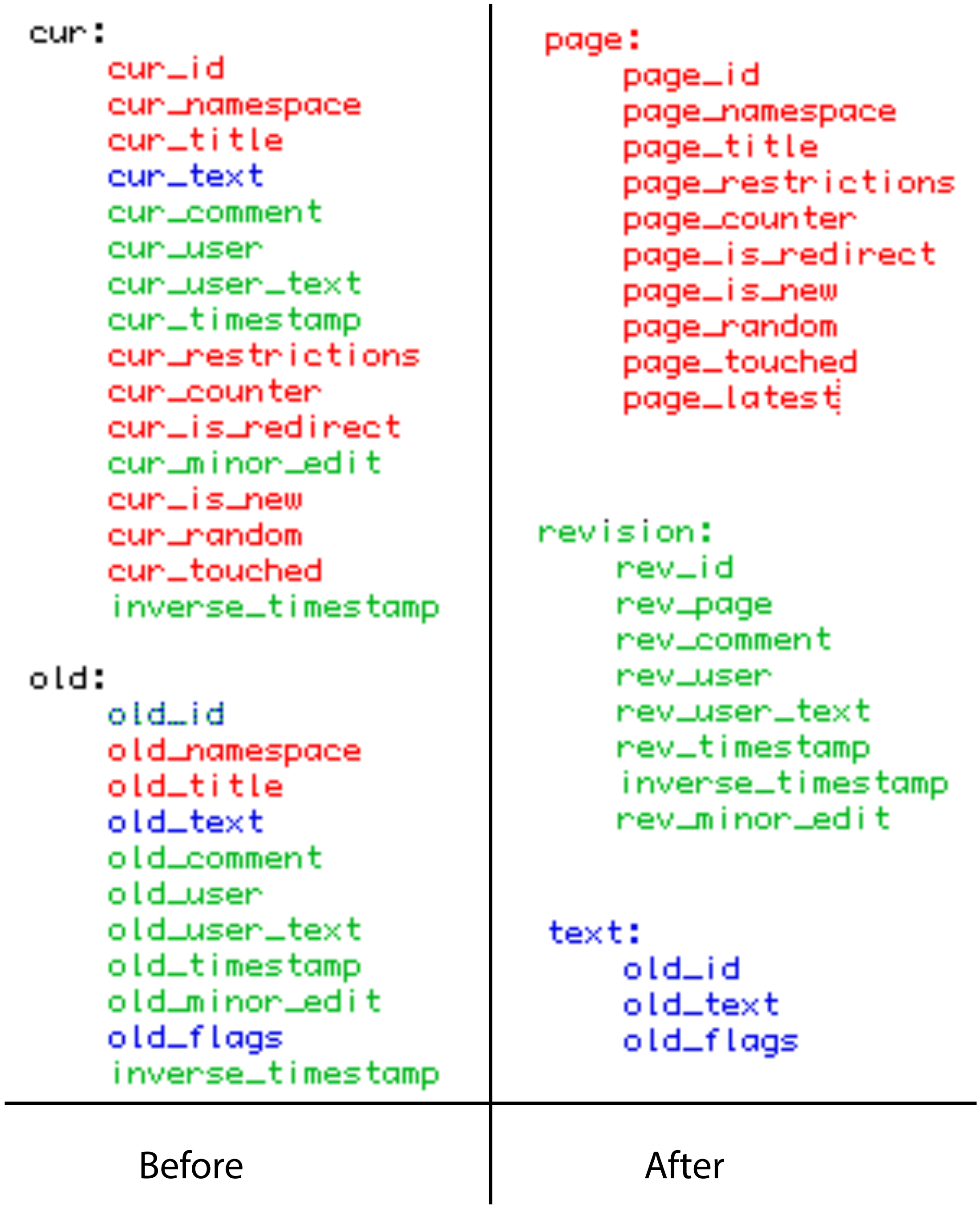
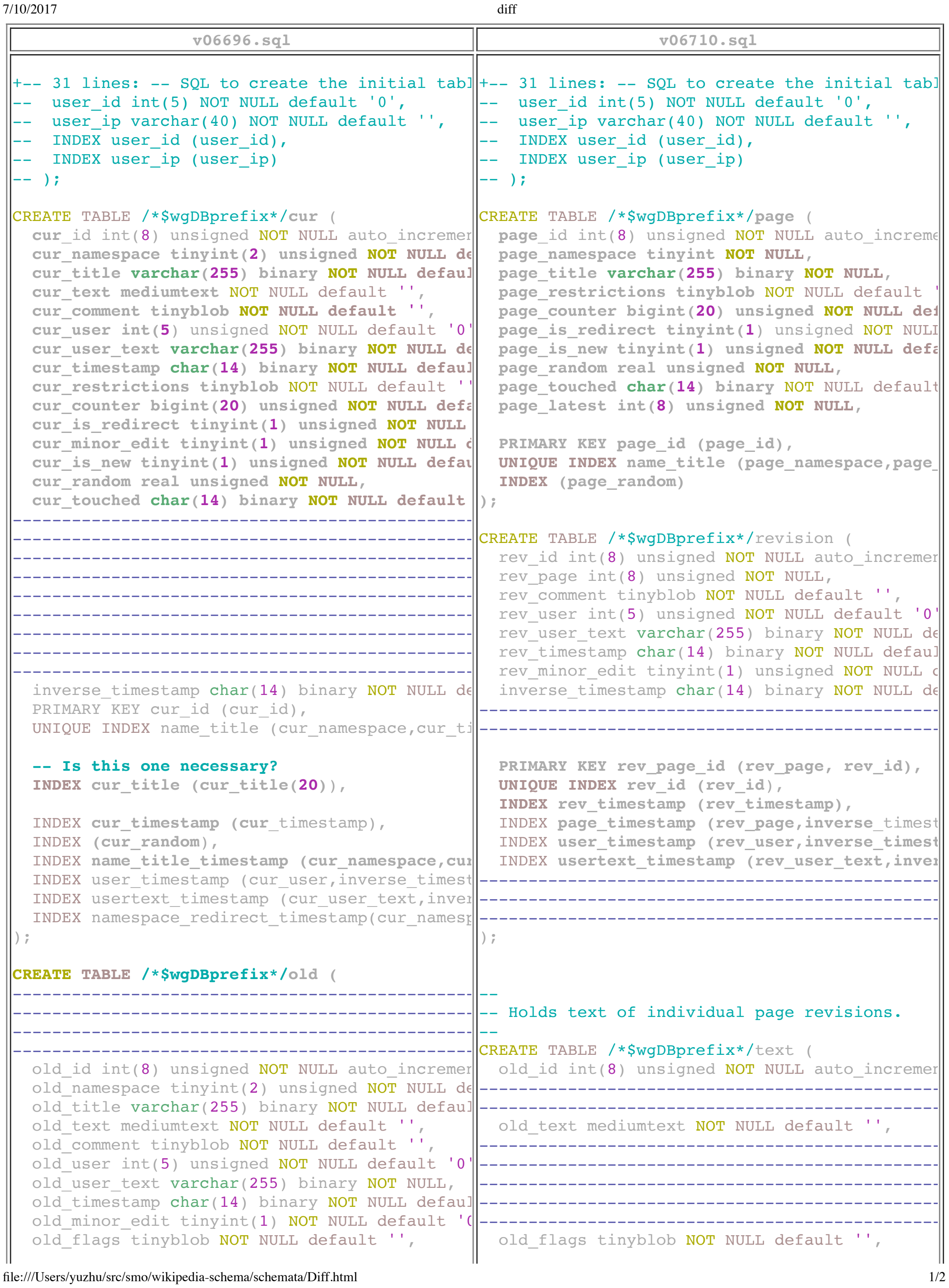


Figure 5.2: Database Restructure
Column colors in the old schema represent which table they belong to in the new schema
Source: `https://www.mediawiki.org/wiki/Proposed_Database_Schema_Changes/October_2004`



7/10/2017 diff

v06696.sql

```
+-- 31 lines: -- SQL to create the initial tabl
--  user_id int(5) NOT NULL default '0',
--  user_ip varchar(40) NOT NULL default '',
--  INDEX user_id (user_id),
--  INDEX user_ip (user_ip)
-- );

CREATE TABLE /*$wgDBprefix*/cur (
  cur_id int(8) unsigned NOT NULL auto_incremen
  cur_namespace tinyint(2) unsigned NOT NULL de
  cur_title varchar(255) binary NOT NULL defaul
  cur_text mediumtext NOT NULL default '',
  cur_comment tinyblob NOT NULL default '',
  cur_user int(5) unsigned NOT NULL default '0'
  cur_user_text varchar(255) binary NOT NULL de
  cur_timestamp char(14) binary NOT NULL defaul
  cur_restrictions tinyblob NOT NULL default ''
  cur_counter bigint(20) unsigned NOT NULL defa
  cur_is_redirect tinyint(1) unsigned NOT NULL 
  cur_minor_edit tinyint(1) unsigned NOT NULL d
  cur_is_new tinyint(1) unsigned NOT NULL defau
  cur_random real unsigned NOT NULL,
  cur_touched char(14) binary NOT NULL default 
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
  inverse_timestamp char(14) binary NOT NULL de
  PRIMARY KEY cur_id (cur_id),
  UNIQUE INDEX name_title (cur_namespace,cur_ti

  -- Is this one necessary?
  INDEX cur_title (cur_title(20)),

  INDEX cur_timestamp (cur_timestamp),
  INDEX (cur_random),
  INDEX name_title_timestamp (cur_namespace,cur
  INDEX user_timestamp (cur_user,inverse_timest
  INDEX usertext_timestamp (cur_user_text,inver
  INDEX namespace_redirect_timestamp(cur_namesp
);

CREATE TABLE /*$wgDBprefix*/old (
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
  old_id int(8) unsigned NOT NULL auto_incremen
  old_namespace tinyint(2) unsigned NOT NULL de
  old_title varchar(255) binary NOT NULL defaul
  old_text mediumtext NOT NULL default '',
  old_comment tinyblob NOT NULL default '',
  old_user int(5) unsigned NOT NULL default '0'
  old_user_text varchar(255) binary NOT NULL,
  old_timestamp char(14) binary NOT NULL defaul
  old_minor_edit tinyint(1) NOT NULL default '(
  old_flags tinyblob NOT NULL default '',
```

v06710.sql

```
+-- 31 lines: -- SQL to create the initial tabl
--  user_id int(5) NOT NULL default '0',
--  user_ip varchar(40) NOT NULL default '',
--  INDEX user_id (user_id),
--  INDEX user_ip (user_ip)
-- );

CREATE TABLE /*$wgDBprefix*/page (
  page_id int(8) unsigned NOT NULL auto_increme
  page_namespace tinyint NOT NULL,
  page_title varchar(255) binary NOT NULL,
  page_restrictions tinyblob NOT NULL default '
  page_counter bigint(20) unsigned NOT NULL def
  page_is_redirect tinyint(1) unsigned NOT NULL
  page_is_new tinyint(1) unsigned NOT NULL defa
  page_random real unsigned NOT NULL,
  page_touched char(14) binary NOT NULL default
  page_latest int(8) unsigned NOT NULL,

  PRIMARY KEY page_id (page_id),
  UNIQUE INDEX name_title (page_namespace,page_
  INDEX (page_random)
);

CREATE TABLE /*$wgDBprefix*/revision (
  rev_id int(8) unsigned NOT NULL auto_incremen
  rev_page int(8) unsigned NOT NULL,
  rev_comment tinyblob NOT NULL default '',
  rev_user int(5) unsigned NOT NULL default '0'
  rev_user_text varchar(255) binary NOT NULL de
  rev_timestamp char(14) binary NOT NULL defaul
  rev_minor_edit tinyint(1) unsigned NOT NULL d
  inverse_timestamp char(14) binary NOT NULL de
-----------------------------------------------
-----------------------------------------------

  PRIMARY KEY rev_page_id (rev_page, rev_id),
  UNIQUE INDEX rev_id (rev_id),
  INDEX rev_timestamp (rev_timestamp),
  INDEX page_timestamp (rev_page,inverse_timest
  INDEX user_timestamp (rev_user,inverse_timest
  INDEX usertext_timestamp (rev_user_text,inver
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
);


--
-- Holds text of individual page revisions.
--
CREATE TABLE /*$wgDBprefix*/text (
  old_id int(8) unsigned NOT NULL auto_incremen
-----------------------------------------------
-----------------------------------------------
  old_text mediumtext NOT NULL default '',
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------
  old_flags tinyblob NOT NULL default '',
```

file:///Users/yuzhu/src/smo/wikipedia-schema/schemata/Diff.html 1/2

Figure 5.3: Visual Diff Between Initialization Scripts

First, we obtained a series of SMOs derived from the pictorial representation of the schema change and the upgrade script. This process was already done in the dataset, but we modified it to add type information to columns. This resulted in a total of 96 SMOs. A detailed listing can be found in Appendix B.

Analyzing these 96 SMOs, they roughly divide into seven major steps.

1. we construct cur_page from Table cur.
2. we construct cur_revision from the Table cur.
3. we construct cur_text from the Table cur.
4. we construct old_page from the Table old.
5. we construct old_revision from the Table old.
6. we rename the table old to old_text.
7. we merge cur_text with old_text to get new Table text, merge cur_page with old_page to get new Table page, and merge cur_revision with old_revision to get new Table revision.

Most of these SMOs are quite simple. The first six steps are preparing tables to have the exact same columns so they can be merged in Step 7. We will take a look at a couple of complex SMOs to see how we expressed the schema transitions.

### Custom Functions

There are four SMOs that used custom functions. They look like this:

```
ADD COLUMN page_id AS function41a(page_namespace, page_title) INTO cur_page;

ADD COLUMN rev_page AS function41b(rev_id) INTO cur_revision;

ADD COLUMN page_id AS function41c(page_namespace, page_title) INTO old_page;

ADD COLUMN rev_page AS function41d(old_namespace,old_title) INTO old_revision;
```

Function41a is a function that looks up either the curr_id field or the old_id field from the cur or old table based on page_namespace and page_title. Function41b is a function that takes rev_id, which is an id for a revision and looks up the page associated with that revision in the old table. Function41c is very similar to function41a, and looks up the old_id field from the old table and inserts it into the old_page table.

Function41d looks up the revision page based on old_namespace and old_title from the old table.

These custom functions essentially allow us to express these new columns as expressions of existing columns and generate them on the fly. For example, function41b(rev_id) is actually just a table lookup where rev_page is the page id of the page that rev_id revision is a part of. Allowing these kind of custom functions made the syntax of SMOs relatively concise, yet very expressive, because we can perform joins and table lookups to generate the new columns. Quite different from a regular join to look up data, this particular operation is trivial to reverse. We simply drop the column.

### Merge Operation

Towards the end of this schema operation, we have three merge operations in which we merge the current tables (the tables that reference the current revision of the wiki pages) with the old tables (the tables that reference the historical revisions of the wiki pages). These are expressed with these SMOs.

```
MERGE TABLE cur_text, old_text  INTO text;

MERGE TABLE cur_page, old_page  INTO page;

MERGE TABLE cur_revision, old_revision  INTO revision;
```

As mentioned in earlier sections, for the merge operation to work, the columns of the current and old versions of the tables must match exactly. Most of the SMOs before these three are preparation to get the columns to match by adding and removing columns from old and current versions of the tables.

### Summary

In this section, we walked through the experience of evolving the most complex schema change in our dataset using our tool Ratchet. We showed how a complex change may be broken into several logical steps and using a merge step at the end to merge all the tables together. We were able to complete these changes automatically.. Using this example, we also explained techniques such as custom functions that allow even more expressibility in the SMOs. This exercise demonstrates that Ratchet is capable of handling the complexity found in real world scenarios.

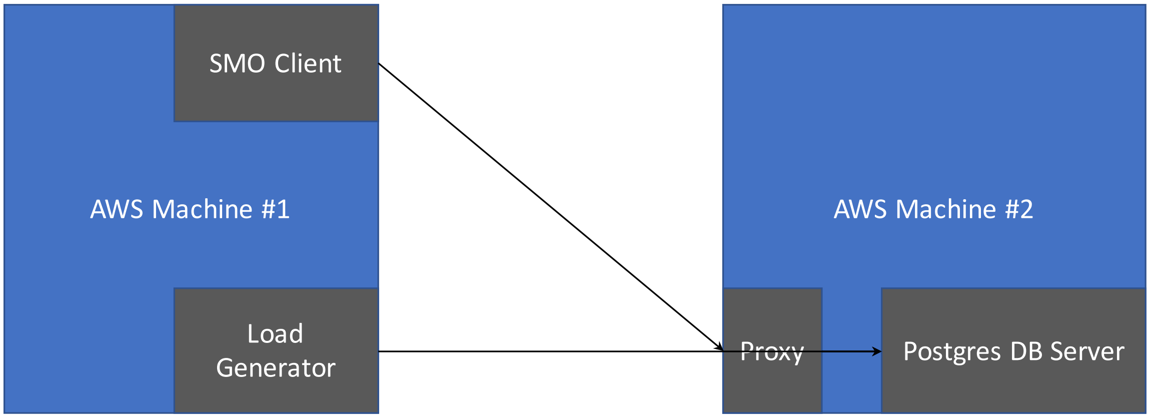


Figure 5.4: Experiment Setup

## 5.3 Performance

In addition to being able to handle various schema upgrade requests, one important aspect of our project is being able to handle them while the system is online. To evaluate this aspect, we setup the experiment with a load generator that constantly issue queries to the back end database. In the meantime, we issue several SMOs to see how the throughput and latency is impacted by our background traffic. In the following sections, we detail our experiment setup, load generation, and present the results and lesson learned.

### Experiment Setup and Load Generation

We first discuss how we set up our experiment. We use two similarly configured machines in the AWS cluster. All of our following tests are conducted using Amazon EC2 instance type m4.large machine hosting the PostgreSQL database server. The version of PostgreSQL we are running is a modified version of PostgreSQL 9.5 release. This particular type of instance has two virtual CPUs, and 8GBs of memory, with 450Mbps of dedicated EBS bandwidth.

As shown in Figure 5.4, one of the machines hosts the database server and our proxy. The other one hosts the client and our load generator. The client takes SMO inputs and initiates schema changes, while the background load generator constantly issues request to the database backend. We then observe how our schema change affects the serving speed measured in queries per second.

To examine how the database reacts to different types of workload, we generate two mixes of query loads A and B. In query mix A, we simulate a read-mostly query load, with reads being 90% of the total mix, and writes the remaining 10%. In query mix B, we simulate a read/write query mix, with reads and writes each making up 50% of the query. In the next section, we will analyze the results of these experiment and make recommendations on further improvements on online schema evolution.

## Analysis of Decomposition Operation

In this section, we take a look at the `DECOMPOSE TABLE` operation. This operation splits a table into two tables, each containing a subset of columns from the original table. These two tables could also share some columns, usually the primary key of the original table.

In this experiment, we run the decompose operation on a large table, with 10 million rows of entries, so the operation itself will take some time to see the effect it has on the foreground queries. As mentioned before, we run two sets of generated queries against the database, at the same time. One is a set containing mostly (90%) read queries, the other one is a set containing about half read queries and half write queries. Note that these queries are issued against the same database, but not necessarily the same table which is undergoing the decompose operation. We measure the number of queries completed in the previous second, and report that as the instantaneous QPS.

Figure 5.5 shows how read-mostly queries vary under the stress of a background schema evolution. Schema evolution, in this case a decomposition of a table starts around the 10 second mark and ends at the 39 second mark. Before 10 seconds, we establish a baseline of about 12-13 queries per second. After the schema evolution starts, it drops down to 6 queries per second at the lowest point. This represents a drop of roughly 50% in query answering capacity.

Let us examine Figure 5.6 next, which represents the read/write queries' variation under schema evolution. Similar to the first graph, the decomposition of the table starts around 10 second mark and ends around 40 seconds. First, we notice the baseline is higher, at above 20 qps. This is because the read queries are mostly selection queries, and they are selecting from relatively large tables without indexes. Thus we can conclude, in the base case, read queries are more expensive than the write queries in our workload mix. However, as soon as schema evolution starts, the qps rate of the read-write query mix is affected much more than the read-mostly query mix, reaching an average qps of 7.5 while the view is materializing. The likely reason for this is the write queries in the query mix acquires more locks in the database and there is more contention of these locks as the schema evolution (itself a write-heavy operation) is taking place.

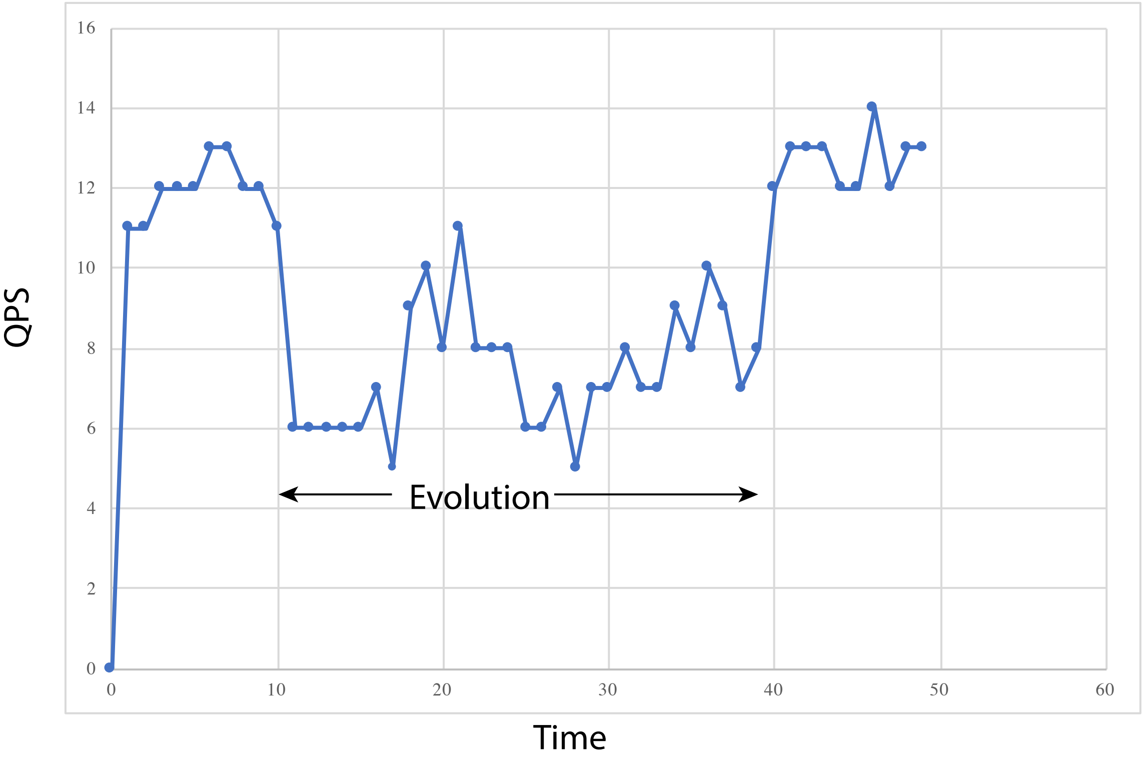


Figure 5.5: Read Mostly Query QPS under Schema Evolution

## Direct vs Delayed Trigger Update

In this section, we evaluate the effectiveness of delayed trigger update vs direct trigger update. As we mentioned before, creating materialized view automatically acquires an exclusive lock on the materialized view object, and can therefore block any triggers trying to update the materialized view. Consequently, any operations that generate these triggers are also blocked.

This is far from ideal when the materialized view is being built, and building materialized views can take quite a long time if the size of the original table is large, as we show below.

Here we look at an example. The desired schema operation is to merge two large tables with identical schema into one. The sizes of the tables are around 10 millions rows in total. Similar to before, we run two sets of generated queries against the database, one set being read-mostly queries, and the other being read-write queries. This time, however, all the queries are referencing one of the tables being merged. This is considered the worst case, because all the modifications will need

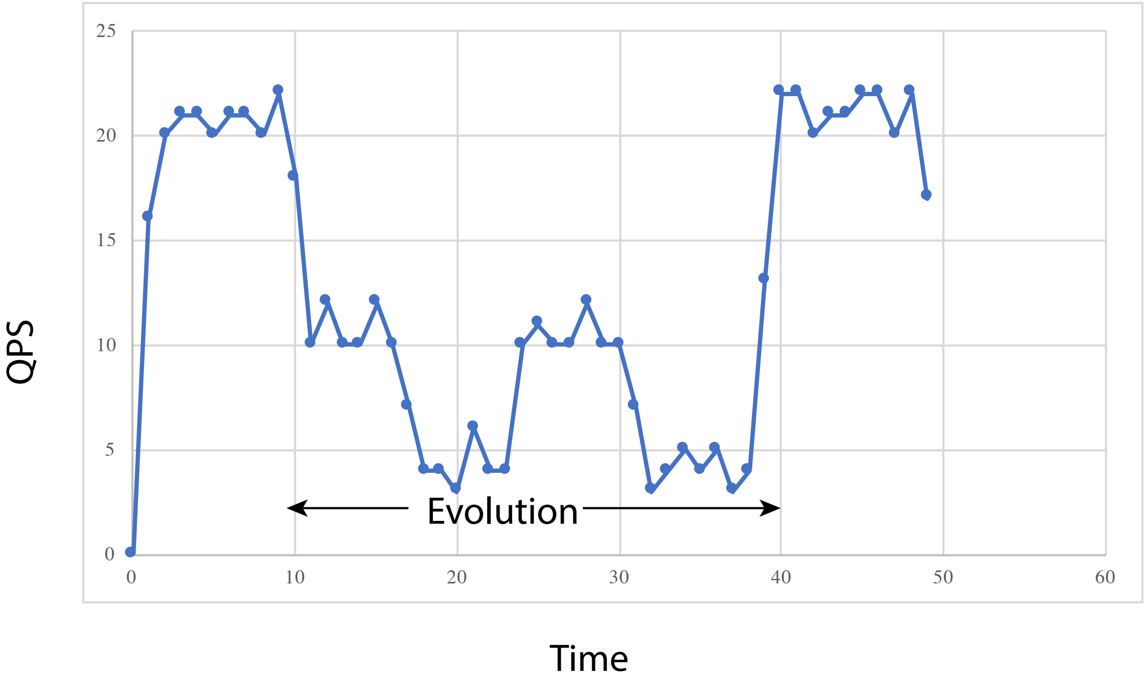


Figure 5.6: Read Write Query QPS under Schema Evolution

to be propagated to the materialized view. We report the instantaneous queries per second in the graphs below.

In Figure 5.7 and Figure 5.8, both graphs start their schema upgrade at the 10 second mark. The merge opeartion with direct trigger immediately dropped to zero QPS, and only recovered after the materialized view completely finish building itself. This is consistent with our earlier analysis of locking mechanism. The lock prevented any triggers from updating the materialized view, which in turn prevented any update queries on one of the tables being merged.

Looking at Figure 5.8, the one where we implemented delayed trigger update while the materialized view is building, the situation is much better. The foreground queries were still able to make progress at around 7 QPS per second. This was achieved without additional priority given to the foreground workload.

Figure 5.9 and Figure 5.10 show the QPS variance under a different type of workload, this time with half of the workload being writes to one of the original tables. Similar to the results we obtained above, when we are using Delay Trigger Update, foreground queries are able to make progress without being blocked by the background workload. When direct trigger update is used, everything comes to a halt.

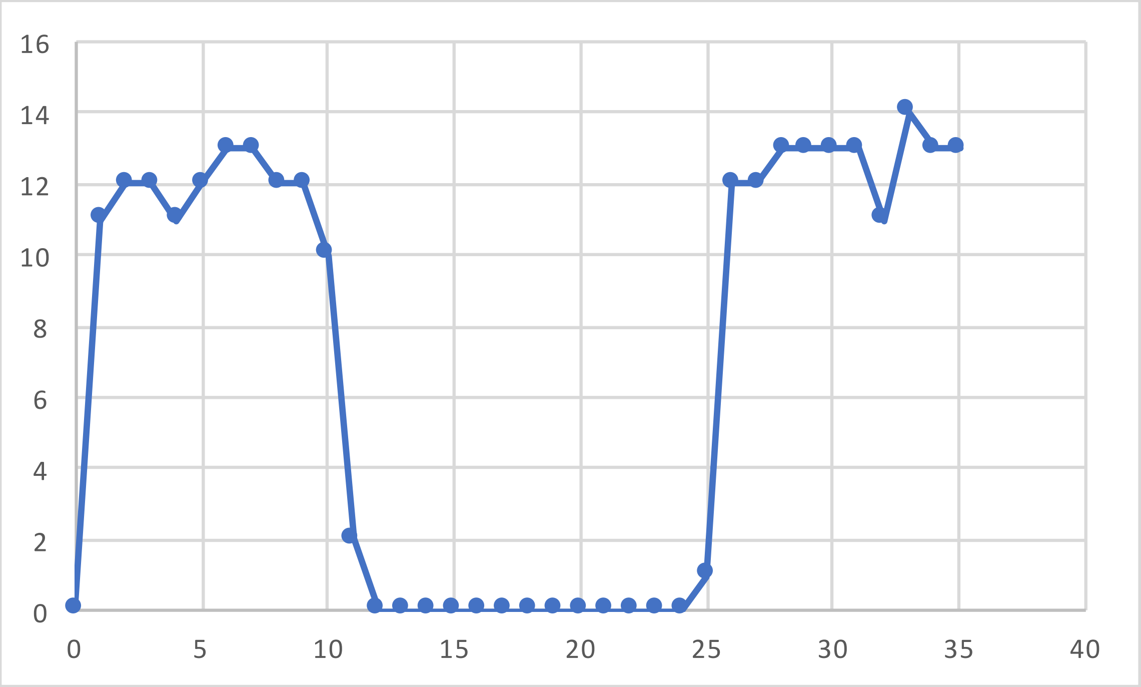


Figure 5.7: Read Mostly Query QPS with Direct Trigger while Merging

## Lessons learned and Recommendations

We learned a number of important lessons from the above evaluation. First, read-write workload gets affected more by schema evolution operations than read-mostly workloads. This is understandable because writes likely run into more locking conflict with the background operation and cause more disk accesses whereas read operations can read information cached in memory. Second, the last set of experiments shows that Delayed Trigger Update must be used when the materialized view is building itself to avoid blocking. Once it is built, the simpler and more efficient Direct Trigger Update can be used instead. Third, without priority levels, databases tend to split their resources between foreground and background tasks fairly.

Because read-only workload are affected less, schema upgrade operations should be done while the workload is read mostly and off-peak, to avoid heavy interference between foreground queries and the background schema upgrade.

Using a combination of delayed trigger update and direct trigger update during different stages of the upgrade process avoids unnecessary blocking due to locking. This is exactly what we have done in Ratchet.

Without explicitly supporting different priority levels for different requests, Post-

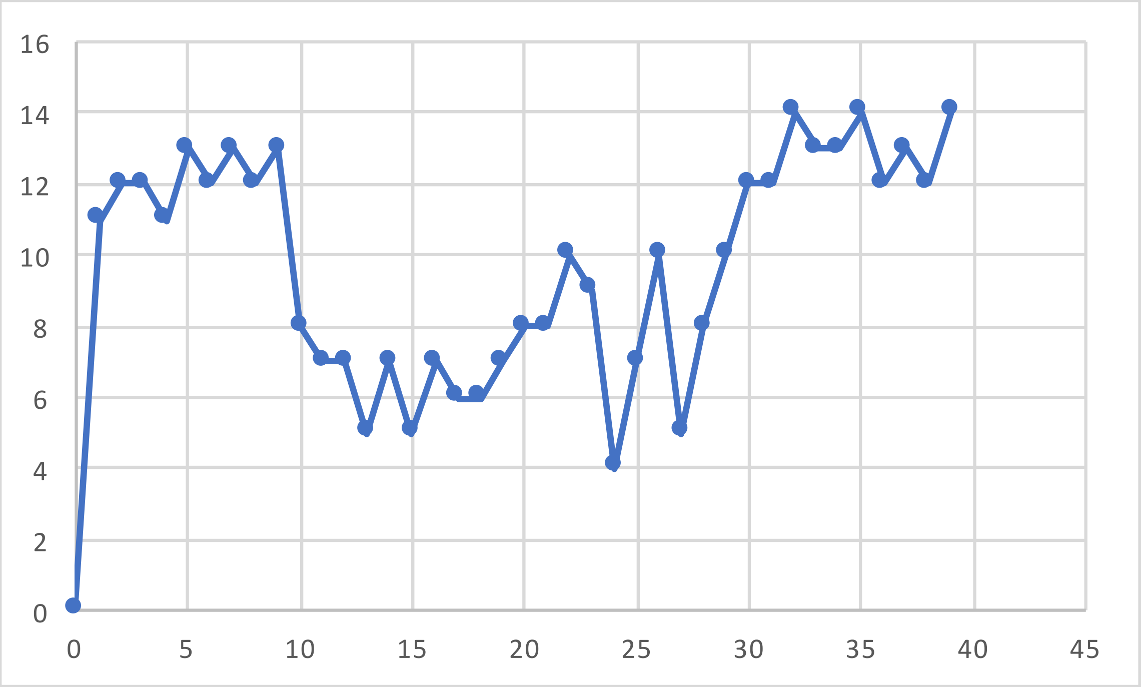


Figure 5.8: Read Mostly Query QPS with Delayed Trigger while Merging

gres treats foreground traffic (our generated query mix) and the background traffic (the schema evolution traffic) exactly the same. Hence, the foreground traffic is likely going to be affected by online schema evolution operations on our database schema with reasonable performance penalties. To further improve the performance of the foreground queries, we can use some of the client-based or server-based priority techniques as discussed in Section 1.1.

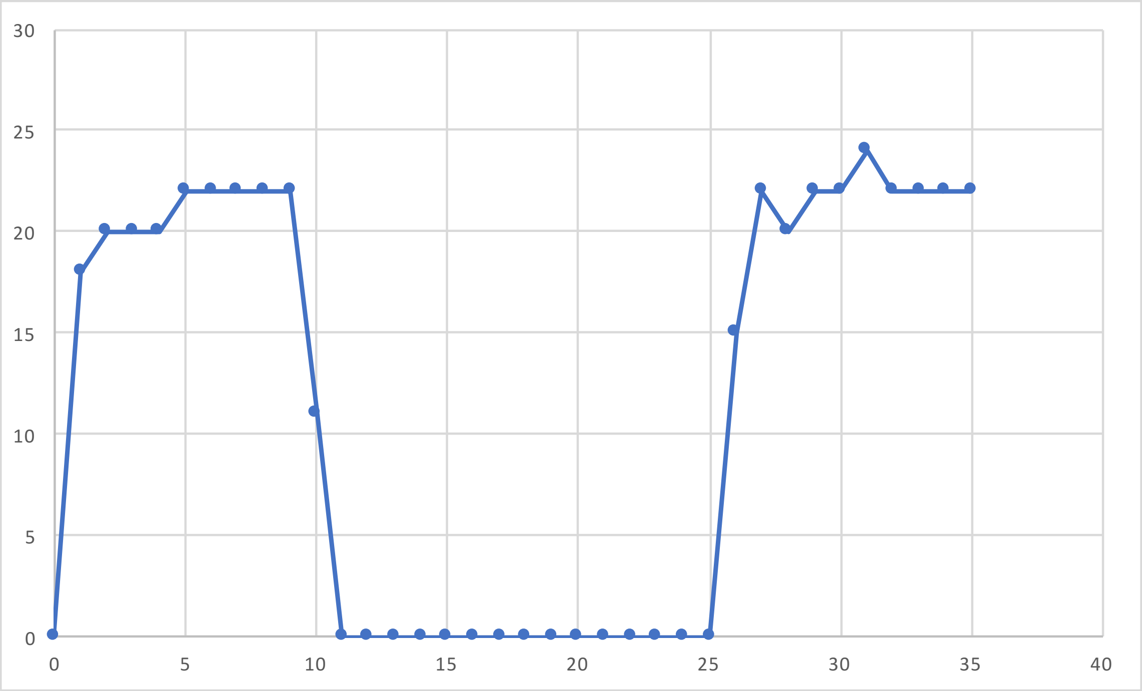


Figure 5.9: Read Write Query QPS with Direct Trigger under Schema Evolution

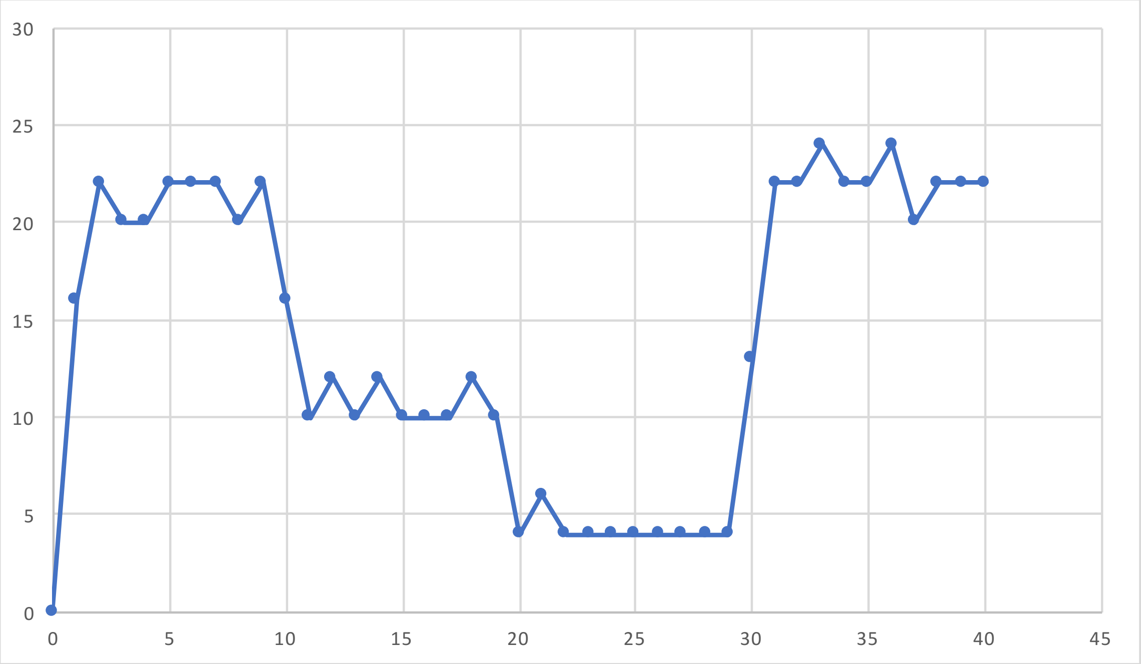


Figure 5.10: Read Write Query QPS with Delayed Trigger Update under Schema Evolution

# Chapter 6

# Related Work

## Schema Evolution in General

Schema change, in particular schema evolution, has been studied extensively in the research community [23]. According to the survey papers published in 1995 and in 2006, more than 300 hundred papers were published that are related to schema evolution. They tend to cover different aspects of schema evolution. Some focus on how to perform schema evolution on object-oriented databases[4] [20]. Some focus on the core operations that address the blocking nature of DDL statements[5] [1]. Some focus on how to recover and revert changes should unexpected events occur [3]. Some focus on how to keep each and every version history of the database, so this evolution history can be studied later[24]. Our work focuses on a design and implementation of a system that supports schema evolution online with acceptable level of interference and rollback capabilities.

## Oracle Edition-based Schema Evolution

Oracle 10g [19] supports online schema evolution by using an edition-based schema system. Each schema has a version number associated with it. Database clients can specify which edition the current connection is using. Additionally, the database provides a mechanism called a cross-view trigger that can take changes in one version of the schema and apply it to tables in another version of the schema. There are several steps to a schema evolution process in this setting. First, the database administrator creates a new schema version and designs the desired schema with the new version. In the meanwhile, the older version of the database continues to serve live traffic. Using cross-version triggers, the database can propagate any changes to the older version of the database to the newer version schema, and therefore keeping the newer version of the database consistent.

This solution is quite similar to ours in principle, but has several differences. Unlike our solution which builds on views, it is tightly integrated with the database, and requires additional feature support from the DBMS. Because the notion of editions is rather database specific, the client that uses this feature is therefore database specific. Additionally, it is rather heavyweight for simple changes such as adding and removing a column to a table, whereas we hide the mechanism to accomplish our schema transformation behind schema operators. This way, we can use simple SQL DDLs when changes are simple and non-blocking.

## Facebook OSC Script

Facebook has used a similar copy-based strategy and applied it to MySQL database. They created a suite of php scripts and released it in a blog post here [5]. It converts `ALTER TABLE` SQL commands into non-blocking operations by creating a copy of the table being changed. It uses database triggers to copy over the changes to the original table. Compared to this solution, our solution uses a logical operation unit SMO that is higher level than `ALTER TABLE` commands. SMOs serve as a convenient unit of rollback and commit.

We use the database's internal support for materialized views when possible rather than using a separate table. By using a combination of *in-place modification* and *copy-based strategies*, we can more efficiently handle simple schema changes to the database when the database supports it well.

## Spanner

Spanner[7] uses its TrueTime API to enable it to assign globally meaningful commit timestamps to its schema change transactions. Hence, it is able to atomically switch schemas while handling reads and writes. Our work is orthogonal to Spanner's work, and can be applied to a distributed environment given Spanner's ability to atomically switch schema.

## Vitess Schema Swap

Vitess[18] is a database solution for scaling MySQL, and extends many of MySQL's features with scalability of a NoSQL database. It is the backend of all YouTube database traffic. Vitess provides two ways of changing schema, a direct ApplySchema command and a Schema Swap approach. The ApplySchema command is used for short-running, simple schema changes, similar to our in-place update approach. The

Schema Swap approach is recommended for long running schema changes, similar to our copy-based approach.

At a high level, Schema Swap uses MySQL's statement based replication and backups to apply the changes to all tablets (replicas). It involves five basic steps.

1. Apply schema change to an offline replica
2. Let the offline replica catch up using replication, then creating a backup of it
3. Restore remaining replica but not the master from the backup
4. Failover the master to one of the replicas with the new schema, restore master from the backup
5. At this point, all tablets have the new schema, and applications can start using it.

Vitess's solution is best applied in a distributed setting with many replicas. This way, taking a replica offline would not drastically degrade the performance of the overall system. Our method can be similarly applied in a distributed setting with replication, and has the additional benefit of not taking a replica offline, but at the cost of more complexity within each replica.

Compared to Vitess, we support different clients accessing the version of schema before transition and the version of schema after transition simultaneously. We also support rollback based on SMOs.

## Amazon Aurora low-latency DDL

Amazon Aurora recently blogged about their approach to schema evolution [1]. Their approach is slightly different from ours, and is complementary. The key idea of their approach is to convert the synchronous action of updating schema and copying data into an asynchronous copy-on-write process.

When Aurora executes a DDL statement such as adding a column, it modifies the INFORMATION_SCHEMA table to reflect the change in the system meta table and records the old schema into a new system table. As soon as these steps are completed, the system returns control back to the users as if the DDL has completed.

Next, when any row of the table is modified, the system checks if there are any pending DDLs affecting that row. If there is, the DDL and DML are applied to the row at the same time. This essentially piggybacks the DDL operation on any future DML operations, and gains efficiency as a result. There is a caveat. When the relevant data is queried, the query engine modifies the row before returning it to the user, even though the underlying storage might have a mix of old and new schema formats.

The approach taken by Aurora is orthogonal to the approach we are taking. While they are reducing the perceived latency of the DDL operations, we are reducing the locking of actually performing the DDL operations. Aurora ultimately needs to perform the DDL operations if there simply are not enough DML operations to piggyback onto. Aurora's approach can be integrated into our system as well to further improve SMO's latency.

### NoSQL system schema evolution

Recently, NoSQL systems have gained popularity due to their flexibility and scalability. These datastores are often schemaless, often addressing database structure at the application layer. Scherzinger, Klettke, and Störl proposed a language that allows application developers to manage the schema evolution process on NoSQL system in a more systematic fashion [25].

Other approaches have been attempted in the NoSQL space as well. Various tools were developed to manage schema evolution and validate schema at application level for NoSQL databases[21]. The benefit of doing this is that database evolution is maintained in synchrony with code evolution, and version control systems maintain a copy for both the code and database structure.

# Chapter 7

# Conclusion

Schema evolution is becoming more important and more frequent in real-life deployment of applications. This thesis proposes Ratchet, a first step towards automated online schema evolution that offers both quality of service for the foreground application queries, and safety and reversibility needed by the database administrators.

Built on the previous work on SMOs, and using a combination of techniques such as in-place update and copy-based strategies, Ratchet handles all proposed SMOs in an efficient manner. Furthermore, it provides a mechanism to rollback any changes for all reversible SMOs and allows for combining manual reversal with automated reversal for schema updates that contain irreversible changes.

To evaluate how Ratchet would handle real-life scenarios, we obtained five years of schema change history of wikimedia, the underlying software for wikipedia. Ratchet was able to handle all cases of schema update automatically, after we added type information to our SMO implementation. In addition, we evaluated how the foreground query performance is affected when we launch a schema evolution. We added additional optimization to remove large locks from schema change process, significantly improved the foreground query performance during schema evolution.

# Appendices

## A Grammar Rules for Parsing Schema Modification Statement

```
grammar SMO;
prog: smo_statement_plus_semi* EOF ;

swallow_to_semi
    : ~( ’;’ )+
    ;

smo_statement_plus_semi : smo_statement ’;’ ;
smo_statement
  :DROP_TABLE ID        # droptable
  |CREATE_TABLE ID (bracketlist)?  # createtable
  |RENAME_TABLE ID INTO ID    # renametable
  |COPY_TABLE ID INTO ID    # copytable
  |MERGE_TABLE ID COMMA ID INTO ID   # mergetable
  |PARTITION_TABLE ID INTO ID COMMA ID WHERE swallow_to_semi  # partitiontable
  |DECOMPOSE_TABLE ID INTO bracketlist COMMA ID bracketlist COMMA ID bracketlist # decomposetab
  |JOIN_TABLE ID COMMA ID INTO ID WHERE swallow_to_semi   #jointable
  |ADD_COLUMN ID (AS expr)? INTO ID  #addcolumn
  |DROP_COLUMN ID FROM ID # dropcolumn
  |RENAME_COLUMN ID IN ID TO ID #renamecolumn
  |COPY_COLUMN ID FROM ID INTO ID (WHERE swallow_to_semi)? #copycolumn
  |NOP #noop
  ;

columnlist: (ID) (COMMA ID)*;
bracketlist:’(’ columnlist ’)’;
paramlist:’()’ | bracketlist;
```

```
expr : function | STRING_LITERAL | NULL;
function: ID paramlist;

STRING_LITERAL: ’"’ (~(’"’ | ’\r’ | ’\n’) | ’"’ ’"’ | NEWLINE)*  ’"’;

DROP_TABLE: ’DROP TABLE’;
CREATE_TABLE: ’CREATE TABLE’;
RENAME_TABLE: ’RENAME TABLE’;
COPY_TABLE: ’COPY TABLE’;
MERGE_TABLE: ’MERGE TABLE’;
PARTITION_TABLE: ’PARTITION TABLE’;
DECOMPOSE_TABLE: ’DECOMPOSE TABLE’;
JOIN_TABLE: ’JOIN TABLE’;
ADD_COLUMN: ’ADD COLUMN’;
DROP_COLUMN: ’DROP COLUMN’;
RENAME_COLUMN: ’RENAME COLUMN’;
COPY_COLUMN: ’COPY COLUMN’;
NOP: ’NOP’;


AS: ’AS’;
INTO: ’INTO’;
NULL: ’null’;
FROM: ’FROM’;
IN: ’IN’;
TO: ’TO’;
COMMA: ’,’;
WHERE: ’WHERE’;

ID
    : (SIMPLE_LETTER) (SIMPLE_LETTER | ’$’ | ’_’ | ’#’ | (’0’..’9’))*;

SIMPLE_LETTER
    : ’a’..’z’
    | ’A’..’Z’
    ;
LINE_COMMENT
    :   ’//’ ~[\r\n]* -> skip
    ;
SPACES
    : [ \t\n\r]+ -> skip
    ;
ANYCHAR : (~[\r\n]);
NEWLINE : [\r\n]+ ;
```

# B SMOs for Major Schema Change in Wikimedia

```
Smo V(41,42) := {

    COPY TABLE cur INTO cur_page;
    DROP COLUMN cur_text FROM cur_page;
    DROP COLUMN cur_comment FROM cur_page;
    DROP COLUMN cur_user FROM cur_page;
    DROP COLUMN cur_user_text FROM cur_page;
    DROP COLUMN cur_timestamp FROM cur_page;
    DROP COLUMN inverse_timestamp FROM cur_page;
    RENAME COLUMN cur_namespace IN cur_page TO page_namespace;
    RENAME COLUMN cur_title IN cur_page TO page_title;
    RENAME COLUMN cur_restrictions IN cur_page TO page_restrictions;
    RENAME COLUMN cur_counter IN cur_page TO page_counter;
    RENAME COLUMN cur_is_redirect IN cur_page TO page_is_redirect;
    RENAME COLUMN cur_is_new IN cur_page TO page_is_new;
    RENAME COLUMN cur_random IN cur_page TO page_random;
    RENAME COLUMN cur_touched IN cur_page TO page_touched;
    RENAME COLUMN cur_id IN cur_page TO page_latest;
    ADD COLUMN page_id AS function41a(page_namespace, page_title) INTO cur_page;

    COPY TABLE cur INTO cur_revision;
    DROP COLUMN cur_namespace FROM cur_revision;
    DROP COLUMN cur_title FROM cur_revision;
    DROP COLUMN cur_text FROM cur_revision;
    DROP COLUMN cur_restrictions FROM cur_revision;
    DROP COLUMN cur_counter FROM cur_revision;
    DROP COLUMN cur_is_redirect FROM cur_revision;
    DROP COLUMN cur_is_new FROM cur_revision;
    DROP COLUMN cur_random FROM cur_revision;
    DROP COLUMN cur_touched FROM cur_revision;
    RENAME COLUMN cur_id IN cur_revision TO rev_id;
    ADD COLUMN rev_page AS function41b(rev_id) INTO cur_revision;
    RENAME COLUMN cur_comment IN cur_revision TO rev_comment;
    RENAME COLUMN cur_user IN cur_revision TO rev_user;
    RENAME COLUMN cur_user_text IN cur_revision TO rev_user_text;
    RENAME COLUMN cur_timestamp IN cur_revision TO rev_timestamp;
    RENAME COLUMN cur_minor_edit IN cur_revision TO rev_minor_edit;

    RENAME TABLE cur INTO cur_text;
    DROP COLUMN cur_namespace FROM cur_text;
    DROP COLUMN cur_title FROM cur_text;
    DROP COLUMN cur_comment FROM cur_text;
    DROP COLUMN cur_user FROM cur_text;
    DROP COLUMN cur_user_text FROM cur_text;
    DROP COLUMN cur_timestamp FROM cur_text;
    DROP COLUMN cur_restrictions FROM cur_text;
    DROP COLUMN cur_counter FROM cur_text;
    DROP COLUMN cur_is_redirect FROM cur_text;
    DROP COLUMN cur_minor_edit FROM cur_text;
    DROP COLUMN cur_is_new FROM cur_text;
    DROP COLUMN cur_random FROM cur_text;
    DROP COLUMN cur_touched FROM cur_text;
    DROP COLUMN inverse_timestamp FROM cur_text;
    RENAME COLUMN cur_id IN cur_text TO old_id;
    RENAME COLUMN cur_text IN cur_text TO old_text;
    ADD COLUMN old_flags AS "" INTO cur_text;
    DROP COLUMN cur_minor_edit FROM cur_page;

    COPY TABLE old INTO old_page;
    DROP COLUMN old_text FROM old_page;
    DROP COLUMN old_comment FROM old_page;
    DROP COLUMN old_user FROM old_page;
    DROP COLUMN old_user_text FROM old_page;
    DROP COLUMN old_timestamp FROM old_page;
    DROP COLUMN inverse_timestamp FROM old_page;
    DROP COLUMN old_minor_edit FROM old_page;
    DROP COLUMN old_id FROM old_page;
    DROP COLUMN old_flags FROM old_page;
    RENAME COLUMN old_namespace IN old_page TO page_namespace;
    RENAME COLUMN old_title IN old_page TO page_title;
```

```
    ADD COLUMN page_id AS function41c(page_namespace, page_title) INTO old_page;
    ADD COLUMN page_restrictions INTO old_page;
    ADD COLUMN page_counter INTO old_page;
    ADD COLUMN page_is_redirect INTO old_page;
    ADD COLUMN page_is_new INTO old_page;
    ADD COLUMN page_touched INTO old_page;
    ADD COLUMN page_latest INTO old_page;
    ADD COLUMN page_random AS random(page_id) INTO old_page;


    COPY TABLE old INTO old_revision;
    ADD COLUMN rev_page AS function41d(old_namespace,old_title) INTO old_revision;
    DROP COLUMN old_namespace FROM old_revision;
    DROP COLUMN old_title FROM old_revision;
    DROP COLUMN old_text FROM old_revision;
    DROP COLUMN old_flags FROM old_revision;
    RENAME COLUMN old_id IN old_revision TO rev_id;
    RENAME COLUMN old_comment IN old_revision TO rev_comment;
    RENAME COLUMN old_user IN old_revision TO rev_user;
    RENAME COLUMN old_user_text IN old_revision TO rev_user_text;
    RENAME COLUMN old_timestamp IN old_revision TO rev_timestamp;
    RENAME COLUMN old_minor_edit IN old_revision TO rev_minor_edit;

    RENAME TABLE old INTO old_text;
    DROP COLUMN old_namespace FROM old_text;
    DROP COLUMN old_title FROM old_text;
    DROP COLUMN old_comment FROM old_text;
    DROP COLUMN old_user FROM old_text;
    DROP COLUMN old_user_text FROM old_text;
    DROP COLUMN old_timestamp FROM old_text;
    DROP COLUMN inverse_timestamp FROM old_text;

    MERGE TABLE cur_text, old_text  INTO text;
    MERGE TABLE cur_page, old_page  INTO page;
    MERGE TABLE cur_revision, old_revision  INTO revision;
};
```

# Bibliography

[1] *Amazon Aurora Under the Hood: Fast DDL.* https://aws.amazon.com/blogs/database/amazon-aurora-under-the-hood-fast-ddl/.

[2] *Antlr Parser Generator.* http://www.antlr.org/.

[3] Marcelo Arenas, Jorge Pérez, and Cristian Riveros. "The Recovery of a Schema Mapping: Bringing Exchanged Data Back". In: *ACM Trans. Database Syst.* 34.4 (Dec. 2009), 22:1–22:48. ISSN: 0362-5915. DOI: 10.1145/1620585.1620589. URL: http://doi.acm.org/10.1145/1620585.1620589.

[4] Jay Banerjee et al. "Semantics and Implementation of Schema Evolution in Object-oriented Databases". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data.* SIGMOD '87. San Francisco, California, USA: ACM, 1987, pp. 311–322. ISBN: 0-89791-236-5. DOI: 10.1145/38713.38748. URL: http://doi.acm.org/10.1145/38713.38748.

[5] Mark Callaghan. *Online Schema Change for MySQL.* https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/.

[6] Donald D Chamberlin et al. "A history and evaluation of System R". In: *Communications of the ACM* 24.10 (1981), pp. 632–646.

[7] James C Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[8] Carlo A Curino et al. "Schema evolution in wikipedia: toward a web information system benchmark". In: *In International Conference on Enterprise Information Systems (ICEIS.* 2008.

[9] Carlo Curino et al. "Automating the database schema evolution process". In: *The International Journal on Very Large Data Bases* 22.1 (2013), pp. 73–98.

[10] Hector Garcia-Molina and Kenneth Salem. “Sagas”. In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’87. San Francisco, California, USA: ACM, 1987, pp. 249–259. ISBN: 0-89791-236-5. DOI: `10.1145/38713.38742`. URL: `http://doi.acm.org/10.1145/38713.38742`.

[11] J. N. Gray et al. “Readings in Database Systems (2Nd Ed.)” In: ed. by Michael Stonebraker. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. Chap. Granularity of Locks and Degrees of Consistency in a Shared Data Base, pp. 181–208. ISBN: 1-55860-252-6. URL: `http://dl.acm.org/citation.cfm?id=190956.190979`.

[12] Timothy Griffin and Leonid Libkin. “Incremental maintenance of views with duplicates”. In: *ACM SIGMOD Record*. Vol. 24. 2. ACM. 1995, pp. 328–339.

[13] *gRPC open-source universal RPC framework*. `https://grpc.io/`.

[14] Ashish Gupta, Inderpal Singh Mumick, et al. “Maintenance of materialized views: Problems, techniques, and applications”. In: *IEEE Data Eng. Bull.* 18.2 (1995), pp. 3–18.

[15] Ziyang Liu et al. “Efficient and scalable data evolution with column oriented databases”. In: *Proceedings of the 14th International Conference on Extending Database Technology*. ACM. 2011, pp. 105–116.

[16] David T McWherter et al. “Priority mechanisms for OLTP and transactional web applications”. In: *Data Engineering, 2004. Proceedings. 20th International Conference on*. IEEE. 2004, pp. 535–546.

[17] *MediaWiki Code Repository*. `https://phabricator.wikimedia.org/source/mediawiki/repository/master/`.

[18] *Overview of Vitess*. `http://vitess.io/`.

[19] *Performing Online Application Upgrade Using the Edition-Based Redefinition Feature*. `http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/11g/r2/prod/appdev/ebr/ebr_otn.htm`.

[20] Young-Gook Ra and E. A. Rundensteiner. “A transparent schema-evolution system based on object-oriented view technology”. In: *IEEE Transactions on Knowledge and Data Engineering* 9.4 (July 1997), pp. 600–624. ISSN: 1041-4347. DOI: `10.1109/69.617053`.

[21] Dave Brondsema Rick Copeland Mark Ramm and Jonathan Beard. *Model Evolution and Migrations*. `http://ming.readthedocs.io/en/latest/migrations.html`.

[22] Craig Ringer. *Priorities*. `https://wiki.postgresql.org/wiki/Priorities`.

[23] John F Roddick. "A survey of schema versioning issues for database systems". In: *Information and Software Technology* 37.7 (1995), pp. 383–393.

[24] John F. Roddick. "A survey of schema versioning issues for database systems". In: *Information and Software Technology* 37 (1995), pp. 383–393.

[25] Stefanie Scherzinger, Meike Klettke, and Uta Störl. *Managing Schema Evolution in NoSQL Data Stores*. Aug. 2013.

[26] Michael Stonebraker et al. "The Design and Implementation of INGRES". In: *ACM Trans. Database Syst.* 1.3 (Sept. 1976), pp. 189–222. ISSN: 0362-5915. DOI: `10.1145/320473.320476`. URL: `http://doi.acm.org/10.1145/320473.320476`.

[27] *Twitter - Merv Adrian*. `https://twitter.com/merv/status/667048388958011392`.