FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Datenzitierbarkeit bei Schemaevolution in relationalen Datenbanken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

BSc. Patrick Säuerl
Matrikelnummer 1125492

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber, Univ. Doz.

Wien, 25. Februar 2018

Patrick Säuerl   Andreas Rauber

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Data Citation under Schema Evolution in RDBMS

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

BSc. Patrick Säuerl
Registration Number 1125492

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber, Univ. Doz.

Vienna, 25th February, 2018

Patrick Säuerl

Andreas Rauber

Erklärung zur Verfassung der Arbeit

BSc. Patrick Säuerl
Address

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Februar 2018

Patrick Säuerl

Danksagung

Ich möchte mich bei meiner Mutter, meinem Vater, meiner Oma und meiner Großtante für ihre Unterstützung bedanken. Jeder von euch half mir auf die für ihn bestmöglichste Art und Weise, Danke!

Besonderer Dank geht auch an meine Freundin Angie. Die vielen hin und hers im Masterstudium sowie bei dieser Arbeit waren nicht leicht, danke für deine Hilfe, Unterstützung und vorallem Verständnis.

Ebenfalls danke an meine "Uni-Gruppe", bestehend aus Thomas, Lisa, Markus und Manuel. Ohne euch hätte das ganze wohl länger gedauert.

Auch ein großes Danke an Manfred, mit dem ich seit der 1. HTL immer über Programmieren reden und Musik machen konnte. Danke fürs Dasein und, dass du meinen Blödsinn aushaltest. Du bekommst dein Nutella bald zurück!

Ebenfalls ein Danke an Werner und Andreas, die Aufgabenstellungen sowie Entwicklungsmöglichkeiten von euch, vorallem am Anfang des Studiums, haben mir sehr geholfen.

Zu guter letzt noch ein Danke an Prof. Rauber für das spannende Thema, die gute Betreuung und die bohrenden Fragen, wodurch ich meine Lösungsansäetze verbessern musste.

Acknowledgements

I want to thank my mother, my father, my grandmother and my great-grandaunt for their support. Everyone helped me in their best possible ways.

Also, i want to especially thank my girlfriend Angie. The many ups and downs during the master's program, and during this thesis, have not been easy. Thank you for your help, your support and your understanding.

Next in my list is my "Uni-Gruppe", consisting of Thomas, Lisa, Markus and Manuel. Everything would probably have taken longer without you, thank you all.

A big thanks also goes to Manfred, with whom i have been talking about programming and making music since the 1st grade of HTL, just for being there and enduring my shenanigans. You'll get your Nutella back soon.

I also want to thank Werner and Andreas for the tasks and opportunities to grow they provided, especially in the beginning of my studies.

Last but not least, i want to thank Prof. Rauber for the very interesting diploma thesis topic, the very good supervision and the challenging questions which made me improve my approaches.

Kurzfassung

Viele Bereiche in der Forschung und Wirtschaft benötigen Daten als Grundlage für Experimente und Analysen. Die Reproduzierbarkeit dieser Experimente ist ein wichtiger Teil in der Wissenschaft und kann auch in der Wirtschaft eine große Rolle spielen aufgrund rechtlicher Natur oder Rechenschaftspflicht bei Entscheidungen. Generell muss man davon ausgehen, dass die zugrunde liegenden Daten für diese Experimente, ebenfalls auch das den Daten zugrunde liegende Schema, Änderungen unterliegen. Dies macht die korrekte Reproduktion von Daten eine nicht triviale Aufgabe.

Wege, Daten die bereits abgefragt wurden korrekt zu reproduzieren, sind bereits entwickelt wurden und sind bekannt als Datenzitierbarkeit. Der Prozess der Schema Änderung und das Migrieren der Abfragen zwischen verschiedenen Schema Versionen ist bekannt als Schema Evolution.

Systeme für diese beiden Bereiche wurden bereits entwickelt, jedoch kein System, welches beides unterstützt. Diese Diplomarbeit kombiniert Systeme aus den beiden Bereichen und beantwortet, welche Design-Entscheidungen man bei der Kombination treffen muss und wie diese die Abfragezeit und Speicherplatzanforderungen beeinflussen. Zusätzlich werden noch unterschiedliche Ansätze für die Daten Historisierung implementiert, welche hinsichtlich ihrer Speicherplatzanforderungen sowie der Zeit für Abfragen evaluiert werden. Darauf aufbauend erstellen wir Richtlinien, welche Ansätze in welchen Situationen verwendet werden sollen und beschreiben verschiedene Bereiche für Optimierungen.

Die wichtigsten Beiträge dieser Diplomarbeit sind die entwickelten Locking Strategien, die sich aus der Kombination den Anforderungen für Datenzitierungssystemen und Schema Evolution ergeben, sowie das Umschreiben von Abfragen für die implementierten Datenhistorisiserungsansätze, ein skizziertes System für Datenbanken mit Millionen von Einträgen, und Verwendungsrichtlinien für diese Systeme.

Abstract

Many areas in science and businesses rely heavily on data for their experiments and analyses. Reproducibility of those experiments is an important part of science and can be important in businesses too, due to legal or accountability issues. In general, the underlying data for those experiments and the schema it is stored in, are subject to change, making correct dataset reproduction a non trivial task.

Ways to correctly reproduce a once created set of data have been developed and those ways are generally known as making data citeable. The process of changing the schema and migrating queries between different schemata versions is known as schema evolution.

Systems that solve data citation, as well as schema evolvable systems for relational database management systems (RDBMS), have been designed, but none for both. This thesis is going to answer how those systems can be combined, which design decisions have to be taken when combining those systems and how they impact query performance and storage size.

To provide these answers, we first take a look at systems that solve data citation and systems that solve schema evolution in RBDMS. We design and implement a system based on existing solutions with different data historization approaches. Afterwards, we evaluate those approaches with respect to their storage requirements and query performance. Based on our evaluation, we conclude usage guidelines for different scenarios and discuss potential optimizations.

The main contributions of this thesis are the developed locking mechanisms that result from combining requirements of schema evolvable systems with those of data citeable systems, as well as the query rewriting techniques for the different implemented data historization approaches, an outlined additional outlined system for databases with millions of rows, and the usage guidelines inferred from our evaluation results.

Contents

CHAPTER 1

Introduction

Data driven experiments and analyses build the foundation for many areas in science and businesses. At one point, an experiment, may it be some statistics that are created or maybe some AI that is trained, are done for the first time. The results of those experiment are published, and someone wants to redo this experiment to verify the correctness of the results. This person now needs access to the original data used for the experiment. The same scenario could happen in the business world. One member of the business intelligence department creates a statistic, upon which business decisions are taken. Assume those decisions go wrong and the company wants to know why. Has the person making the decisions of the statistic made a mistake, or was the statistic flawed? To answer this, the original data used for the statistic needs to be reproduced. However, the nature of data is, that it changes over time. Old data may be deleted or updated and new data is inserted. To make things worse, not only may the data be dynamic, the schema in which it is stored in may change too. In the case of relational database management systems (RBDMS), tables and columns may be added, dropped or renamed and tables may also be split or merged. The dynamic nature of the data, combined with a dynamic schema, makes the process of reconstructing a once retrieved dataset a non trivial task.

A current solution done by many is to use a static dataset that is not subject to any data change. This avoids many problems but it is not suitable for many situations. Usually, researchers are interested in state of the art methods and processes, and having up to date data is crucial for the importance of their work. Also, in the realm of business, most decisions depend on up to date data. Therefore, having the possibility to correctly retrieve and recreate current data is crucial for them.

Providers of data now face the issue, that they have to provide an interface for their data that retrieves data and can also reconstruct already retrieved data. A naive approach is, to store all retrieved datasubsets in a separate database. The size of this database would grow incredibly fast and is therefore not a viable approach. Another approach is to store the issued query and designing the data storage system in a way, that it enables the re-execution of already issued queries. The Working Group on Data Citation (WGDC)[1] created recommendations for such query-based system that enable correct and verified data subset reconstruction, effectively making data citeable The recommendations consist of 14 requirements a system has to fulfill to support data citation and implementation approaches are described in: Data Citation of Evolving Data[RAvUP15]. The core of those recommendations is, that the systems needs to version data via timestamping, ensure that queries uses a stable sorting, that the result set can be verified and that data is not altered during the query execution process. They, on a high level, also touch the aspect of technology migration, which happens when the database is migrated to a new database systems or the underlying data schema is changed. If such a change occurs, they recommend to rewrite the queries to target the new schema and they need to be verified to still work correctly.

The act of technology migration by changing the underlying schema may sound like a very rare occurring event at the first look. However, scientific databases like the CERN-DQ2 had 51 different versions in 1.3 years and business oriented applications like the CRM system DekiWiki had 16 schema versions in 4.2 years[CMDZ13]. Based on this data, we can safely conclude, that technology migration is a regularly occurring task and is an important factor for relevance of the database.

Current solutions for data citation assume, that the schema is rather static and schema evolution is a special process that usually requires a lot of manual work with migrating queries. Solutions that solve schema migration usually support query migration and data timestamping, but they are not designed with the recommendations of data citation in mind. They usually lack the core recommendations on queries we identified and can therefore not guarantee correct query result reconstruction.

It is now open to answer, how to build a system that enables data citation and enables schema evolution in a way, that it can be seen as a regular task. The recommendations give us a good overview and guideline one the properties a system needs to have to support data citation, and therefore build the foundation on how to reconstruct already extracted data. However, it is left open on how those recommendations can be implemented. What approaches does one have on how to historize data? Should data be split up by attributes or should complete tuples be historized, and where should the historized data be stored? What is the influence of those decisions on the query execution time and storage size of the database? For schema evolution, we are interested in how the data schema can evolve and how the schema evolution impacts the database. Is it required to take down the database during the schema evolution process to guarantee that no queries are issued that cannot be reconstructed? How is historized data handled during the schema evolution and how to migrate historic queries between different schemata?

To answer these questions, we explore the design decisions we have to take when designing an data citeable, schema evolvable system for RDBMS. The design decisions are based

[1]`https://www.rd-alliance.org/groups/data-citation-wg.html`

on the recommendations for Data Citation[RAvUP15], different approaches for data historization[PR13] and schema modification operators[CMDZ13]. After having identified those options, we choose certain designs, and based of them, we infer strategies to rewrite queries that target different schemata. Mechanisms that ensure the data integrity are designed as well. Based on the decisions and defined strategies, we implement a working solution that allows us to choose different data historization approaches. The implemented system is then evaluated for the impact on storage size and query performance. We also discuss a separate system design that is more suited for big databases and focuses on reduced storage requirements.

This work is structured as follows. In Chapter 2, we cover related work in the fields of Data Citation and Schema Evolution to get an understanding of the problems we have to solve. Existing systems for those areas are also discussed and based on our understanding for the problem we need to solve, we describe our system design in Chapter 3. All design decisions that have been taken are discussed there, as well as implementation details. We follow with our evaluation in Chapter 4 and end with our conclusions in Chapter 5.

CHAPTER 2

Related Work

In the following sections we cover the related work in the two subjects that build the foundation for our system design.

2.1 Data Citation

Many areas in science and businesses rely heavily on data for their experiments and analysis, especially areas that make use of statistics and AI, as frequently encountered in Business Intelligence tasks. The underlying data for those experiments is subject to change, new data is added, old one is deleted or current one is updated. Due to this dynamic nature of data, it is not possible in general to retrieve identical data again at a later point in time, unless special measures are taken. Without the ability to retrieve the same data again, it is not possible to redo the experiments and analyses on them, thus making the results not reproducible. Apart from being a major problem in science, this also constitutes a significant legal/accountability challenge in business settings, and also limits traceability of erroneous decisions.

To correctly use data in those experiments, the data used has to be precisely identified and re-created. Therefore the retrieved data set needs an identification by which it can be reconstructed again from the source system. This is part of a concept named Data Citation.

A naive solution for data citation is storing once retrieved data again and then just retrieving this stored data. The drawback is the big amount of storage required for this, making it infeasible for many applications, as well as the associated complexity in data management. Therefore, we have to find a solution which can reconstruct the data only based on the query and auxiliary information.

The Working Group on Data Citation (WGDC)[1], put out 14 recommendations that, when

[1]`https://www.rd-alliance.org/groups/data-citation-wg.html`

implemented, allow a system to reconstruct already extracted data based on the issued query. Those recommendations have been thoroughly discussed in: Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use[RAvP16] and implemented in: Scalable data citation in dynamic, large databases: Model and reference implementation [PR13]. We will now take a look at the recommended steps and afterwards we discuss the reference implementation.

The WGDC[RAvP16] recommends to implement the following steps to make data citeable:

- Preparing the Data and the Query Store
  - R1 - Data Versioning
  - R2 - Timestamping
  - R3 - Query Store Facilities
- Persistently Identifying Specific Data Sets
  - R4 - Query Uniqueness
  - R5 - Stable Sorting
  - R6 - Result Set Verification
  - R7 - Query Timestamping
  - R8 - Query PID
  - R9 - Store the Query
  - R10 - Automated Citation Texts
- Resolving PIDs and Retrieving the Data
  - R11 - Landing Page
  - R12 - Machine Actionability
- Upon modifications to the Data Infrastructure
  - R13 - Technology Migration
  - R14 - Migration Verification

Recommendation R1 specifies, that all accessed data needs to be stored in versions, in a way, that enables easy access to previous versions. Change logs or rollbacks are basically a versioning but they are not flexible enough to support retrieving states of data at a given time. In the realm of databases, approaches to support versioning include history tables, direct integration of historization in the live database tables, or hybrid solutions.

Recommendation R2 specifies, that operations on data (create, update, delete) have to be timestamped. In the realm of databases, this means that all INSERT, DELETE, UPDATE statements assigned a timestamp to the tuples/columns they effect.

Recommendation R3 specifies, that a facility need to be set up that handles the storage of executed queries and the metadata for them. The metadata that needs to be stored emerges from the recommendations R4-R10. This metadata includes the original issued query, a potentially re-written query by the system, hash of the query, hash of the result set, the execution timestamp, a persistent identifier of the data store as well as the query, and other metadata required by the landing page (R11).

Recommendation R4 specifies, that queries have to be re-written to a normalized form to detect identical queries via comparing the checksum of the normalized forms. This enables that sementically identical subsets can be identified by the same persistent identifier (PID).

Recommendation R5 specifies, that the sorting of the returned records is unambiguous and reproducible, as the ordering of records can influence experiment outcomes. Databases may return data in arbitrary order if no sorting criteria is specified. Therefore, in the realm of databases, an ORDER BY clause needs to be added that ensures the unambiguous and reproducible sorting.

Recommendation R6 specifies, that checksum of the query result needs to be calculated in order to verify the correctness of the result upon re-execution.

Recommendation R7 specifies, that all queries need a timestamp assigned based on the last update of the database or the subset they are targeting. The current active timestamp could also be used but this may lead to privacy issues as it reveals the moment a query has been issued by the user.

Recommendation R8 specifies, that a query is assigned a new PID if either the query is new, or if the result set returned from an earlier identical query produced not the same hash. This allows to detect, that the same query issued at different times resulted in the same data and therefore they refer to the same data-subset. If they refer to the same data-subset, they should have the same PID, otherwise their PID should differ.

Recommendation R9 specifies, that the query and all the metadata for it, is stored in the query store. It is also required, that a transaction concept needs to be in place that ensures that: the underlying data is not changed during query execution and storage, ensuring isolation; the process of executing the query and storing it is handled at once, ensuring atomicity.

Recommendation R10 specifies, that an automatic citation text needs to be generated that contains the PID identifying the retrieved subset, allowing the users to easily cite and share the data.

Recommendation R11 specifies, that the PIDs should resolve to a human readable landing page that provides the data and metadata.

Recommendation R12 specifies, that an API should exist to access the metadata and data via query re-execution.

Recommendation R13 specifies, that the queries need to be rewritten and the fixity information recalculated, when the data is migrated. Data migration occurs when the used database management system is changed, the schema is changed or a completely different technology used in the system changes. The queries that worked in the old system have to be migrated to work in the system to ensure that the data can be reconstructed. However, if the new system cannot reproduce the exact hashes, new fixity information has to be calculated and the landing page needs to redirect accordingly.

Recommendation R14 specifies, that after data and query migrations, the queries have to be verified in order to ensure that they can be re-executed correctly.

Based on a closer look on the recommendations, we see, that a system that implements them is suitable to reconstruct already extracted data, thus allowing the data to be citeable.

Reference Implementation The reference implementation [PR13] is a data citeable system for RDBMS. Timestamping is applied there on tuple level, meaning every row in the database has a *starttimestamp* and an *endtimestamp*. They described that they could use three different historization models for tuples: integrated, separated and a hybrid model; and went with the separated model. There, each table has a table that has the same schema as the original table, but contains the *starttimestamp* and the *endtimestamp*. The original table only contains current active rows. They stated, that the evolution of the database schema is an open question that needs to be tackled. The different historization approaches have been described on a high level, but they have not been evaluated against each other. The questions raised by this reference implementation are the main areas tackled in this diploma thesis.

2.2 Schema Evolution

Schema Evolution is the process of changing the schema inside a database, in our case, a RDBMS. This usually consists of two steps. First, the underlying schema needs to be changed, usually through DDL (data definition language) statements. Afterwards, queries that were working against the old schema need to be migrated to work against the new schema.

This long standing challenge has been elegantly solved by the system described in: Automating the database schema evolution process [CMDZ13]. They base the schema evolution on a schema evolution language consisting of schema modification operators (SMOs), covering the modifications of tables and their columns, and integrity constraint modification operators (ICMOs), covering the modifications of primary key, foreign key and value constraints. These operators have been designed after analyzing the schema changes that occurred in systems like MediaWiki, the software behind Wikipedia, and Joomla, a content management system, among others. By expressing the schema evolution with those operators, they are able to map queries issued against one schema into equivalent queries in another schema. This systems is, to the best of our knowledge, the most advanced and complete solution to schema evolution. This system was described here was not intended for data historization, as the authors see the main research goal in database archival and historical queries solved in their previous work [MCD+08], which will be discussed in the next Section 2.3.

We base the schema evolution steps in our system on those SMOs, which are defined as:

An SMO is defined as a function that receives as input a relational schema and the underlying database and produces as out-put a (modified) version of the input schema and a migrated version of the database.[CMDZ13]

The SMOs used in the described system, as well as ours, are:

- CREATE TABLE R(a,b,c)
- DROP TABLE R
- RENAME TABLE R into T
- MERGE TABLE R, S into T
- PARTITION TABLE R into S with *cond*, T
- DECOMPOSE TABLE R into S(a,b), T(a,c)
- JOIN TABLE R,S INTO T WHERE cond
- ADD COLUMN d [AS const | func(a,b,c)] INTO R
- DROP COLUMN c from R
- RENAME COLUMN b IN R TO d

The main idea of most SMOs should be clear from their definition , so we focus on a few more subtle behaviors. All table level SMOs consume their input tables. For example, the resulting schema of a MERGE TABLE SMO contains the table T, but not R and S, as they are consumed. PARTITION TABLE horizontally partitions the table into S and T, where the tuples moving to S fulfill the given condition. The other tuples are migrated to T. The remaining SMOs will be explained in Section 3.4.

In the system we develop, we ensure that data is preserved during SMO execution in a way that we can easily rewrite our historic queries to reproduce the same result.

2.3 Related Systems

We take a look on other state of the art systems that could be used to solve data citation but have certain shortcomings.

In Managing and Querying Transaction-time Databases under Schema Evolution [MCD$^+$08], a system called PRIMA, is designed that could be well extended to support data citation. The history of the database is represented in an XML, which contains a unified view of how the schema of the database evolved, as well as the data in it. The schema of the database can evolve via 10 different schema modification operators (SMOs). Updates to the current relational database via create, update and delete statements, as well as SMOs, are propagated to the XML database. A version at a given time of the database is represented in the XML database as the following node hierarchy: database/table/row/-columns. All elements there have a start and endtimestamp. If the value of a columns is updated, only the node containing the current value needs its endtimestamp set and a new node containing the new value can be inserted. SMOs introduce a new database version in the XML database and current values are only migrated to the new schema if necessary. SQL queries can be issued against any schema version and are migrated to the current version, and afterwards they are migrated to an equivalent XQuery that operates on the historic data. On average, rewritten queries run 4.5 times slower. They saw the performance limited by the XQuery engine and proposed to use a RDBMS-backed storage for query execution. The system they designed here could probably be extended to support data citation, although the transaction mechanism required by recommendation R9 could be hard to implement as the XML database needs to be locked in order with the actual RDBMS.

A quite interesting system is AIMS (Archival Information Management System)[MCZ10]. It is the successor of the previous described system and they tackled the following points: (i) the complexity of rewriting temporal XQuery statements and (ii) the lack of reliable techniques for optimizing these queries. Their system aims to achieve perfect archival quality. Therefore, they have to support a wider range of queries, like "Find employees who worked as an assistant staff for one year or more". Data citeable systems do not have to support those queries, they only need to reproduce queries executed in the past. Their system stores data historized on attribute level and supports schema evolution via schema modification operators. Their systems is implemented as an extension of the MySQL master/slave replication technology. They enabled one slave to work as the history-enabled slave that observes the MySQL binary log, which leads to low overhead on the actual database. The system could be extended by a querystore and be used as the backbone for a data-citeable system. However, it would be required to check how the atomicity required by R9 can be implemented. As this system is basically a distributed system with a master-slave topology, introducing a distributed atomic operation is usually a non trivial task that impacts the whole systems performance. Sadly, the project seems to be dead as no follow up papers have been presented since 2010, the code for the system is not available and the wiki page for the system was last updated on December 6th 2010[2]. Nevertheless, the system we see here is the blueprint for efficient attribute-level-timestamped data historizaiton in RDBMS.

Another rather old system is designed in: A Model for Schema Versioning in Temporal

[2] http://yellowstone.cs.ucla.edu/schema-evolution/index.php/AIMS

Database Systems[Rod95], nevertheless it can be seen as the basis for the systems we have seen so far and which we will implement. Note, that his system has only be designed but, to the best of our knowledge, never implemented. The system designed here versions data via tuple-level timestamping with a start and an endtimestamp. It can evolve through various schema modifications, that basically achieve the same as the SMOs we have introduced. Retrieving different versions of the data is based on the concept of a non-temporal completed relation scheme C. This scheme C contains the minimal union of all explicit attributes that have been defined during the relation and all attributes are considered to be able to hold every data of every domain (effectively ignoring column datatypes). A view function V is defined that maps C to the subset of columns defined at a given time. Another function W is defined that maps columns of a given column of a given schema version back to C. The whole system designed there can be seen as a blueprint for a schema evolvable, data historization system using the so called integrated historization approach, which we will see later. However, it was never implemented and does not consider column types (which is important in real-life applications). Also, the system was not designed with the atomicity and isolation criteria in mind that are needed for a data citeable system.

2.4 Summary

We have learned about the related work in the fields of Data Citation and Schema Evolution. The recommendations for implementing a data citeable system have been studied, as well as related systems that solve data citation and the reasons why they are not useable for our scenario. Schema Evolution in RDBMS, based on Schema Evolution Operators (SMOs) has been found suitable as basis for our implementation. We can now start to design our system based on the data citation recommendations and SMOs.

CHAPTER 3

System Design

We describe our system design from a top-down perspective, starting with a system overview, which gives us a thorough understanding of our components and how they behave. Afterwards, we diver deeper and discuss specific design decisions and how some problems of our system have been solved. General optimization ideas are discussed as well as strategies for special scenarios, like petabyte databases. We end with a short look on code metrics from our implemented system and the summary.

3.1 System Overview

The aim of this section is to give an overview on the architecture and the behaviour of the developed API that serves as the basis for our evaluation. Many design decisions need a more in depth explanation, which are covered in the following subsections.

Our API is designed as a middleware that handles the execution of CRUD (Create, Read, Update, Delete) statements on data, SMOs, and acts as an interface for the querystore functionality, which is the execution of queries that are stored for re-execution, and the re-execution of them in the future. The API is implemented in C#, targeting the .Net Framwork V4.6 and uses MySQL as the RBDMS. It is implemented to be easily portable to the .Net Core Framework and to target other RBDMS.

We explore our system through an example which touches every aspect we need to cover to get an understanding on how the API works. First, we start from an empty database and choose which historization approach we want to use, for the example we go with the so-called "separated approach". In the separated approach, we have a history table for every actual table, storing the rows of the original table augmented by a starting timestamp and ending timestamp. This allows us to query the original table in a way that only rows are returned that were active at a given time and is named Tuple Based Timestamping.

3.1.1 Setup and Initialization

Our system initializes itself, creating a table to hold schema information, called *schematable* and one that holds information about queries stored for re-execution, called *querystore*. The *schematable* consists of the following columns:

- ID
- Schema - a serialization of the schema we store
- SMO - storing which SMO leads to this schema
- Timestamp - documents when this schema was created

The schema we store is an object structure from our C# API serialized as a JSON string. It consists of a list of tables. Each table consists of its name and schema, the definition of the columns in the table and a GUID which unambiguously defines the table. Additionally, we store the name and schema of the table that will be used to store the historized rows of the table and the name of the supporting table which stores: a timstamping that represents the last update that occured to the table; a boolean flag which marks that the table is currently used in an ongoing SMO and can therefore not be queried.

The *querystore* table consists of the following columns:

- ID - an auto-incrementing integer acting as the primary key
- Query - the original query that was issued
- ReWrittenQuery -the query as it was rewritten for execution
- Timestamp - the timestamp when the query was issued
- Hash - the fixity information needed to check re-execution
- GUID - a GUID assigned to identify this query uniquely
- AdditionalInformation - here we store the GUIDs of the tables that were used during execution

The data stored in additional information could be omitted as the GUIDs could be recalculated by extracting the schema at querytime and looking up there. We decided to leave them in there to ease the implementation of our rewriting algorithms. The other columns correspond directly to the metadata fields listed by the RDA recomme dations on dynamic data citation.

3.1.2 SMO execution and supporting tables

We create a table *employees* by issuing a Create Table SMO. Besides the actual table that we create, we also create a history table *employees_hist* that stores timestamped versions of the rows of the original table. In our implementation, the suffix *_hist* is replaced by the schema version this table was introduced. The history table has the same schema as the original table, augmented by two columns *starttimestamp* and *endtimestamp.* This approach of timestamping is called Tuple Based Timestamping and explained in more detail in Section 3.2.

A second auxiliary table called *employee_metadata* is created which holds two values: *lastupdatetime*, holding the last update that occurred to the table or its rows, and *canbequeried*, a boolean flag that indicates if the table is allowed to be queried by CRUD statements. This flag is needed as MySQL permits DDL statements on locked tables and we need to ensure that no CRUD statements, especially data citation statements, are executed on tables that are concurrently targeted by SMos. The algorithm on how SMOs are processed is described as pseudocode in Algorithm 1.

Algorithm 3.1: SMO Execution

```
Data: SMO
Result: Updated Database Schema
1  Aquire lock: SMO;
2  Set current update time ;
3  Get current schema;
4  Update current schema by SMO;
5  Aquire write lock on metatables of tables used by the SMO;
6  Set can be queried flag of metatables to false;
7  Release write lock on metatables;
8  Historize data from tables consumed by the SMO ;
9  Do SMO transformation ;
10 Store new Schema;
11 Set can be queried flag of metatables to true;
12 Set last update time of metatable to update time;
13 Unlock lock: SMO;
```

3.1.3 CRUD - Handling

We created the table *employees*, the supporting table for historization, as well as the auxiliary metatable for it. Now, we attempt to insert data into the *employees* table by issuing an INSERT statement. The statement handling has to consider the following requirements to guarantee a consistent view and reproducible queries on the system:

- No re-executable SELECT statement may be issued against the table while the table data is altered
- No SMO may be in progress currently on the table
- The historization table has to be updated accordingly
- The last update time has to be set accordingly

The INSERT statement is handled as described in the pseudocode in Algorithm 2, which takes care of the requirements above. We refer to Section 3.6 for a more in depth explanation on the locking mechanisms that are used to guarantee the first two requirements.

Algorithm 3.2: INSERT statement handling

```
Data: INSERT Statement on employee table
Result: Inserted data in table
1  Acquire write lock for: employee table, employee history table, employee metatable;
2  if $metatable.canbequeried == false$ then
3      Abort insert as table is currently used by SMO;
4      Unlock tables ;
5  end
6  Set insert time;
7  Insert into employee;
8  Insert into history table with $startts$ = insert time, $endts$ = null;
9  Set metatable.lastupdate = insert time;
10 Unlock tables ;
```

UPDATE statements are handled in a similar way. Rows that will get updated have their end timestamp set in the history table. The rows in the actual table are updated and their result is added to the history table with the start timestamp being set to the update time. The locking mechanism is the same as for INSERT statements.

DELETE statements set the end timestamp in the history table and delete the rows on the actual table. The locking mechanism is the same as for INSERT statements.

A more in depth description of the handling of UPDATE, DELETE and INSERT statements can be found in Section 3.3.

SELECT statements that need not to be re-executed at a later point in time can be issued as they are.

As the INSERT, DELETE and UPDATE statements are handled in a way, that guarantees a defined state of the database at each time, we can execute queries that can be reproduced in a later point of time. How those queries are executed and re-executed is described in the next section.

3.1.4 Query Store Capabilities

The way SMOs and CRUD statements are handled leaves us with a system with the following properties: the schema is defined at each time; data can be queried with a specific point in time; the changes occur in a consistent and well behaved manner. Those properties allow us to implement a Query Store that can store queries and re-execute them at a later time producing the same result.

When we execute a re-executable SELECT statement, we consider the following recommendations to achieve a reproducibility at a later point in time:

- R5 - Stable Sorting
- R6 - Result Set Verification
- R7 - Query Timestamping
- R8 - Query PID
- R9 - Store the Query

R5 - Stable Sorting is needed as the order of the returned rows may have an impact on the experiments done with the data[RAvP16]. The stable sorting is achieved by adding an ORDER BY clause where all selected columns are added and sorted in ascending order.

R6 - Result Set Verification recommends to calculate a fixity information about the returned data which is later on used to verify the correctness of the result upon re-execution[RAvP16]. Our fixity information is calculated by generating a hash of the result set in the following way: We concatenate every column in a row, separated by #, resulting in a single string per row. Those strings are then again concatenated by a separating #, resulting in one string for the whole data set. Then we take the MD5 hash of this value which acts as our checksum for the result set. All this is done in a single statement inside the database to avoid data alterations introduced by drivers. More information about why we have chosen this approach and about the sorting can be found in the paragraph Fixity Calculation in Subsection 3.5.4.

R7 - Query Timestamping is achieved by using the current time after all locks have been acquired. We could have used the maximum of all *metatable.lastupdate* values of the meta tables of the tables involved in the SELECT statement instead, but this could result in an old schema being returned.

R8 - Query PID is achieved by storing a GUID with the query.

R9 - Store the Query is achieved by storing the following information inside the *querystore* table: $ID, Query, Timestamp, Hash, GUID, AdditionalInformation$. Those columns have been described in the Paragraph Setup and Initilazation of this Section.

The complete algorithm in pseudocode is described in Algorithm 3. It is based on the algorithm to avoid data updates during the query execution in Identification of Reproducible Subsets for Data Citation, Sharing and Re-Use[RAvP16]. We modified it by taking into account currently running SMOs (*canbequereid* flag of metatables) and by calculating the hash on the database (reproducability considerations).

We want to specially mention on how we obtain the hash of the result set. To calculate the hash, we modify the query to only return a single value acting as the hash of the returned dataset. We achieve this by replacing all columns by their hashed values, then calculating the hash of the concatenated column hashes. This results in a single hash per row, we again concatenate this and calculate the resulting hash. This modified query is executed at: $ExecuteQueryforhashcalculation$. More details on this approach are described in the Paragraph Fixity Calculation in Subsection 3.5.1.

Our algorithm may fail if we acquire the READ LOCKs on the tables but not all *canbequeried* flags are true. In this case, our algorithm aborts. Currently, we have not implemented any retry algorithm, however the API can be easily extended with such a retry algorithm, for example Fibonacci Retry [1].

[1]`https://github.com/ballance/Fibonacci-Retry`

Algorithm 3.3: Execute citeable query

```
   Data: SELECT statement
   Result: Citeable executed Statement and PID
 1 Rewrite Query to ensure stable sorting;
 2 Lock all tables and their auxiliary tables used in the SELECT statement;
 3 if $any\ metatable.canbequeried == false$ then
 4     Abort execution as not all tables can be queried;
 5     Unlock all tables;
 6 end
 7 Set UpdateTime to current time;
 8 Execute Query for hash calculation;
 9 Execute Query for actual dataset;
10 Store Query, Hash, Timestamp in querystore;
11 Unlock all tables;
12 Return actual dataset and PID;
```

To re-execute a query, we select the information from the querystore via the PID. This results in the original query, the time of execution and which tables have been selected (stored in AdditionalInformation). We then rewrite the query to target the historized tables instead of the actual tables (rewriting depends on the used historization approach which are described in Section 3.5). A clause that filters the rows according to the timestamp is added to the WHERE part. Now, we have a query *query_hist* that targets historized tables and only returns rows active at the execution time of the original query. To obtain the hash for the result of this query, we use the same rewriting used to obtain the hash of the original query resulting in *query_hist_hash*. We then execute *query_hist_hash* to obtain the hash for the result set which we check against the stored hash. After we verified that we can produce the correct hash, we know that the dataset created by *query_hist* will be the same as the original extracted one. Now, we execute *query_hist* and obtain the reproduced correct dataset. This process is described as pseudocode in Algorithm 4.

One might ask why there are no locking mechanisms added to the re-execution. SMOs and INSERT/CREATE/DELETE operations could alter the data in the used history table while a re-execution is in progress. A SMO could only set the *endtimestamp* of the data in a history table of a consumed table. As the timestamp we use for re-execution, called *originalquerytimestamp*, is strictly less than *endtimestamp*, and the WHERE clause we use checks for:

$$originalquerytimestamp <= endtimestamp \text{ OR} riginalquerytimestamp == null$$

we can conclude that SMOs have no impact on the returned results. INSERT/CREATE/DELETE operations, issued with *updatetimestamp* alter the data in the history table in the following way: on existing tuples they may set the *endtimestamp* to *updatetimestamp*, new inserted tuples have their *starttimestamp* set to *updatetimestamp*. As *originalquerytimestamp* is strictly less than *updatetimestamp*, we can conlcude that those operations do not change the returned tuples. As both cases, by design, do not alter the returned tuples, we have no need to isolate our re-execution from them.

Algorithm 3.4: Re-execute query from querystore via PID

```
  Data: PID for Querystore
  Result: Reconstruced original Dataset
1 Extract query information from querystore via PID;
2 Rewrite query to use historized tables;
3 Add starttimestamp <= executiontimestamp && (executiontimestamp <=
    endtimestamp || endtimestamp == null) clause to WHERE clause;
4 Execute query for hash;
5 if hash != stored hash then
6   | Abort execution as we could not create the same result;
7 end
8 Execute query for actual dataset;
9 Return actual dataset;
```

We have seen how our system handles the use cases for: SMOs, CRUDs and citeable query execution and re-execution from a very high level, giving us a good understanding of the overall system. In the following sections, we explain some design decision for our system and deep dive into the algorithms we used.

3.2 Attribute vs. Tuple Based Timestamping

In order to implement the re-execution of queries at a later time, we have to know which data was active at the time of the original execution. The period for which data is active is usually described as the valid time [SA86]. Based on the work in Temporal Data

Management [JS99], we explore two techniques that enables us to tell the valid time for the data stored in our history tables. After describing both techniques, we explain why we recommend to choose one over the other.

3.2.1 Tuple Based Timestamping

In Tuple Based Timestamping, we choose a single datarow as the unit of data for which we want to store the valid time. We extend the tuple by two attributes: *starttimestamp* and *endtimestamp*, both holding a timestamp. When a tuple is created, the *starttimestamp* is set to the current timestamp and *endtimestamp* is set to null, indicating that the tuple is currently valid. As soon as any attribute of the tuple is changed, the *endtimestamp* is set to the current timestamp and a new tuple with the changed attributes is added. When a SELECT statement is issued against a table, we can add the following clause to the WHERE clause:

```
starttimestamp <= currenttimestamp &&
(endtimestamp <= currenttimestamp || endtimestamp == null)
```

and we will only get the rows returned that are active at currenttimestamp. This allows us to extract which tuples were active at a given time. One drawback of this approach is, that if only one attribute is changed, the whole tuple needs to be recreated which may lead to significant storage overhead.

3.2.2 Attribute Based Timestamping

The level of granularity in Attribute Based Timestamping is a single attribute of a tuple instead of the whole tuple. In this approach we split the tuple into separate attributes, each with its own *starttimestamp* and *endtimestamp.* In addition, we have to add a unique identifier in order to being able to reconstruct the whole tuple. Assume a table consisting of the following three columns: *ID*, *Name*, *Job.* We have to create three tables, one for each attribute, to store this tuple historized. Reconstructing a tuple active at *currenttimestamp* requires the following SQL statement:

```
    SELECT id.ID, name.Name, job.Job
    FROM idtable id
    INNER JOIN nametable name on id.id = name.id
    INNER JOIN jobtable job on id.id = job.id
    WHERE
    (id.starttimestamp <= currenttimetsamp &&
        (id.executiontimestamp <= currenttimestamp ||
        id.endtimestamp == null)
    ) &&
    (job.starttimestamp <= currenttimetsamp &&
        (job.executiontimestamp <= currenttimestamp ||
        job.endtimestamp == null)
    ) &&
    (name.starttimestamp <= currenttimetsamp &&
        (name.executiontimestamp <= currenttimestamp ||
        name.endtimestamp == null)
    )
```

Consider a very wide table with a lot of attributes where a lot of updates occur that change only one attribute. In this case Attribute Based Timestamping saves some storage space, but comes with the cost of drastically more complex, and significantly longer running, SQL statements.

3.2.3 Design Decision

We have chosen to use Tuple Based Timestamping as our way to store historized data. Attribute based timestamping would require more complicated CRUD statements and a more complicated query rewriting in turn to save some storage space, in some scenarios. We have no data on the query time of both approaches, but we suspect that attribute based timestamping will be slightly slower due to the more complex tuple retrieval in cases where the complete tuple is needed. Retrieving single attributes should be quicker in attribute based timestamping, but, as we stated, we have no relying data here and can only speculate. In general, we see no big advantage of using the attribute based approach except in settings where extremely wide tables (i.e. tables with a massive number of attributes per tuple) see repeated changes to individual attributes only. Such settings may be encountered in domains where high-dimensional data is being analyzed (e.g. text mining, signal processing), but are less frequent in business settings. We thus opted for the tuple based timestamping approach.

SQL:2011 The upcoming standard SQL:2011 introduced some temporal features that could be used to solve the problem of getting data valid at a certain point in time [KM12]. Some major vendors, Microsoft, Oracle, DB2 and Postgres support some of those features. As MySQL, being ranked as the second most popular RDBMS in DB-Ranking[2], does not support these enhanced concepts, and we have chosen not to use those features. Our target was to design a system that is based on the current SQL Standard that can be implemented in every RDBMS supporting basic locking mechanisms.

3.3 Integrated vs. Separated vs. Hybrid historization approaches

We now have decided to use tuples as the level of granularity for historization of our data. It is now open how and where store those historized tuples. In Scalable data citation in

[2]https://db-engines.com/de/ranking

dynamic, large databases: Model and reference implementation[PR13], three models have been described: Integrated, Separated and the Hybrid approach; for storing historized tuples. For evaluation purposes we implemented all three and we now explore them in more detail on how they handle CRUD statements on a single table *employess* with the following columns: *ID, Name, Job.* All INSERT/DELETE/UPDATE statements in every approach also update the update the last update value in the metatable of the tables they alter and act according to the locking mechanism described in Section 3.6.

3.3.1 Separated Approach

In the separated approach, a second table *employeehist* is created that holds the same data as the original table, augmented by the *starttimestamp* and *endtimestamp*. This allows users of the database to query the original table as it is and no queries need to be changed. INSERT statements on the original table are also inserted to the the history table, with the *starttimestamp* being set to the query execution time. Issuing the following statements:

```
    INSERT INTO employees (1,'John', 'Developer');
    INSERT INTO employees (2,'Marie', 'CTO');
    INSERT INTO employees (3,'Jane', 'QA');
```

will result in tables holding the data as described in Table 3.1.Note that *starttimestamp* is abbreviated with *sts* and *endtimestamp* with *ets* due to formating reasons. We achieve this by rewriting the insert statement into one that also inserts data into the history table. The INSERT statement is rewritten to a the script in Listing 5.

Table 3.1: Separated Tables after Inserts

Employee		
ID	**Name**	**Job**
1	John	Developer
2	Marie	CTO
3	Jane	QA

Employeehist				
ID	**Name**	**Job**	**sts**	**ets**
1	John	Developer	T1	null
2	Marie	CTO	T2	null
3	Jane	QA	T3	null

UPDATE statements set the *endtimestamp* of every row that will be updated to the execution time. Then, the updated rows are added with the *starttimestamp* set to the execution time. The rows in the original table are updated normally. Issuing the following statement:

```
UPDATE employees SET Name = 'McJohn' employees.ID = 1;
```

will result in tables holding the data as described in Table 3.2. Note, that the *endtimestamp* of the old row in the history table equals the *starttimestamp* of the added updated row. The code that is issued can be seen in Listing 3.

Table 3.2: Separated Tables after Update

Employee		
ID	**Name**	**Job**
1	McJohn	Developer
2	Marie	CTO
3	Jane	QA

Employeehist				
ID	**Name**	**Job**	**sts**	**ets**
1	John	Developer	T1	T4
2	Marie	CTO	T2	null
3	Jane	QA	T3	null
1	McJohn	Developer	T4	null

DELETE statements set *endtimestamp* of every row in the history table that will be deleted in the actual table. Issuing the following statement:

```
DELETE FROM employees WHERE employees.ID = 2;
```

will result in tables holding the data as described in Table 3.3.

Table 3.3: Separated Tables after Delete

Employee		
ID	**Name**	**Job**
1	McJohn	Developer
3	Jane	QA

Employeehist				
ID	**Name**	**Job**	**sts**	**ets**
1	John	Developer	T1	T4
2	Marie	CTO	T2	T5
3	Jane	QA	T3	null
1	McJohn	Developer	T4	null

Normal SELECT statements that target this approach do not result in any changes.

As described by Pröll and Rauber[PR13], this approach has high storage demand as all data is copied. On the other hand, it enables to easily write queries and the historization does not influence query performance of the actual table. We talk about when this approach is useful in our evaluation in Section 4.

3.3.2 Integrated Approach

The integrated approach is like the separated approach, except that the historization information is stored in the actual table and no extra historization table is added. As the approaches are so similar, we omit the ongoing example.

Storing the timestamps for historization in the actual tables requires normal SELECT statements, that are interested in the current active data, to include the following WHERE clause:

```
    endtimestamp == null
```

You can find how INSERT statements are handled in Listing 5, UPDATES in Listing 6 and DELETES in Listing 7. This approach is quite useful in approaches where updates and deletes hardly occur like in sensor data that produce time series. Normal queries, interested in the current active data, suffer a performance hit due to the tables increased depth caused by UPDATE and DELETE statements.

We have got to note here, that we still create a history table at the creation of employee, but it will stay empty. It is only filled when the actual table is consumed by a SMO.

3.3.3 Hybrid Approach

The Hybrid Approach is similar to the Integrated approach and tackles the issue that queries interested in the current active data suffer a performance hit and need to be rewritten. We achieve this by storing the current data in the actual *employees* table together with the timestamp when it was added. The data is only moved to the historization table when it is updated or deleted.

Issuing the following statements:

```
    INSERT INTO employees (1,'John', 'Developer');
    INSERT INTO employees (2,'Marie', 'CTO');
    INSERT INTO employees (3,'Jane', 'QA');
```

will result in tables holding the data as described in Table 3.4. The way the INSERT statements are handled is described in the code in Listing 8

Table 3.4: Hybrid Tables after Inserts

Employee			
ID	**Name**	**Job**	**sts**
1	John	Developer	T1
2	Marie	CTO	T2
3	Jane	QA	T3

Employeehist				
ID	**Name**	**Job**	**sts**	**ets**

UPDATE statements move the rows that will be updated to the history table and set the *endtimestamp.* The rows in *employees* will be updated according to the update statement and their *starttimestamp* will also be set. Issuing the following statement:

```
UPDATE employees SET Name = 'McJohn' employees.ID = 1;
```

will result in tables holding the data as described in Table 3.5. Note how the *starttimestamp* in the employee table is the same as the *endtimestamp* in the history table. The way the UPDATE statements are handled is described in the code in Listing 9.

Table 3.5: Hybrid Tables after Update

Employee			
ID	**Name**	**Job**	**sts**
1	McJohn	Developer	T4
2	Marie	CTO	T2
3	Jane	QA	T3

Employeehist				
ID	**Name**	**Job**	**sts**	**ets**
1	John	Developer	T1	T4

DELETE statements move the rows they delete into the historization table with their *endtimestamp* being set. The rows in the actual table are deleted. *endtimestamp* of every row in the history Issuing the following statement:

```
DELETE FROM employees WHERE employees.ID = 2;
```

will result in tables holding the data as described in Table 3.6. The way the UPDATE statements are handled is described in the code in Listing 10.

Table 3.6: Hybrid Tables after Delete

Employee			
ID	**Name**	**Job**	**sts**
1	McJohn	Developer	T4
3	Jane	Qa	T3

Employeehist				
ID	**Name**	**Job**	**sts**	**ets**
1	John	Developer	T1	T4
2	Marie	CTO	T2	T5

The big advantage of this approach is, that it combines the reduced storage requirements of the Integrated Approach while keeping the useability of the Separated Approach. The drawback, however, is that queries that ask for a the tuples valid at a certain time have to be rewritten to take the actual table into account. A query asking for the tuples active at T3 needs to be formulated (or any other equivalent to it) to:

```
SELECT `employees_ref`.`id`,
       `employees_ref`.`name`,
       `employees_ref`.`job`
FROM   (SELECT `id`,
               `name`,
               `job`,
               `startts`,
               NULL AS `endts`
        FROM   `employees`
        UNION
        SELECT `id`,
               `name`,
                `job`,
                `startts`,
                `endts`
          FROM   `employees_1`) AS `employees_ref`
WHERE  ( `employees_ref`.`startts` <= t3 )  AND
       (( t3 < `employees_ref`.`endts` )  OR
        ( `employees_ref`.`endts` IS NULL )
       )
```

It is clear to see, that the queries that are issued to retrieve tuples active at a certain time require more rewriting than with the other approaches and will result in longer execution time.

3.4 Schema Evolution and Schema Management

As the underlying schema evolves over time, and we have to reproduce data as it was queried at a certain point in the past, we need to know the schema at a given point in the past. Therefore, we log the schema-information we need every time the schema is changed by an SMO into the database table *qubadcsmotable*. The name stands for Query-Based-Data-Citation Schema Modification Table. We will now take a look at how the table is structured, what schema information we need to track and how SMOs behave on our tables.

SMO-Table The table is created during system initialization and consists of the following columns:

- Id - an auto-incrementing integer acting as the primary key
- Schema - a field holding a C# Object serialized as JSON that describes our schema
- SMO - a field holding a C# Object serialized as JSON that describes the SMO that lead to this schema
- Timestamp - timestamp when this schema became active

Every time a SMO is issued against our system, a new row is created containing the resulting schema.

Schema-Information We consider a schema to be the aggregation of tables that exist in the system. A table in our case exists of the following fields:

- Name - the table name
- Schema - database schema in which the table is stored

- ColumnDefitions - an array containing how columns are defined, see below
- GUID - a GUID uniquely identifying the table across multiple schemas
- HistTableName - name of the history table for this table
- Histtableschema - schema of the history table for this table
- MetaTableName - name of the metatable for this table
- MetaTableSchema - schema of the metatable for this table

The GUID is needed as name and schema are not strong enough to identify a table across multiple schema-versions. Consider creating a table A, dropping it, and then creating again a table A. The GUID helps us to distinguish those two which in term makes the rewriting process easier.

A *ColumnDefintion* consists of the following fields:

- ColumnName - name of the column
- DataType - the datatype of the column
- Nullable - boolean that indiciates if the column is nullable
- AdditionalInformation - string field that can cointain additional information that is renderd at create table statements

Those columns allow us to render it in create table and add column SQL statements. In Listing 17, a sample schema has been serialized in JSON. Note, the additional array *histtables* in it. This could be omitted and derived from the information available in tables, it is only there to ease some coding in C#.

Schema Modification Operators We have now defined which elements a schema consists of in our environment. This schema can evolve through the use of SMOs (schema modification operators) that have been introduced in Section 2.2. When we evolve our schema through a SMO, we have to take care that no data is lost and that it is correctly historized. Therefore we have to take the following requierements into account:

- No citeable query is executed on the tables targeted by the SMO
- No INSERT/UPDATE/DELETE statement is executed on the Tables targeted by the SMO
- Only one SMO may be executed at any given time
- The schema stored in *qubdadcsmotable* needs to be updated accordingly

- If tables are consumed, the containing data needs ot be historized

The first two points are handled via setting the *canbequeried* flag of the metatables of the tables used by the SMO to false. As our citeable queries and INSERT/UPDATE/DELETE statements check that this flag is true, we know that they won't be executed and therefor those points are covered. A more detailed explanation is given in Section 3.6.

The third point, that only one SMO may be executed at a given time, is handeld first acquiring a lock in MySQL via:

```
SELECT GET_LOCK('SMO UPDATES',10);"
```

This line acquires a lock in MySQL or times out after 10 seconds. As all SMOs need this lock, we have ensured that only one will be active at any given time.

To cover the fourth point, storing the new schema, we take a look on the resulting schema produced by a SMO. For this, consider Table 3.7, which is also described as Table 2 in Graceful Database Schema Evolution: the PRISM Workbench[CMHZ09]. We use the following notation: $R(\overline{A}, B, C)$ denotes a table R that consist of a set of columns $\overline{A}$, as well as columns B and C. $R_2(...)$ is considered a different table to R(...) although they have the same name (i.e. their GUID is different in our system). The column Input Schema describes the schema before the operation (tables not targeted by the SMO are omitted), the column Output Schema describes the schema after the SMO (again, not targeted tables are omitted). The last column historized tables, shows which tables are consumed and therefore their data is moved to their historization tables.

Table 3.7: SMOs and their behaviour on schematas

SMO	**Input Schema**	**Output Schema**	**Historized Tables**
CREATE TABLE $R(\overline{A})$	-	$R(\overline{A})$	-
DROP TABLE R	$R(\overline{A})$	-	R
RENAME TABLE R INTO T	$R(\overline{A})$	$T(\overline{A})$	-
COPY TABLE R INTO T	$R(\overline{A})$	$R(\overline{A})$, $T(\overline{A})$	-
MERGE TABLE R, S INTO T	$R(\overline{A})$, $S(\overline{A})$	$T(\overline{A})$	R,S
PARTITION TABLE R INTO S WITH $\{$cond$\}$, T	$R(\overline{A})$	$S(\overline{A})$, $T(\overline{A})$	R
DECOMPOSE TABLE R INTO $S(\overline{A}, \overline{B})$, $T(\overline{A}, \overline{C})$	$R(\overline{A}, \overline{B}, \overline{C})$	$S(\overline{A}, \overline{B})$,$T(\overline{A}, \overline{C})$	R
JOIN TABLE R,S INTO T WHERE cond	$R(\overline{A}, \overline{B})$, $S(\overline{A}, \overline{C})$	$T(\overline{A}, \overline{B}, \overline{C})$	R,S
ADD COLUMN C [AS $\{$const$\}$ | $\{$func($\overline{A}$)$\}$ INTO R	$R(\overline{A})$	$R_2(\overline{A}, C)$	R
DROP COLUMN C FROM R	$R(\overline{A}, C)$	$R_2(\overline{A})$	R
RENAME COLUMN B IN R TO C	$R(\overline{A}, B)$	$R_2(\overline{A}, C)$	R

The question may arise, why renaming a table does not consume the input table, but renaming a column does. As noted above, we assign a GUID to every table when it is created, but not to its columns This GUID allows us to identify the table uniquely in every schema. This approach could be extended to columns, which would allow a better rewriting algorithm that allows the RENAME COLUMN SMO to not consume it's input table.

Last but not least, we have to handle the historization which depends on the historization approach. All tuples moved into the new created tables are handled as if they were issued by INSERT statements. The tuples of the tables that are consumed are handled as if a DELETE statement was issued, expect for setting the *endtimestamp* to indicate that they have not been deleted.

Listings 12 shows how a CREATE TABLE SMO is handled in the hybrid approach, Listing 14 shows how MERGE TABLE is handled and Listing 14 shows an ADD COLUMN SMO. The other SMOs are part of the delivered code package but have been left out here due to their length.

3.5 Query Store and Query Rewriting

When talking about the Query Store, we refer to the part of our API that handles the following two use cases: preparing and executing queries in a way that allows them to be re-executed, and re-executing already executed queries. At first, we will cover how queries are handled and stored, afterwards we will cover their re-execution.

3.5.1 Executing a Query

The basic algorithm has been described in Algorithm 1. It can be summarized as: rewrite query to ensure sorting, acquire read locks on tables, check can be queried, set updatetime, execute statement for fixity calculation, execute statement for real data, store information in table *querystore*. The information is stored in the table *querystore* which is defined as described in the Initialization part at Section 3.

Before we cover how we rewrite the query, we take a look on what queries are allowed in our system, as not all will be reproducible.

3.5.2 Query Restrictions

Consider the following select statements:

```
SELECT RAND();
SELECT NOW(3);
SELECT SQRT(PI());
```

It is obvious, that we would have to take special measures to rewrite the first two queries in a way, that we can re-execute them at a later point in time producing the same result. The third query, however, should be reproducible but it is based on floating-point arithmetic, and there are many pitfalls in floating point arithmetic. The actual calculation does not solely depend on the source code, it also depends on the used architecture, the compilation of the code, and many other factors. Consider executing the query on Version X.Y on Server Z running Windows. The database is now migrated to a new Server A running Linux and updated to Version B.C. It is highly unlikely that every floating point operation of the first setup will produce the same result in the second one. For more details on this topic, we refer to What Every Computer Scientist Should Know About Floating-Point Arithmetic[Gol91], and The pitfalls of verifying floating-point computations[Mon08].

Based on the observations above, to ensure that the queries issued by our system will be reproducible, we restrict them in the following way:

- No arithmetic calculations are allowed
- No function calls are allowed

Considering the restrictions above, our statements only allow references on columns, literals, and boolean comparisons of them in the WHERE and JOIN clauses. There is still one case to consider: float equality should be done via checking if the absolute difference of two values is below a certain threshold. In MySQL this check looks like:

```
ABS(a - b) <= 0.0001
```

Our prototype currently does not support this scenario nor does it throw any exception if float comparison is made, nevertheless, it could be easily extended to cover this case. We have to note that we encourage, from a reproducability point of view, to not use this comparison.

3.5.3 Ensure Stable Sorting

As described in R5 - Stable Sorting [RAvP16], many experiments done on data, especially ones in machine learning, are sensitive on the order of the data. Besides those experiments, we also need to have a fixed sorting for our fixity calculation. As data is moved between the actual tables holding the currently active data and the historization tables, we have to take special measures to ensure our data is sorted.

One way, as it is was used by Pröll and Rauber[PR13], is to sort by the primary keys of the tables involved. As our developed API does not track which columns of a table are marked as the primary key (as this would extend our schema evolution approach to contain integrity constraint evolution by ICMOs[CMDZ13]), we cannot use this approach. We therefore resort to extending the user defined sorting by sorting all remaining queried columns in ascending order.

To illustrated this, assume the following query is issued:

```
SELECT `employees_ref`.`id`,
       `employees_ref`.`name`,
       `employees_ref`.`job`
FROM   `employees` AS `employees_ref`
ORDER  BY `employees_ref`.`name` DESC
```

The query will be rewritten to:

```
SELECT `employees_ref`.`id`,
       `employees_ref`.`name`,
       `employees_ref`.`job`
FROM   `employees` AS `employees_ref`
ORDER  BY `employees_ref`.`name` DESC,
          `employees_ref`.`id` ASC,
          `employees_ref`.`job` ASC
```

We are aware that sorting is no cost free operation. Therefore, we suggest extending our implemented API to track information about the primary keys of tables and use this information to implement more efficient sorting.

3.5.4 Fixity Calculation

Fixity calculation is the process of calculating a single value that can be used to verify that the re-executed result is the same as the original one. Pröll and Rauber mention two methods they used: either computing a hash over the complete result set or, if possible, over the unique identifiers represented in the query result set[PR13]. They did not mention if they calculate the hash inside the database or on any other layer of their architecture, which can influence the computed fixity.

Suppose, in our architecture, we issue the following two select statements:

```
SELECT TIMESTAMP('2014-09-08 17:51:04.777')
SELECT TIMESTAMP('2014-09-08 17:51:04.7779')
```

MySQLs native C# database driver returns the same datetime value for both statements as the .Net datetime datatype only supports three significant digits for milliseconds. Other database drivers handle similar scenarios without truncation, for example Oracles ODBC driver returns the values of columns defined with the datatype NUMBER as strings. The lesson to learn here is, that drivers and their settings/configuration, may alter data, and thus may produce different results on the same query against the same database. Therefore, we conclude that we have to calculate the fixity information inside the database, as this information allows us to detect if the actual query we issue produces different results or not.

As we do not track primary key informations, we fall back to the option of calculating a hash over the resulting dataset. We do this by modifying the SQL statement. In the modified version, all columns are hashed, then concatenated by #, producing one string per row. Those rows are again concatenated, separated again by a # and hashed again.

To illustrate our hashing, consider the following query:

```
SELECT   `employees_ref`.`id`,
         `employees_ref`.`NAME`,
         `employees_ref`.`job`
FROM     `employees` AS `employees_ref`
ORDER BY `employees_ref`.`id` ASC,
         `employees_ref`.`NAME` ASC,
         `employees_ref`.`job` ASC
```

The query above is rewritten to:

```
SELECT   md5(
            group_concat(
                Concat_ws('#'
                    ,Md5(`employees_ref`.`id`),
                    Md5(`employees_ref`.`NAME`),
                    Md5(`employees_ref`.`job`)
                )
                separator '#'
             )
        )
FROM     `employees` AS `employees_ref`
ORDER BY `employees_ref`.`id` ASC,
         `employees_ref`.`NAME` ASC,
         `employees_ref`.`job` ASC
```

This way of fixity calculation puts a lot of computational load on the database but allows us to detect if the actual query on the data produced the same result. Suppose someone installed our middleware API on a different server and uses a different MySQL driver. We can still detect that the core of our system, the database, executes the query correctly, even if migration mistakes were introduced on the driver level. Still, the result we extract could be altered by the driver or changed behavior in our middleware (using different .Net Framework or migrating to .Net Core). To counter this scenario, we suggest extending our prototype to include a fixity information of the resulting data exchange format that is returned by the C# API, as a fixity on this level acts as a check of our whole API.

3.5.5 Additional Rewriting

For the integrated approach, we also extend the where clause to only include currently active rows. For the hybrid approach, we extend the where clause to contain a check that the starttimestamp is lower than the execution timestamp, although it could be omitted as we acquired READ LOCKs on the table and therefor no INSERT statements can take place. For the separated approach, we rewrite the queries to target the history tables (they could also target the actual tables).

3.5.6 Locking

As usual, we acquire read locks on the tables we want to query to prevent the data in it from being changed while we execute our query. We also check that the schema we are executing against has not changed since the query was issued to our middleware to ensure that our rewrites are correct. The usual check for the *canbequeried* flag on all tables that are queried is also included.

3.5.7 Query Execution

We have now all bits together on how to execute a query that returns the current active data with ensured sorting and fixity calculation on correctly locked tables. The resulting fixity information, timestamp etc. is stored in the *querystore* table. An example on the SQL statements issued for a query of the data in *employees*, using the hybrid approach, can be found in Listing 18.

3.5.8 Re-executing a Query

The whole re-execution process is described in Algorithm 4 and can be summarized as: extract query store information via PID, rewrite query, execute query for fixity information, execute query to retrieve the actual dataset.

At the rewriting step, we have to consider that the underlying data, as well as the schema, might have changed. Our historization approaches cover, that the underlying data is changed in a way that allows us to still identify the tuples active at the original execution time. We cover the impact of a changed schema in the next subsections for each historization approach. Taking into account the changed data and schema into our rewriting, we can get back the same tuples at re-execution, and therefore producing the same result, as when we originally executed the query.

Note, that we have not applied any read locks on the tables here as the data may only exist either in the table holding the current data or the historized table. As we query both tables (except for the separated approach) for the data active at the original query execution time, we can be sure that we get all the data that existed at the query execution time (the tuples can either be in the actual table or the historization table). If, however, any SMO is currently running and alters the original table, we may not be able to reproduce the result, but eventually, after the SMO is finished, the data will be in the historized table and we will be able to re-execute the query and get the correct result.

We considered using locks to ensure that the second case may not occur. A correct locking for this approach would be again to lock the tables with read locks and to check the *canbequeried* flag. However, as SMOs hardly occur, and we eventually will be able to reproduce the query, we concluded no need to implement the locking to guarantee a correct working system.

3.5.9 Rewriting - Integrated Approach

We recall from the integrated approach described in Section 2.2, that the current data is stored in the actual table and that the historization table only contains data after the actual table is consumed via a SMO. Therefore, our rewriting distinguishes, for every table in the query, the two following cases:

- the actual table was not consumed by a SMO and therefore still exists
- the actual table has been consumed by a SMO and therefore the data has been moved to the historized table

In the first case, we can query the actual table in the same way as it was issued in the original query (note, that the filtering for the current active rows had been applied in the original query, compare this with the paragraph Additional Rewriting in the last section). In the second case, we replace the references to the consumed table with the history table (as this is where the data now resides).

Detecting if a table has been consumed by a SMO is achieved by looking up if the table is still contained in the current active schema. To check this, we compare the GUID stored in the *additionalinformation* in the *querystore* table with the GUIDs of the tables of the current active schema. If we do not find a table with the matching GUID in the current schema, we look up the schema active at the query execution time and extract from there the table information, containing the schema and name of the history table.

3.5.10 Rewriting - Hybrid Approach

We recall from the hybrid approach described in Section 2.2, that the current data is stored in the actual table and all historized data (data having a set endtimestamp) is moved to the historization table. We again, distinguish if the actual table has been consumed by an SMO or not. In the case, it has been consumed, we only have to look for the data in the historized table (using the same way to find out which one this is as in the Integrated Approach). In the case the actual table still exists, we have to query the data from both tables using a subselect that creates the union of both tables. An example for this rewrite has been provided in the paragraph Hybrid Approach in Section 3.3.

3.5.11 Rewriting - Separated Approach

As we know that all data active at any given time can be retrieved from the history tables, we rewrite all queries to only target history tables.

3.6 Locking Mechanisms

In order to produce a reproducible query result, we have to ensure that the underlying data is not changed during the process of executing the query and storing the result in the *querystore* table. Also, we got to ensure that INSERT/UPDATE/DELETE statements are run in isolation, as well as SMOs. Therefor, our design goal is: given a set of tables, only one of the following operations is run at any given time:

- Citeable Query
- INSERT/UPDATE/DELETE statement
- SMO

A simple solution is, that each of those statements acquires a LOCK (READ in the case of citeable query, otherwise WRITE) on all tables affected. Unfortunately, MySQL does not allow us to alter tables via DDL statements when a WRITE LOCK is acquired for them. We therefor came up with the following solution: Each table t is associated with a metatable *t_metatable* consisting of only one row, storing the following two values: *last_update* and *canbequeried*. When a statement needs to acquire a lock on table t, the same lock needs to be acquired on *t_metatable*. After a SMO aquired the locks for all tables, the flag *canbequried* will be set to true on all *metatables*. Aftewards, the SMO releases the locks and continues to do the actual schema modification. When the SMO is finished, *canbequeried* will be set again to false on the *metatables*. Citeable queries and INSERT/UPDATE/DELETE statements acquire READ LOCKS on the *metatable* and afterwards check that *canbequeried* is true on all of them. If any *canbequeried* of the required tables is false, the operation will be aborted. To get a better understanding, we look at the issued SQL code of those three statement types targeting the *employees* table and discuss again why our locking mechanism works.

3.6.1 Citeable Query

A citeable query will be issued inside the following locking code which is part of Listing 18 (the code in the exmaples are exports from our test-suite which use a different dynamic schema each time, therefor may contain schematas in the form of: $hybrid\_ < guid >$):

```
SET autocommit=0;
SET sql_safe_updates=0;
LOCK tables
```

```
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees`
AS employees_ref READ,
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees_metadata`
READ,
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`qubadcsmotable`
READ;

SELECT canbequeried
FROM
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees_metadata`
-- C# Code checking canBeQueried
--throwing exception if one is false

--Code for actual selecting goes here

UNLOCK TABLES;

--INSERT into querystore
COMMIT;
```

By acquiring a read lock on the tables, we ensure that no INSERT/UPDATE/DELETE is in process and block SMOs from getting to set the *canbequeried* flag to true. If a SMO had been able to acquire the WRITE LOCK for one of the tables, the *canbequried* flag would have been set to true, indicating an ongoing SMO. In that case, our C# framework throws an exception and aborts the query. MySQL does not support any raise exception statement in normal scripts, otherwise we would have moved this check into the SQL code (note however, this handling could be simulated via the use of stored procedures).

3.6.2 INSERT/UPDATE/DELETE Statements

Those statements will be issued inside the following locking code which is part of Listing 9:

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES
    `hybrid`.employees WRITE,
    `hybrid`.`employees_metadata` WRITE
    `hybrid`.`employees_1` WRITE,
    `hybrid`.`QubaDCSMOTable` READ;

-- C# ensuring hist table has not changed
-- since statement was rewritten
```

```
SELECT canBeQueried
FROM `hybrid`.`employees_metadata`

-- C# Code checking canBeQueried,
-- throwing exception if it cannot be queried
-- C# Code ensuring that:
-- the schema has not been changed and that
-- employees_1 is still the history table
-- for employees

-- Actual INSERT/UPDATE/DLETE Handling

COMMIT;
UNLOCK TABLES;
```

By acquiring WRITE LOCKs on the *employees* table as well as the corresponding meta and history table, we ensure that no citeable query can be run in parallel. If a SMO had been able to aquire the WRITE LOCK for one of the tables, the *canbequried* flag would have been set to true, indicating an ongoing SMO. In that case, our C# framework throws an exception and aborts the query. We also acquire a READ lock on the table *QubaDCSMOTable* which contains our schemainformation. We make this to ensure to make sure that we have aquired the WRITE LOCK for the correct history table. Note, that we need a WRITE LOCK on the *metatable* as we update the *lastupdate* value.

3.6.3 SMO

All SMOs will be issued inside the following locking code which is part of Listing 14:

```
SET autocommit=0;
SELECT GET_LOCK('SMO UPDATES',10);
LOCK TABLES
    `hybrid`.`employees_metadata` WRITE;

UPDATE `hybrid`.`employees_metadata`
    SET canBeQueried = false;

COMMIT;
UNLOCK TABLES;

-- Actual SMO Handling

UPDATE `hybrid`.`employees_metadata`
```

```
    SET canBeQueried = true;

COMMIT;
SELECT RELEASE_LOCK('SMO UPDATES');
```

At first, we require that only one SMO can be run at any given time, allowing us to have discrete schema evolution steps. This is ensured by acquiring the global lock $'SMOUPDATES'$. Afterwards, the SMO aquires a write lock on the metatables of all tables involved. This can only be acquired when no INSERT/UPDATE/DELETE statement or citeable query is currently holding a lock on those tables. Afterwards the *canbequeried* flag is set to false, so all upcoming INSERT/UPDATE/DELETE statements and citeable queries will fail by design. When the SMO is finished, *canbequeried* is set to true again, allowing other statements to be executed. The last part is releasing the SMO lock.

3.6.4 Implementation notes

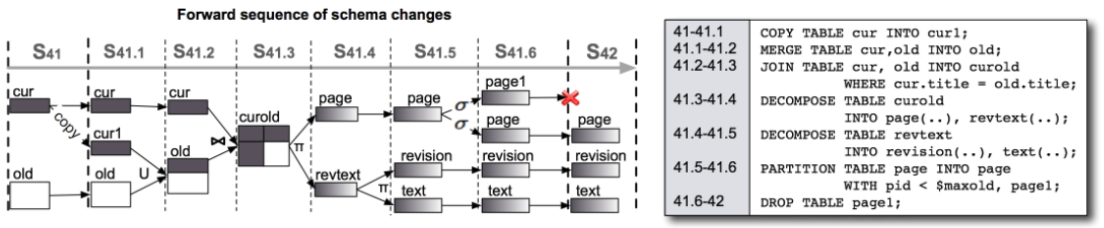
The implementation we have chosen was mainly driven by the capabilities of MySQL. We are unaware if other RBDMS provide locking features that make the use of the *canbequeried* flag obsolete. Implementing the locking inside our C# framework was considered, but we stayed away from it as MySQL provided everything we needed and it allows us to have our middleware stateless.

3.7 Optimizations and Extensions

We talk about some points were we thinkg our prototype could be optimized or extended.

3.7.1 SMO Batch Processing

Usually, more than one SMO is issued when the database schema evolves. We consider here an example provided for the PRISM Workbench[CMZ08] in Figure 3.1. The examples shows the starting schema S41 and the SMO evolution steps that lead to the schema version 42. In our system, each SMO would lead to a new schema version and to new history tables, although we can conclude from the figure that the actual resulting schema only contains the tables: *page*, *revision* and *test*. All in all we would only have to historize: *cur* and *old* from S41 and the three resulting tables in schema S42, and nothing in between.



```
41-41.1      COPY TABLE cur INTO cur1;
41.1-41.2    MERGE TABLE cur,old INTO old;
41.2-41.3    JOIN TABLE cur, old INTO curold
                       WHERE cur.title = old.title;
41.3-41.4    DECOMPOSE TABLE curold
                       INTO page(..), revtext(..);
41.4-41.5    DECOMPOSE TABLE revtext
                       INTO revision(..), text(..);
41.5-41.6    PARTITION TABLE page INTO page
                       WITH pid < $maxold, page1;
41.6-42      DROP TABLE page1;
```

Figure 3.1: Schema Evolution from S41 to S42 in Wikipedia

Table SMOs Consider the evolution of a table through ADD COLUMN, RENAME COLUMN and DROP COLUMN statements. Currently each statement produces it's own historization table. RENAME COLUMNs statement actually do not need to create a separate new historization table. Extending our rewriting to cover renamed columns would solve this and save some storage.

Consider a table with the following schema: $R_1(A, B, C)$. When DROP COLUMN C OF R is issued, all rows of R_1 are moved into R_1Hist, and we create a new historization schema for $R_2(A, B)$, named R_2Hist. As R_1 is consumed by the DROP COLUMN SMO, the containing data is moved into R_1Hist. As R_2Hist schema is a subset of R_1Hist, we could extract the current active rows from R_1Hist, instead of having to duplicate them to R_2Hist. This would allow us to save space by extending the query rewriting to consider R_1.

The same idea could be applied to ADD COLUMN statements, by using a view over the rows in the old historization table.

The drawback of the DROP COLUMN and ADD COLUMN approach is, that update and delete handling will get more complicated.

3.7.2 Uniqueness Tracking

Knowing any form of uniqueness for a tuple would allow us to use more efficient sorting strategies and more efficient fixity calculations, as described in the next paragraphs. We could achieve this by tracking information about the Primary Key for a given table. Our prototype could be extended to track the primary key with the stored schema information and require that every create table defines a primary key. SMOs like JOIN TABLE, that consumes two tables and results in a joined table based on the join condition, need to be updated as well for primary key support. DROP Column statements would also need special treatment, as well as how the primary key is handled in the integrated approach (*starttimestamp* and *endtimetsamp* columns are added, so the *starttimestamp* should be part of the primary key).

Another, easier to implement approach is adding a column to each table containing a GUID.

Sorting Strategies As mentioned earlier in Section 3.5, knowing a primary key, or any other uniqueness characteristic for tuples, would allow us to sort only on those columns and not on all columns. This should usually result in faster sorting.

Fixity Calculation As described earlier in Section 3.5, the fixity information could be calculated by only considering the unique values (primary key, or GUID as mentioned above) of the tuples returned. This should bring a performance boost.

Integrity Constraint Modification Operators Our system could be extended by the Integrity Constraint Modification operators introduced in: Automating the database schema evolution process[CMDZ13]. They contain primary keys, which would fit nicely with our other optimization and extensions points and seems like the logical next step for our API.

Using MySQL Binary Log In Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas[MCZ10], a system has been described that fulfills similar requirements as ours. It is an Information Archival System built in MySQL and they made the following remark about their implementation:

Finally, to enable the usage of our system in practice we implemented it as an extension of the MySQL master/slave replication technology—a history-enabled slave. This provides us with the capabilities of storing the history of the DB content, simply observing the MySQL binary log—leading to minimal performance overhead in the production database.

Basing our historization on the Binary Log could potentially increase the performance. It is left to check if it would fulfill our locking requirements

3.8 Big Database Considerations

Currently our implementation completely historizes a table as soon as an SMO is issued against it, effectivly introducing additional storage requirements of the resulting table of the SMO. This can be a problem for huge datatables consisting of millions of rows. Assume we are using the separated historization approach and issue an ADD COLUMN SMO, adding a column that is nullable to our big table. Our current implementation adds the column to the current active table and creates a new historization table, containing all tuples from the current active table. We observe here, that the old historization table basically contains the same data as the new historization table, expect for the null value of the newly added column.

Due to this behaviour, our currently proposed solution is not suitable for databases with huge tables containing millions of rows, as SMOs introduce too much storage overhead. We want to discuss a solution for the separated historization approach that does not need to duplicate historized tuples and tries to minimize the additional storage requirements by SMOs. Solutions for the integrated and the hybrid approach can be built upon the proposed solution.

3.8.1 Desired SMO Behaviour

SMOs issued in our described behaviour should have the following properties:

- Column Modification Operators (CMOs) do not create a new historization table, they modify existing ones or possibly migrate tuples
- Table Modification Operators (TMOs) do not create new tuples, they only migrate tuples between historization tables

Column Modification Operators As CMOs (ADD COLUMN, RENAME COLUMN, DROP COLUMN), we have to solve the following situation: column C, with datatype integer, is added, afterwards it is dropped, and then a new column C, with datatype varchar, is added. The problem we face is, that the historization table would now contain the column C twice. To solve this, we could use one of the following solutions:

- Each columnname is represented by a GUID in the history table
- Each columnname is suffixed with the version it is introduced
- Each columnname is suffixed with the timestamp it was created

We suggest solving it with the timestamp suffix. This allows us, that we can conclude the schema of a table at a given time simply by looking at the timestamps of the column names. If two columns have the same name, we know from the suffix which is the currently active one. If a column in the historization table has the highest timestamp suffix of all columns with the same name, and is not present in the current active table, we can conclude that this column was dropped. If we apply the same pattern to tablenames, as they suffer from the same name uniqueness problem, and never delete historization tables, we get the whole schema information encoded in the names and do not need a separate version tracking system. Encoding the schema information this way requires every issued query to be rewritten at the time it is being issued to the currently valid table and column names.

Table Modification Operators The desired behaviour for TMOs is, that they do not introduce any new tuples, instead they only migrate tuples between history tables.

Assume we have a table R and issue the following SMO: DECOMPOSE TABLE R into S(a,b), T(a,c). Where should the tuples of R_HIST be, after the SMO? We could move them into S_HIST and T_HIST and rewrite all queries targeting R_HIST that they have to target S_HIST and T_HIST now. By migrating all tuples to the new historization tables, tuples with a set *endtimpstamp* are also migrated to the new historization tables, although any new queries would never retrieve those. Additionally, they also impact the insert/update/delete performance on table R as they contribute to a longer historization table. Therefore, we conclude that tuples with a set *endtimpestamp* stay in R_HIST.

We now have to decide, where do we store tuples of R_HIST with no *endtimpestamp* now? They could reside in R_HIST, but this would force insert/update/delete statements on tables S and T to check for tuples in R_HIST. Considering following up SMOs on S and T, we will quickly get into complicated insert/update/delete handling logic that will impact the operational databases performance. Therefor, we decide that historized tuples with no set *endtimestamp* will be migrated to new historization tables.

As we migrate tuples between historization tables, historic queries targeting R_HIST have to be migrated to take S_HIST and T_HIST into account. They need to union the query result of querying R_HIST, S_HIST and T_HIST. Queries on S that have impact on S_HIST can be performed as described by our system as all tuples currently active in S are also represented in S_HIST. The same applies to T and T_HIST. This leaves us open with how historic queries have to be migrated, which will be covered in the next section.

3.8.2 SMO Handling and Historic Query Migration

Based on the desired behaviour, our new SMO handling is now done as described in the pseudocode in Algorithm 5. We will now take a look on how each of those steps looks for each SMO assuming we use the separated historization approach. Additionally, we assume that each table has a primary key and, without loss of generality, it is a column named ID. The discuss SMOs in the following order: first we have CREATE TABLE and DROP TABLE as they are the simplest, followed by CMOs and the other TMOs. If nothing is mentioned for the step "Save tuple reconstruction information", no information needs to be saved. Examples for the schemata will be given and follow this convention:

```
<Table>(<Columns>*)
The first column is always the primary key.

Example:
Employee(id,name)

Comments on the content of the table will be written
directly beneath it, starting with --.
STS is an abbreviation for starttimestamp.
ETS is an abbreviation for endtimestamp.
```

Algorithm 3.5: SMO handling

```
Data: SMO to apply, historic queries
Result: Tables altered by SMO, historic queries migrated
1 Modify actual tables according to SMO;
2 Create historization tables for new tables;
3 Save tuple reconstruction information;
4 Migrate tuples in historization tables;
5 Rewrite historic queries;
```

Save tuple reconstruction information We want to discuss this step of our algorithmn first, as it is part of all SMOs and depends on previously executed MERGE TABLE and DROP COLUMN SMOs as after those SMOs, the historization table may contain columns, that do not reflect any columns of the current active table.

A MERGE TABLE SMO may introduce a column that describes from which table the tuple originated, called *merge_origin*. When the resulting table of MERGE TABLE is consumed again, by a PARTITION TABLE R into S(a,b), T(a,b) for example, it is open what to do with the introduced column *merge_origin* Should it be migrated to S or to T? As the column does not bear any information of tuples in S or T we suggest creating an auxiliary table that stores this *merge_origin* column with the primary key columns of the historization table it was used. This saves all information needed to reconstruct the tuples of the historization table that was introduced by the MERGE TABLE SMO. The SMO MERGE TABLE, as we will see, can be implemented to not need this column and introduce the auxiliary table when the SMO is applied.

A similar scenario can happen with the DROP COLUMN SMO. If a column is dropped, one can choose to create a new historization table that also has the column dropped, or use the existing historization table and use null values for the dropped column. When a table is consumed, that has dropped columns in the historization table, we suggest not migrating the dropped column to new tables and instead, create an auxiliary table that contains the keys of the tuples and the values for the dropped columns. This scenario is discussed in more detail in the paragraph for the DROP COLUMN SMO.

CREATE TABLE R(a,b,c) Handling this SMO in our system is actually pretty simple as we create the actual table and the historization table with a suffix containing the timestamp. No tuples need to be migrated and no historic queries need to be rewritten. INSERT/UPDATE/DELETE statements that target R need to rewrite the column names when they target the historization table of R.

To illustrate this, consider the following example:

```
Starting schema:
/

Schema after CREATE Table R(a,b,c) at T0:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
```

DROP TABLE T Dropping a table is also relatively simple. The actual table is dropped no historized tuples have to be migrated and no queries have to be rewritten. The endtimestamp of tuples in the historized table has to be set to indicate when the active table T was deleted, and therefore the tuples of T too.

To illustrate this, consider the following example:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)

Schema after DELETE R at T1:
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- Rows before the DELETE with ets !== null have ets of T1 now
```

ADD COLUMN d [AS const | func(a,b,c)] INTO R First, we modify the actual table according to the SMO. Then, we modify the historization table by adding the column, with the timestamp suffix of the timestamp of the SMO execution added to the column name, and applying the values for the column in the following way:

Tuples with a set *endtimestamp* get the value *null* set for the column. If the default value for the new column is *null*, all tuples with *null* as *endtimestamp* also get the value *null* for the new column. However, if the SMO supplies an actual value for the column, may it be a *constant* or via $func(a, b, c)$, we treat these like we would treat a regular update to a tuple, i.e. we mark the existing active tuples (with no *endtimestamp* set) as deleted by adding the current timestamp as *endtimestamp* and re-insert the tuples including the new value for the added column with the current timestamp as *starttimestamp*.

We advice this way to enhance the archival quality of the historization table, as we consider adding a column without a *null* value to be tuple modifiying, thus new tuples should be created in the history table (one can think of ADD COLUMN with a default value different from *null* as the same ADD COLUMN with a *null* value and then updating all columns). However, depending on the real world situation, this tuple recreation may be skipped, as we know from the *starttimestamp* of the tuple compared to the columnname's creation timestamp suffix, that this column was added to the tuple later on, which allows to have no storage overhead introduced (which is the goal of this system).

No historic queries need to be rewritten.

To illustrate this, consider the following example:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)

Schema after ADD COLUMN d as null into R at T1:
R(a, b, c, d)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets, d_T1)
-- as null was used as the default value, no tuples are modified
```

DROP COLUMN d We modify the actual table by dropping the column. No new historization tables need to be introduced, as well as no tuples need to be introduced.

The historization table we have needs from now on the have *null* as the default value for this column.

Historic queries do not need to be rewritten as they still operate on the old schema with unchanged column names, wheres new queries from the live system will not include this column anyway.

If this table is later targeted by a PARTITION TABLE SMO, this column will not be moved to the new historized tables (the plural is intentional as a DECOMPOSE TABLE SMO introduced two follow up historization tables). However, we might have the case that some tuples have no set endtimestamp, but values inside a dropped column. We have two ways to solve this.

We could treat dropping column as "tuple modifying" and introduce new tuples when issued, the same way we handled the ADD COLUMN SMO. This way introduces a storage overhead and is not desired. The other way is, as described above, we keep the tuples unmodified, but we have to take care to keep this column information when the table is consumed by TMOs. When the table is consumed by a TMOs (like PARTITION TABLE), we create a new table $<Table>\_DROPPED\_COLUMNS$ that stores the *primarykey*, the *starttimestamp* and the dropped column. This allows us, to reconstruct the original tuple by joining the tuples we migrated to the new history table introduced by PARTITION TABLES with the information we migrated to $<Table>\_DROPPED\_COLUMNS$. This also implies, that the historic query rewriting step in the TMO that consumes this table (PARTITION TABLE in our example) has to take this into account when rewriting historic queries in the "Save tuple reconstruction information".

The question may arise, why to not migrate the column to the new table, as it will mostly contain *null* values, which usually take only one bit to represent. Depending on the implementation of the RDBMS, null values may take space, for example MySQLs MyISAM storage engine only stores a flag that indicates that a column value is null, nevertheless the space for the complete column is allocated[3]. However, migrating all dropped columns, although they will never contain values anymore, puts additional work on the database. Column and table statistics need to be updated and they decrease the maintainability of the overall system. In worst-case scenarios, the amount of columns in a table could lead to some system maximum limits being exceed (e.g. number of columns in a table, SQL statement length, number of columns in a single query, etc...). Therefore, we recommend not migrating dropped columns, although they usually only consume one bit storage per row.

To illustrate the influence of a DOP COLUMN SMO, consider the following example:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)

Schema after DROP COLUMN c at T1:
R(a, b)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- sts > T1 implies c_T0 is null
-- Future SMOS that migrate tuples away from R_HIST_T0
-- have to create a dropped columns table for c_T0
```

RENAME TABLE R into T The actual table R is renamed to T, as well as $R\_HIST\_ < TIMESTAMP >$ is renamed to $T\_HIST\_ < TIMESTAMP >$. Historic queries targeting R_HIST have to be rewritten to target T_HIST. Note, that we do not suggest changing the timestamp of the table. If this is desired (to have better archival quality) we suggest storing this renaming information in a separate table. An alternative solution is not renaming $R\_HIST\_ < TIMESTAMP >$, instead we create a new table $T\_HIST\_ < NEWTIMESTAMP >$ and migrate all tuples with no set *endtimestamp* to from R_HIST to the new created T_HIST. Historic queries targeting R_HIST now have to target T_HIST too and take the union of both results. In the process of migrating tuples, already dropped columns could be removed.

To illustrate this, consider the following example that also removes dropped columns: To illustrate the influence of a DOP COLUMN SMO, consider the following example:

```
Starting schema (c was deleted at T1):
R(a, b)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
--Note that R_HIST_T0 does not contain c_T0
--Therefore, c_T0 is dropped and needs to be removed
```

[3]https://dev.mysql.com/doc/internals/en/myisam-introduction.html

```
Schema after RENAME TABLE R into Tat T2,
with new history table and tuple migration:
T(a, b)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- contains all tuples with set ets of R_HIST_T0 before rename

R_HIST_T0_DROPPED_COLUMNS(a_T0,c_T0, sts, ets)
-- contains the values for c_T0 of tuples migrated to T_HIST_T1

T_HIST_T1(a_T2, b_T2, sts, ets)
-- contains all tuples with no set ets of R_HIST_T0 before rename
```

RENAME COLUMN b IN R TO d The actual table *R* is modified accordingly to this SMO.

The column *b* in *R_HIST* has to be renamed to *d* with the current timestamp of the SMO and historic queries need to be rewritten to query *d* instead of *b*. By this, we do not need to migrate any tuples and preserve the column name information in *d*. We only loose information on when the column *d* was originally added (which could be stored in a separate table).

Another way would be to create a new historization table with the column *d* instead of *b* with the correct timestamp in the tablename and in the columnname. Historic queries would now need to union the query results of both tables. Already dropped columns in *R* could be omitted in the new table.

Which strategy to apply, has to be decided on a case by case basis. Basically, we recommend creating a new historization table with the column *d* instead of *b*, as this preserves all schema information we need in the table and column names and also removes dropped columns. However, if for some reason, the process of migrating tuples is too expensive in terms of time, the first strategy could be employed. This could be a case if an API accessing the database has changed to access column *d*, but the database migration did not migrate *b* to *d*. Creating a hot-fix for this, without migrating tuples, to ensure the database and the APIs accessing it are online, could be a reason to employ this strategy.

To illustrated a RENAME, consider the following example that creates a new historization table:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)

Schema after RENAME COLUMN c to D at T1:
R(a, b, d)
```

```
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- contains all tuples with set ets of R_HIST_T0 before rename

R_HIST_T1(a_T0, b_T0, d_T1, sts, ets)
-- contains all tuples with no set ets of R_HIST_T0 before rename
```

MERGE TABLE R,S into T We modify the actual tables according to the SMO. A new historization table is created for the table T, containing an additional column *merge_origin*. We take all tuples from R_HIST that have no *endtimestamp* and save all tuple reconstruction information we need for them (depending on dropped columns and other *merge_origin* columns). Then, those tuples are migrated to T_HIST, with *merge_origin* being set to R. All historic queries, targeting R_HIST have now to union the query results from querying T_HIST with an additional WHERE clause that filters that the *merge_origin* column has to have the value R. The table S_HIST is handled accordingly.

Alternatively to using the column *merge_origin* in T_HIST, we could create a table $R\_HIST\_TUPLES\_IN\_T$, containing the primary key in T_HIST of tuples migrated from R to T. The same would apply to the tuples of S_HIST. This would not require keeping track of the *merge_origin* column.

If a table containing a *merge_origin* column is consumed, this information needs to be persisted in an additional table containing the *id*, *starttimestamp* and the *merge_origin* column, during the "Save tuple reconstruction informations" phase.

To illustrated a MERGE of R and S into T, consider the following example:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
S(a, b, c)
S_HIST_T1(a_T1, b_T1, c_T1, sts, ets)

Schema after MERGE R,S into T at T2, using a separate table
to store the tuple origin:
T(a, b, c)
T_HIST_T2(a_T2, b_T2, c_T2, sts, ets)
-- contains all tuples of: R_HIST_T0 and S_HIST_T1
-- that had no ets set

T_HIST_T2_ORIGIN(a_T2,origin,sts)
-- if a tuple in T_HIST_T2 came from S_HIST_T1, origin is
-- set to S, otherwise it is set to R

R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
```

```
-- contains all tuples with set ets of R_HIST_T0 before merge

S_HIST_T1(a_T1, b_T1, c_T1, sts, ets)
-- contains all tuples with set ets of S_HIST_T1 before merge
```

PARTITION TABLE R into S with *cond*, T We modify the actual tables according to the SMO. Two new historization tables are created, S_HIST and T_HIST.

The tuples from R_HIST with no set *endtimestamp*, satisfying *cond* are migrated to S_HIST and those not satisfying *cond* are migrated to T_HIST.

Historic queries targeting R_HIST have now to union the results of the query on S_HIST, T_HIST and R_HIST.

```
 Starting schema:
 R(a, b, c)
 R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)

 Schema after PARTITION TABLE R into S with true, T, at T1:
 S(a, b, c)
 S_HIST_T1(a_T1, b_T1, c_T1, sts, ets)
-- containing all rows of R_HIST_T0 that had no set ets
-- and satisfy the condition true

 T(a, b, c)
 T_HIST_T1(a_T1, b_T1, c_T1, sts, ets)
-- containing all rows of R_HIST that had no set ets
-- and do not satisfy the condition true

 R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- containing all rows that had set ets before PARTITION TABLE
```

DECOMPOSE TABLE R into S(a,b) T(a,c) For this decompose table operation, we assume that the columns a contain the keys, otherwise they cannot be reconstructed. If the key for those tuples is not desired in the resulting tables, we have to migrate it nonetheless, but name the column containing the primar key *decompose_key*. Subsequent consumptions on the table have to store this key during "Save tuple reconstruction information;".

We modify the actual tables according to the SMO. Two new historization tables are created, S_HIST and T_HIST.

The tuples from R_HIST with no endtimestamp, are split according to the SMO and are migrated to S_HIST and T_HIST. Historic queries targeting R_HIST, now have to union the query result on *R_HIST* with the query result on the reconstructed tuples from *S_HIST* and *T_HIST*.

To illustrate a DECOMPOSE consider the following example:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)

Schema after DECOMPOSE R into S(a,b) T(a,c), at T1:
S(a, b)
S_HIST_T1(a_T1, b_T1, sts, ets)
-- containing all rows of R_HIST_T0 that had no set ets
-- only column values that got migrated to S are represented

T(a, c)
T_HIST_T2(a_T1, c_T1, sts, ets)
-- containing all rows of R_HIST_T0 that had no set ets
-- only column values that got migrated to T are represented

R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- containing all rows that had set ets before PARTITION TABLE
```

JOIN TABLE R,S INTO T WHERE cond This SMO is quite tricky, depending on the join condition and what is the resulting primary key in T.

First, we modify the actual tables according to the SMO. Afterwards, the historization table *T_HIST* is created. All tuples of R that found a tuple in S such that they satisfy *cond* will exist in T one or more times. As we desire that *T_HIST* looks exactly like T after the SMO, we have to store some tuple reconstruction information. In a separate table called *R_HIST_T_HIST_Key_Mapping*, we store the keys of all tuples of *R_HIST* (i.e. *id* and *starttimestamp*) and the key of one tuple of *T_HIST* that they are part of. All tuples with a key in this table are deleted from *R_HIST*. This allows us to reconstruct all tuples that have been in *R_HIST*. The same has to be done for *S_HIST*. Historic queries need to be rewritten for this query reconstruction.

To illustrate a JOIN TABLE consider the following example:

```
Starting schema:
R(a, b, c)
R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
S(x, y, z)
S_HIST_T1(x_T1, y_T1, z_T1, sts, ets)
--Note that a is the key of R and x is the key of S
```

```
 Schema after JOIN TABLE R,S into T WHERE R.a == S.x  at T2:
 T(a, b, c, y, z)
-- Primary Key of T is now a

 T_HIST_T2(a_T2, b_T2, c_T2, y_T2, z_T2, sts, ets)
-- This one contains all tuples that found join partners

 R_HIST_T0_T_HIST_T2_Key_Mapping(a_T2, sts)
-- This contains a distinct list of keys of tuples that
-- existed in R_HIST_T0 that got migrated to T_HIST_T2.
-- Tuples with no set ets that found a join partner
-- were migrated to T_HIST_T2.

 R_HIST_T0(a_T0, b_T0, c_T0, sts, ets)
-- Contains all tuples that had a set ets, as well as
-- all that had no set ets that found no join partner

 S_HIST_T1_T_HIST_T2_Key_Mapping(a_T2, sts)
-- same like R_HIST_T0_T_HIST_T2_Key_Mapping, but for S_HIST_T1

 S_HIST_T1(a_T1, b_T1, sts, ets)
-- Contains all tuples that had a set ets, as well as
-- all that had no set ets that found no join partner
```

3.8.3 Implementation Notes and System Properties

We want to discuss a few things considering on how to implement this designed system and some properties of the designed system.

Tuple Reconstruction Information If DROP COLUMN is implemented by creating a new historization table without the dropped column (and all column values for the dropped column for tuples with no set *endtimestamp* is stored), and MERGE TABLES is implemented by storing the *merge_column* information in a separated table, the step: "Save tuple reconstruction information" can be omitted.

Relationship between Active Table and Historization Table In our designed system, the columns of the active table are all part of the historization table. Every tuple in the active table is represented in the currently active historization table, and not in any previous historization table. This simplifies the handling of insert/update/delete statements on the current active table and ensures that their performance is independent on previous performed SMOs.

Inferring Schema The current schema can be inferred from the active tables. This allows us to detect in the historized tables, if a table has been dropped.

If RENAME COLUMN, DROP COLUMN and RENAME TABLE produce a new historization table, and they both migrate tuples and handle historic query rewriting, we can have complete information about how a schema looked like at any given time considering the table and column suffixes.

Historic Query Rewriting We want to give an example on how we would implement rewriting queries using common table expressions. They are supported by every major SQL vendor and can lead to easier historic queries in our case.

We start our example with a table R and we query the columns ID,A,B. Querying R at a time $T1$ would result in the following query:

```
-- Assume Table R was created at T0

-- Original Query on R

SELECT ID,A,B
FROM R

-- Query on History table with CTE
WITH R (ID, A, B)
AS
(
    SELECT ID_T0, A_T0, B_T0
    FROM R_HIST_T0
    WHERE
 R_HIST_T0.STARTTIMESTMAP <= T1 AND
(T1 < R_HIST.ENDTIMESTAMP ||
 R_HIST_T0.ENDTIMESTAMP == NULL)
)
SELECT ID,A,B
FROM R
```

Assume now, a MERGE TABLE R,S into T was issued at time $T2$, with moving the merge information into $R\_HIST\_T\_HIST\_MERGEINFO$. Rewriting our query would result in:

```
WITH R_HIST (ID, A, B)  AS  (
    SELECT ID_T0, A_T0, B_T0
    FROM R_HIST_T0
    WHERE
 R_HIST_T0.STARTTIMESTMAP <= T1 AND
(T1 < R_HIST.ENDTIMESTAMP ||
 R_HIST_T0.ENDTIMESTAMP == NULL)
)
WITH R_TUPLES_IN_T (ID,A,B) AS (
SELECT ID_T0, A_T0, B_T0
FROM R_HIST_T_HIST_MERGEINFO_T2 r1
inner join
 T_HIST_T2 t1 on
 r1.id = t1.id AND
 r1.starttimestamp = t1.starttimestamp
WHERE
t1.STARTTIMESTMAP <= T1 AND
(t1.ENDTIMESTAMP == NULL ||
 T1 < t1.ENDTIMESTAMP)
)
WITH R (ID, A, B)  AS  (
    Select ID,A,B
    FROM R_HIST
    UNION
    SELECT ID,A,B
    FROM R_TUPLES_IN_T
)

SELECT ID,A,B FROM R
```

This way of migration historic queries with CTEs reflects the SMOs that changed the historization tables. It also reflects the tree representing where tuples of R_HIST have been migrated too and it is still quite readable.

Performance Considerations The problem of this historic query migration rewriting algorithmn is the fact, that it very much relies on UNIONs, resulting in bad historic query execution performance. We will see the impact of UNIONs on queries in our evaluation in Section 4.6.2.

Primary Key Tracking The system designed here relies on information about the primary key of each table. SMOs that create new tables need to know what the resulting primary keys of the created tables are. Special cases, like dropping the primary key or decomposing a table in a way that one table that does not contain the original primary key column have to be considered. We suggest to consider the addition of integrity constraint modification operators (ICMOs) when implementing this system. ICMOs have been described int Automating the database schema evolution process [CMDZ13].

3.8.4 Final Notes

The system we designed here is an alternative version that is built around minimizing tuple recreation when issuing SMOs. The result is a trade-off for historic query performance. We did not implement this system, but are looking forward seeing it implemented and compared to our current system.

3.9 Code Metrics

At last, we want to talk about some code-metrics of our API. Our developed API has been tested with 73 automatized integration tests implemented via .Net xUnit tests. They cover the following areas: globalupdatetimestamp is correctly set after INSERT/UPDATE/DELETE and SMOs, querystore re-execution works after INSERT/UPDATE/DELETE statemetns, querystore re-execution works after SMOs (including combined scenarios of SMOs for renaming) and some tests for the used schemamanger, SQL renderer and some internally used classes. The tests have proven useful as we needed to ensure that the different historization approaches behave the same, and also to ensure that refactorings were done correctly.

Figure 3.8 shows the Code Metrics calculated for our C# solution by Visual Studio 2017. The maintainability index calculated by Visual Studio has three ranges: Red, from 0-9, Yellow from 10-19, and Green from 20-100. We therefor can consider our API with a value of 79 quite maintainable.

Table 3.8: Code Metrics of Visual Studio 2017

C# Project	**QubaDC**	**QubaDC.Evaluation**	**QubaDC.Tests**
Maintainability Index	79	74	80
Cyclomatic Complexity	2031	468	255
Depth of Inheritance	3	2	2
Clas Coupling	341	155	117
Lines of Code	5401	156	985

3.10 Summary

We started with an overview of our system and how the use-cases for it are handled. Afterwards, we examined how we handle tuple based timestamping in the three historization approaches we implemented. We described how schema evolution is managed and how this affects query rewriting in our historization approaches. The last part of our system we examined are the implemented locking mechanisms that guarantee reproducible queries. At last, we discussed our implementation with respect to databases containing a huge amount of rows and proposed a different way to implement SMOs and historic query rewriting, resulting in a system that needs less storage space for history tables but has worse historic query execution time. At last, we took a look at Code Metrics of our system implementation.

As we now have now implemented the system, we can start evaluating our three different historization approaches and see the impact they have on query time and storage size.

CHAPTER 4

Evaluation

In this chapter, we present an experimental evaluation of the implemented prototype. Our evaluation targets to answer the following two questions for each historization strategy we have:

- How long does it take our system take to execute certain operations (CRUD, querystore operations)?
- How is the storage size of our database effected by those operations?

As a baseline, we use a simple implementation of our API which basically executes the given operation without historization or locking. This allows us to estimate the additional cost introduced by our historization approaches. After examining how our system behaves, we end with a discussion of the strengths and weaknesses of our approaches and finish with guidelines on when to use which approach.

We have chosen to not evaluate the performance of SMO statements as they rarely occur in the real world and we advice executing them in a maintenance mode where no other operations are allowed on the database.

4.1 Environment

We use the experiment environment summarized in Table 4.1.

Table 4.1: Setup Environment

Envrionment	**Description**
CPU	Intel(R) Core(TM) i7-6700 CPU, 3.4Ghz, 4 Cores
RAM	16 GB, DDR3
Hard Disk	250 GB, SSD
OS	Windows 10, x64 (Version 10.0.15063)
.Net	.Net Framework 4.5.2
MySQL	MySQL 5.7.20

All our tests are run against the table *datatable* with the following columns:

- Phasenumber - integer set to identify in which iteration the tuple was added
- ID - unique integer
- Section - A column holding random generated values in a given range. Used to make statements that target a certain percentage of the rows.
- ValueToUpdate - a value we use for updates
- CLOBPayLoad - a MEDIUMTEXT field we fill with random strings of length 10000

For index, we use a primary key (which implies a clustered index) for the column ID. In the integrated approach, the column *starttimestamp* is added to the primary key. No other index are used, if not mentioned otherwise. The table is created with InnoDB as the storage engine.

4.2 Measuering Methods

We got two different parameters to measure: the time it takes our API takes to execute a given operation, and the database size.

To measure how long it takes our API to execute a given operation, we make use of the class $StopWatch$ provided by the .Net Framework. An example call taken from our evaluation library to measure how long a CRUD operation takes, can be seen in Listing 1.

```
 List<long> insertValues = new List<long>();
QubaDC.CRUD.InsertOperation[] inserts =
 ↪  InsertGenerator.GenerateFor(1, Inserts, dbname);
Stopwatch sw = new Stopwatch();

int cnt = 0;
foreach (var insert in inserts)
{

    sw.Start();
    quba.CRUDHandler.HandleInsert(insert);
    sw.Stop();
    insertValues.Add(sw.ElapsedMilliseconds);
    sw.Reset();
    cnt++;
    if (cnt % 1000 == 0)
        Console.WriteLine("Inserted " + cnt);
}
long sum = insertValues.Sum();
```

Listing 1: Example Insert operation measuring in .Net

To measure the size of the tables in a database, we have to choose between the two following methods: Selecting the data from the command `SHOW TABLE STATUS` (equivalent to `information_schema.tables`), or selecting the data from the table `information_schema.INNODB_SYS_TABLESPACES`.

We extract the actual used data from the `SHOW TABLE STATUS` command via the following formula: $DATA\_LENGTH + INDEX\_LENGTH - DATA\_FREE$. However, the `SHOW TABLE STATUS` approach suffers from the disadvantages described in a blog post[1]. They can be summarized as the following: the data is not updated in real time, and it is not accurate. We can counter those by ensuring the following:

- `innodb_stats_persistent` is turned off
- `innodb_stats_on_metadata` is turned on
- ensure tables are flushed via `FLUSH TABLES` statement [2]
- ensure tables are analyzed

[1] https://www.percona.com/blog/2016/01/26/finding_mysql_table_size_on_disk/
[2] https://stackoverflow.com/questions/3169525/mysql-trouble-with-information-schema-tables

Our evaluation scripts ensure that those four requirements are fulfilled. Still, we experienced evaluation problems due to not accurate values. For example, we inserted 100.000 rows in the *datatable* of our three historization approaches. The $SHOW\ TABLE\ STATUS$ command showed more space consumption for the table *datatable* in the separated approach compared to the integrated approach, although the latter one stores an additional timestamp per row.

We tried getting better results via `information_schema.INNODB_SYS_TABLESPACES`, and we ran into other anomalies. For 100.000 inserted rows, the hybrid table used the same space as the separated table, although the *datatable* stored in the hybrid mode used an additional timestamp per row.

As both approaches are not reliable, we have chosen to go with the `SHOW TABLE STATUS` approach as it returned more fine grained data, showing the actual $DATA\_LENGTH$, the $INDEX\_LENGTH$ and $DATA\_FREE$, compared to $FILE\_SIZE$ of the `information_schema.INNODB_SYS_TABLESPACES` table. As `SHOW TABLE STATUS` also returns the estimated rows, we considered correcting the data by using the following formula:

$$\frac{DATA\_LENGTH + INDEX\_LENGTH - DATA\_FREE}{TABLE\_ROWS} * ActualRows \quad (4.1)$$

Using this formula did not improve the data and we decided to not introduce this formula.

4.3 Insert Performance

Setup To measure the insert performance, we inserted 100.000 rows into our *datatable*.

Storage We expect our measures of the sum of the actual and the table sizes to fulfill the following formula:

- $Size(Separated) > 2 * Size(Simple) > Size(Integrated) = Size(Hybrid) > Size(Simple)$

As we remember, the separated historization approach stores the tuples in the actual table and in the historized table. Therefore, each tuple is stored twice and we store an additional *starttimestamp* in the historization table, which should result in a size bigger than twice of the simple approach. We expect the integrated approach to consume the same amount of space as the hybrid approach, as the *endtimestamp* is stored as null and null values do not consume space in InnoDB[3]. The simple approach stores no historization information and should consume the least amount of space. The results of our evaluation can be seen in Table 4.2, note that we use Mebibytes instead of Megabytes.

[3]`https://dev.mysql.com/doc/internals/en/innodb-field-contents.html`

Table 4.2: Table Space consumption after 100.000 Row inserts - Values in MiB

Measured Value	**Simple**	**Separated**	**Hybrid**	**Integrated**
DataTable size	1565.52	1565.52	1567.52	1564.52
DataTable_Hist size	0.00	1566.52	0.02	0.02
Sum	1565.52	3132.04	1567.54	1564.54
Percentage of Simple	100%	~ 200,1%	~100,1%	~100,0%

Our results mostly conform with our prediction - we consider the deviations are due to inaccurate results from the `SHOW STATUS TABLES` command.

Run-Time We expect our measured values to fulfill the following formula:

- $time(Separated) > time(Integrated)$ ~$time(Hybrid) > time(Simple)$

As separated inserts two rows instead of one, it should take slightly longer than inserting data into the integrated approach. Integrated and hybrid should be roughly equal, and simple should be the fastest as no locks need to be acquired.

The results of our evaluation can be seen in Table 4.3.

Table 4.3: Insert Time consumption for 100.000 Rows

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
Avg Time	39.0 ms	97.2 ms	81.6 ms	80.0 ms
Min Time	31.0 ms	71.0 ms	64.0 ms	64.0 ms
Max Time	2256.0 ms	2997.0 ms	348.0 ms	2189.0 ms
Complete Time	65.0 min	162.2 min	136.1 min	133.5 min

We see that the average time for a insert statement reflects our expectations. The difference between the average time and the minimum time of separated and hybrid/integrated are big enough for us to conclude: separated is the slowest approach on insert time. The maximum time needed for an operation was only measured to see how much influence the running system can have. We can only conclude that the hybrid system was lucky in its execution and got no high max time, although it was slightly slower than the integrated approach. The difference between hybrid and integrated approach can not be considered significant. It can also be concluded, that no matter which system we use, insert time will at least double.

4.4 Update Performance

We measure the update performance in the following four different ways:

- Updating every row by ID of a table once
- Updating 1/10th of the rows by $CLOBPayLoad$ of a table once
- Updating 1/3 of the rows by $Section$ of a table multiple times
- Updating the whole table 20 times

4.4.1 Update Every Row By Id

Setup To measure the update performance of a single row with an index on the column in the WHERE clause, we update the column $ValueToUpdate$ by increasing it by one, for every row filtered by the column ID. Our tests are run against a table with 1000 rows. Indexes are still only applied to the actual tables.

Storage Size We expect to achieve the following measures (note that we distinguish between the history and the actual tables here):

- $size(Separated) + size(Separated\_hist)$ will increase by 50%
- $size(Integrated) + size(Integrated\_hist)$ will double
- $size(Hybrid) + size(Hybrid\_hist)$ will double
- $size(Simple)$ will stay the same
- $Size(Separated) = Size(Simple)$
- $Size(Separated\_Hist) = 2 * Size(Separated)$
- $Size(Integrated) = 2 * Size(Simple)$
- $Size(Integrated\_Hist) = 0$
- $Size(Hybrid) = Size(Simple)$
- $Size(Hybrid\_Hist) = Size(Simple)$

The results of our evaluation can be seen in Table 4.4, note that we use Kibibytes. The values in the table roughly match up with the expected values, except for the size of the integrated table.

Table 4.4: Update size after updating each row once - Values in MiB

Measured Value	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**Result**	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
DT size	13.52	13.52	13.52	13.52	15.52	15.52	15.52	32.52
DT_Hist size	0.00	0.00	13.52	28.52	0.02	14.52	0.02	0.02
Sum	13.52	13.52	27.04	42.04	15.54	30.04	15.54	32.54
% of Simple	100%	100%	200.0%	310.8%	114.8%	222.2%	114.8%	240.7%
% of Growth	/	0%	/	155.4%	/	193.3%	/	209.4%

Run-Time We expect to achieve the following measures:

- $time(Simple) > time(Hybrid)$ ~$time(Integrated) > time(Separated)$

Updates on simple should be the fastest, as only one update needs to occur. Separated should be the slowest, as two updates (actual table and setting the *endtimestamp* in the history table) and one insert (inserting the new tuple in the history table) need to occur. Hybrid needs to insert the value into the history table and update the actual table. Integrated needs to update the *endtimestamp* in the actual table and insert the tuple again into the actual table. We suspect that they both will roughly need the same time, although hybrid could be faster.

The results of our evaluation can be seen in Table 4.5. The most important result is, that separated is extremely slow. All systems managed to compute in under 3 minutes while it took our separated approach around 38 minutes. The main problem we found was that no index had been specified for the history table. We added a index on the hist table columns of the separated and the hybrid approach for the columns ID and *starttimestamp*. The results of a second run can be seen in Table 4.6. We see, that after adding an index on the history table, that the separated approach can handle updates as fast as our other approaches. The required storage for the history tables did not increase as the primary key is stored with the actual data[4,5].

Table 4.5: Update time after updating each row once - without indexes

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
Avg Time	51.1 ms	2290.6 ms	103.9 ms	112.1 ms
Min Time	43.0 ms	1198.0 ms	91.0 ms	94.0 ms
Max Time	78.0 ms	7167.0 ms	218.0 ms	2460.0 ms
Complete Time	51.2 s	38.2 min	103.9 s	112.1 s

[4] https://dba.stackexchange.com/questions/44520/does%2Dthe%2Dsize%2Dof%2Dthe%2Dprimary%2Dkey%2Dcontribute%2Dto%2Dtable%2Dsize

[5] https://dev.mysql.com/doc/refman/5.5/en/innodb-index-types.html

Table 4.6: Update time after updating each row once - with index on hist table

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
Avg Time	51.6 ms	114.6 ms	103.9 ms	114.3 ms
Min Time	46.0 ms	101.0 ms	91.0 ms	102.0 ms
Max Time	93.0 ms	406.0 ms	218.0 ms	373.0 ms
Complete Time	51.6 s	114.6 s	103.9 s	114.3 s

4.4.2 Update 1/10th of Rows By CLOBPayload

Setup To measure the update performance of a single row with no index on the columns used in the WHERE clause, we update the column $ValueToUpdate$ by increasing it by one, for 100 random selected rows filtered by the column $CLOBPayload$ of a table containing 1000 rows. A primary key (i.e. clustered index) is applied to ID and *starttimestamp* in the actual tables and history tables.

Storage Size The results of our evaluation can be seen in Table 4.7. The values for simple, hybrid and integrated line up with what we would expect. For separated tough, we see that the result values are pretty inaccurate. We observed that the values for the history table increased as follows:

- $ROWS$ changed from 1000 to 1100
- *Data_Lenght* changed from 17317888 to 20447232
- *Data_Free* changed from 1048576 to 4194304

We conclude from this observation that: a.) our data was stored correctly as the row count is correct (verified with a `SELECT COUNT(*)` statement) b.) some reorganization happened as *Data_Length* increased c.) the storage was probably overestimated and now underestimated.

Table 4.7: Update size after updating 1/10 of the rows once - Values in MiB

Measured Value	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**Result**	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
DT size	13.52	13.52	13.52	13.52	15.52	15.52	15.52	16.52
DT_Hist size	0 .00	0.00	15.51	15.5	0.02	2.52	0.02	0.02
Sum	13.52	13.52	29.03	29.02	15.54	18.04	15.54	16.52
% of Simple	100%	100%	214.7%	214.6%	114.8%	133.4%	114.8%	122.2%
% of Growth	/	0%	/	-0%	/	116%	/	106.3%

Run-Time The results of our evaluation can be seen in Table 4.8. We can clearly see in the complete time of our results, that the separated approach is most sensible to existing indexes, followed by integrated approach, because they both have to touch the tables they are updating more often.

Table 4.8: Update time after updating1/10 of the rows once

Measured Values	Simple	Separated	Hybrid	Integrated
Avg Time	737.9 ms	2059.3 ms	1384.3 ms	1661.8 ms
Min Time	543.0 ms	1576.0 ms	1141.0 ms	1310.0 ms
Max Time	4500.0 ms	4257.0 ms	3693.0 ms	3979.0 ms
Complete Time	73.8 s	3.4 min	2.3 min	2.7 min

4.4.3 Updating 1/3rd of the rows by *Section* of a table 20 times

Setup To measure the update performance of multiple row update with no index on the columns in the WHERE clause, we update the column $ValueToUpdate$ by increasing it by one, for every row in a given *section*. The column *section* holds one of the following values: $0, 1, 2$ and UPDATEs targets all rows of one of those sections. We started our test from a table with 1000 rows, primary indexes were added on the history tables and we issued the update statement 20 times.

Storage Size The results of our evaluation can be seen in Table 4.9. We only want to point out, that the result in the separated approach is slightly bigger than for hybrid and integrated. This may be due to MySQL internals. The actual row lengths have been checked and are equal.

Table 4.9: Update size after updating all rows of a given *section* 20 times - values in MiB

Measured Value	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**Result**	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
DT size	13.52	13.52	13.52	13.52	15.52	15.52	15.52	119.52
DT_Hist size	0.00	0.00	15.52	121.52	0.02	103.52	0.02	0.02
Sum	13.52	13.52	29.04	135.04	15.54	119.04	15.54	119.54

Run-Time The results of our evaluation can be seen in Table 4.10. We observe that the performance of the separated and the integrated approach are terrible compared to the hybrid and the simple approach, and that it also decreased over time. We suspect a missing index on the *endtimestamp* column to be the reason. This will be investigated in the next Section.

Table 4.10: Update time of running 20 updates on all rows of a given *section*

Measured Values	Simple	Separated	Hybrid	Integrated
Avg Time	699.6 ms	6.9 s	1558.5 ms	6.0 s
Min Time	608.0 ms	2553.0 ms	1348.0 ms	2098.0 ms
Max Time	757.0 ms	11218.0 ms	2044.0 ms	11436.0 ms
Complete Time	13.9 s	2.3 min	31.2 s	2.0 min

4.4.4 Update 1/10th of Rows By CLOBPayload with Indexes on *endtimestamp*

Setup To measure the update performance of a single row with no index the columns used in the WHERE clause, we update the column $ValueToUpdate$ by increasing it by one, for 100 random selected rows filtered by the column $CLOBPayload$ of a containing 1000 rows. A primary key (i.e. clustered index) is applied to ID and *starttimestamp* in the actual tables and history tables. An index was added on the column *endtimstamp* on the history tables of the separated and the hybrid approach, as well as on the actual table of the integrated approach.

Storage Size The results of our evaluation can be seen in Table 4.11. Besides the usual inaccuracies of the `SHOW TABLE STATUS` approach, we found nothing special to note.

Table 4.11: Update size after updating all rows of a given *section* 20 time - widh *endtimestamp* index, values in MiB if not otherwise stated

Measured Value	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**Result**	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
DT size	13.52	13.52	13.52	13.52	15.52	15.52	13.56	119.52
DT_Hist size	0.00	0.00	14.56	121.71	0.02	103.52	0.02	0.02
Endts. Index Size in KiB	0.00	0.00	48	208	0.00	192	48	208
# Rows for Endts Table	0	0	1000	7706	0	6706	1000	7706
Sum	13.52	13.52	28.08	135.04	15.54	119.04	13.58	119.54

Run-Time The results of our evaluation can be seen in Table 4.12. Having an index on the *endtimestamp* column improved the performance of the separated and integrated approach dramatically, compared to the measures we observed in 4.10.

Table 4.12: Update time of running 20 updates on all rows of a given *section* - with *endtimestamp* index

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
Avg Time	702.9 ms	2749.6 ms	1733.9 ms	2105.1 s
Min Time	599.0 ms	2476.0 ms	1459.0 ms	1915.0 ms
Max Time	792.0 ms	2999.0 ms	4033.0 ms	2314.0 ms
Complete Time	14.1 s	55.0 s	34.6 s	42.1 s

4.4.5 Updating the whole table 20 times - with index on *endtimestamp*

Setup To measure the update performance of the whole table, we update the column $ValueToUpdate$ by increasing it by one, twenty times for a table containing 1000 rows. A primary key (i.e. clustered index) is applied to ID and *starttimestamp* in the actual tables and history tables. An index was added on the column *endtimstamp* on the history tables of the separated and the hybrid approach, as well as on the actual table of the integrated approach.

Storage Size The results of our evaluation can be seen in Table 4.12. We have added the number of rows of the table that contains the index on the *endtimestamp* column as well as the size of the index (the column $INDEX\_LENGTH$ of the `SHOW TABLE STATUS` command was used). Note that the sizes stayed the same, expect for the initial setup at the separated approach, which unexplainably dropped.

Table 4.13: Update size of updating 1000 rows 20 times - with *endtimestamp* index, all values in MiB

Measured Value	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**Result**	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
DT size	13.52	13.52	13.52	13.52	15.52	15.52	13.56	326.98
DT_Hist size	0.00	0.00	14.52	326.97	0.02	315.02	0.02	0.02
Sum	13.52	13.52	28.03	340.49	15.54	330.04	13.56	327.00

Run-Time The results of our evaluation can be seen in Table 4.14. The update times show similar behavior to the other results we got (hybrid being the fastes, followed by integrated then by separated).

Table 4.14: Update size of updating 1000 rows 20 times - with *endtimestamp* index

Measured Values	Simple	Separated	Hybrid	Integrated
Avg Time	625.0 ms	4214.7 ms	2048.0 ms	3820.0 ms
Min Time	584.0 ms	3740.0 ms	1513.0 ms	3429.0 ms
Max Time	682.0 ms	4798.0 ms	3996.0 ms	5159.0 ms
Complete Time	12.5 s	84.3 s	40.1 s	76.4 s

4.4.6 Update Performance - Discussion

We want to sum up the most interesting points found in our evaluation of update performance.

Indexes We have found that the following indexes should be applied to our systems:

- Primary Key - each table, actual and historized, should have a primary key. If the column *starttimestamp* is part of the table, it needs to be added to the primary key.
- Index on the column *endtimestamp* - each table that has the column *endtimestamp* should index it.

Without those indexes, the separated and the integrated approach suffer significant performance hits.

Hybrid Updates are the fastest This can be seen especially in our last test run where we updated the whole table consisting of 1000 rows. Update times in the hybrid approach took around 3x the time of simple, integrated took around 6x the time of simple and separated around 6.7x the time of simple.

The reason for this is, that the update mechanic in the hybrid approach has a much simpler implementation than in the two other approaches. It can be summed up as:

- Insert tuples that satisfy the WHERE condition of the update into the history table with *endtimestamp* set
- Update the actual tuples and set their *starttimestamp*

In contrast, we had to use the following implementation in the integrated approach (the separated approach is similar):

- Create temporary table
- Set *endtimestamp* in the actual table

- Update temporary table with actual update and set *starttimestamp* in it
- Insert temporary table into actual table

We have chosen this implementation, as other approaches (setting the *endtimestamp* inside the table and inserting new updated rows) failed due to restrictions imposed by locking tables and transactions. We suggest reviewing this implementation if a more performant implementation can be found.

4.5 Delete Performance

We measure the update performance in the following four different ways:

- Delete every row by ID
- Delete every row by $Section$
- Delete all rows at once

4.5.1 Delete every row by ID

Setup To measure the delete performance of deleting a single row with an index on the column in the WHERE clause, we delete all rows by their ID. Our tests are run against a table with 1000 rows. Indexes have been applied as described in Subsection 4.4.6.

Storage Size We expect to achieve the following measures (note that we distinguish between the history and the actual tables here):

- $size(Separated) = 0$
- $size(Separated\_Hist) unchanged$
- $size(Integrated) unchanged$
- $size(Hybrid) = 0$
- $size(Hybrid\_Hist) = size(Separated\_Hist)$
- $size(Simple) = 0$

The results of our evaluation can be seen in Table 4.15. They conform, taking into account the inaccuracies of the used measuring method, to our expectations.

Table 4.15: Delete Size after deleting every row by id - sizes in MiB

Measured Value	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**Result**	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
DT size	13.52	0.02	13.52	0.02	15.52	0.02	13.56	14.56
DT_Hist size	0.00	0.00	14.57	14.56	0.03	13.58	0.02	0.02
Sum	13.52	0.04	28.09	14.56	15.55	13.58	13.58	14.58

Run-Time The results of our evaluation can be seen in Table 4.16. All in all, our approaches showed the same performance and each one took about double the time of the simple approach.

Table 4.16: Delete time of deleting every row by id

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
Avg Time	50.3 ms	104.7 ms	104.3 ms	102.6 ms
Min Time	44.0 ms	94.0 ms	92.0 ms	90.0 ms
Max Time	76.0 ms	252.0 ms	228.0 ms	183.0 ms
Complete Time	50.3 s	104.6 s	102.6 s	102.6 s

4.5.2 Delete every row by *Section*

Setup To measure the delete performance of deleting 10% of the rows at a given time, we delete rows of a given *Section*. Our tests are run against a table with 2000 rows, where the column *section* takes a value between 0 and 9. The whole evaluation was run five times. Indexes have been applied as described in Subsection 4.4.6.

Storage Size The evaluation for the storage size was omitted as it showed the same result as in Table 4.15.

Run-Time As our test setup contained 10 DELETE statements and was run five times, we have chosen to present the averages of every delete statement in Table 4.17.

There are a few interesting points to note:

- Hybrid and Separated take around double the time of simple, except for deleting everything (last row of the table)
- Integrated is on par with the simple approach, although this approach contains a overhead from the issued locking statements.. This may be due to the following reasons: only *endtimestamp* needs to be set to a value instead of actually deleting a row, the index on *endtimestamp* could have been utilized.

Table 4.17: Delete time of deleting every row by section - averages of deleting one *section* of a table 5 times

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
1st Delete Avg	1300.4 ms	2708.2 ms	2782.0 ms	1293.8 ms
2nd Delete Avg	1208.2 ms	2554.6 ms	2666.6 ms	1363.6 ms
3rd Delete Avg	1097.8 ms	2275.6 ms	2268 ms	1213.4 ms
4th Delete Avg	984.2 ms	1933.4 ms	1925.6 ms	1073.0 ms
5th Delete Avg	801.2 ms	1695.6 ms	1700.6 ms	904.0 ms
6th Delete Avg	683.4 ms	1483.8 ms	1514.2 ms	810.8 ms
7th Delete Avg	593.4 ms	1250.2 ms	1258.8 ms	663.8 ms
8th Delete Avg	419.2 ms	894.8 ms	947.2 ms	500.6 ms
9th Delete Avg	337.4 ms	695.6 ms	736.0 ms	355.2 ms
10th Delete Avg	113.0 ms	443.0 ms	416.8 ms	110.0 ms

4.5.3 Delete every row

Setup To measure the delete performance of deleting all rows of a table, we delete all rows of a table with 2000 rows, 5 times and look at the average it takes to complete. Indexes have been applied as described in Subsection 4.4.6.

Storage Size The evaluation for the storage size was omitted as it showed the same result as in Table 4.15.

Run-Time We examined the same behavior as in our last evaluation, as integrated is nearly as fast as simple approach and that hybrid and integrated take around 4x the time of simple. Compare those results here with the last row of Table 4.18.

Table 4.18: Delete time of deleting all rows of a table with 2000 rows 5 times

Measured Values	**Simple**	**Separated**	**Hybrid**	**Integrated**
Avg Time	1290.2 ms	3955.2 ms	3625.2 ms	1462.6 ms
Min Time	1258.0 ms	3798.0 ms	4296.0 ms	1339.0 ms
Max Time	1331.0 ms	4166.0 ms	3211.0 ms	1606.0 ms

4.5.4 Delete Performance - Discussion

We sum up here the results of our delete performance evaluation.

Deleting a single row We consider this to be the most important test as deleting a single row by key will probably be mainly used in real world applications. Here, we examined that all approaches took around double the time of the simple approach.

Deleting multiple rows Here, hybrid and separated took around double the time of simple, while integrated showed the same performance as simple. This is because the simple approach only needs to set a value for the column *endtimestamp* in one table.

Deleting all rows Here, hybrid and separated took around 4x the time of simple while integrated showed the same performance as simple. This is because the simple approach only needs to set a value for the column *endtimestamp* in one table.

4.6 Select Performance

For evaluating the performance of SELECT statements, we distinguish the following cases:

- SELECTs that need not be reproduced (i.e. return current data)
- SELECTs that need to be re-executed and their re-execution (querystore operations)

For the first case, executing a SELECT statement for the current data, we observe that the query only targets the actual table. As the actual table in the simple approach equals the actual table of the separated approach, which equals the hybrid actual table (except for the column *starttimestamp*) we can conclude that those three tables will show the same behavior. The only actual table that is different is the integrated table as it contains the *endtimestamp* column, which we should be indexed as we have shown in the update performance section, as well as it contains all the historized tuples. Therefore, we are interested in evaluating the following question: how much does storing the historized tuples in the integrated approach impact the query time of queries interested in the current active data?

For the second case, we are interested in how long does the initial query take (returning one row by ID or multiple via *Section*), compared to the simple approach and how much space is needed in our *querystore* table. For re-execution of queries, we are interested in the execution time of the following scenarios: re-execution without any changes to the data, re-execution after historized tuples have been created (inserted or updated), re-execution after all rows have been deleted.

4.6.1 Select performance to retrieve current active data

Setup To measure the SELECT performance of retrieving current active data, we selected 50 random rows identified by ID from the actual table which consisted of 100.000 rows. Afterwards, the not retrieved rows have been updated and the rows have been retrieved again. The aim of this evaluation is to show that the query time of actual data is the same for the simple, hybrid and separated approach. We expect the integrated approach to have the same query time in the first run. It is open how the integrated approach behaves after the table length nearly doubled. All SELECT statements were issued with the SQL_NO_CACHE modifier. Indexes have been applied as described in Subsection 4.4.6.

Storage Size Storage size is omitted as we got nothing new to measure here.

Run Time The results of our evaluation can be seen in Table 4.17. We want to especially mention the following points:

- The max value, was always the first value we retrieved
- All subsequent values retrieved later had been faster
- Removing the SQL_NO_CACHE made this small queries actually a little bit slower (around 10%).
- The high max value for the separated approach could not be reproduced in other runs, there, the separated max value was in accordance to the other retrieved values

We suspect that the first run of those queries was slower, as the used indexes were not loaded yet and that SSD caching was used in subsequent query calls. From this evaluation, we draw the following conclusion: **Well tuned queries (using indexes) for the current active rows, get no performance hit by using historization approaches**.

Table 4.19: Selecting 50 rows of 100.000 five times - Initially and after Updating all other rows - all values in ms

Measured Values	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Init**	**a.U.**	**Init**	**a.U.**	**Init**	**a.U.**	**Init**	**a.U.**
Avg Time	54.5	53.3	52.8	49.3	51.5	46.7	50.8	54.7
Min Time	48.0	47.0	45.0	44.0	42.0	43.0	47.0	47.0
Max Time	80.0	83.0	152.0	95.0	89.0	75.0	71.0	91.0

4.6.2 Querystore operations filtering on ID - without data changes

Setup To measure the querystore operation performance of executing a SELECT statement that retrieves one row and re-executing it, we select 10 random rows rows by ID from a table with 2000 rows. Indexes have been applied as described in Subsection 4.4.6. Note that this setup eliminates the overhead introduced by sorting.

Storage Size We are mainly interested in how much space is consumed in the table *querystore*. As we issue 10 SELECT statements that we want to re-execute later, 10 rows are added to the table *querystore*. The resulting sizes can be seen in Table 4.20. All systems needed the same space, which is about 27 kb per query. We want to put the 27 kb per query into context. For example, we could store about 38.836 queries in 1 GB of tablespace. It would take a year with around 106 queries daily to reach this size.

We want to note here that no optimizations were employed to reduce this space consumption. Some suggestions to reduce this storage space:

- All columns except: *OriginalQuery*, *ID*, *HASH*, *QueryExecutionTime* could be omitted, this would reduce the required storage at least of a factor 3.
- The original query is stored as a JSON string. The string could be stored compressed.
- A more storage efficient serialization like Protocol Buffers could be employed.
- Queries that have already been issued (having the same normal form and all tables have the same last update value) need not to be stored again.

Table 4.20: Size of the table *querystore* after storing 10 queries - values in KiB

Measured Value	**Separated**		**Hybrid**		**Integrated**	
Stage	**Initial**	**Result**	**Initial**	**Result**	**Initial**	**Result**
QueryStore	16.0	272.0	16.0	272.0	16.0	272.0

Run Time The results of our evaluation can be seen in table 4.21. Observing the results yield some interesting question we are going to cover.

Table 4.21: Querystore select run time of extracting 10 random rows by id and re-execution - all values in ms

Measured Values	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Exec.**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**
Avg Time	69.2	/	130.0	107.0	74.5	8331.4	79.6	216.1
Min Time	55.0	/	113.0	104.0	63.0	7585.0	64.0	212.0
Max Time	85.0	/	151.0	111.0	90.0	10932.0	96.0	227.0

Our first question is: *Why is re-execution of the hybrid approach so extremely slow?* For the re-execution, the hybrid approach has to consider the actual table and the historized table, as a tuple that was active at query execution time could be in either one of those tables. We rewrite our query, by replacing the tables in the FROM part, with the union of two select statements (one for the actual table and one for historized table). The query that was issued for re-execution is shown below.

```
SELECT   `t1`.`phasenumber`,
         `t1`.`id`,
         `t1`.`section`,
         `t1`.`valuetoupdate`,
         `t1`.`clobpayloade`
FROM     (
    SELECT `phasenumber`,
           `id`,
           `section`,
           `valuetoupdate`,
           `clobpayloade`,
           `startts`,
           NULL AS `endts`
    FROM   `qs_hybrid_093e580530ed4da09e199a720d61ba6e`.`datatable`
    UNION
    SELECT `phasenumber`,
           `id`,
           `section`,
           `valuetoupdate`,
           `clobpayloade`,
           `startts`,
           `endts`
    FROM   `qs_hybrid_093e580530ed4da09e199a720d61ba6e`.`datatable_1`)
    AS `t1`
WHERE    ((( `t1`.`startts` <= timestamp '2017-11-02 15:36:50.434')
         AND  ((timestamp '2017-11-02 15:36:50.434' < `t1`.`endts`)
                OR (t1`.`endts` IS NULL)))
         AND (((id) = 1))
         )
ORDER BY `t1`.`phasenumber` ASC,
         `t1`.`id` ASC,
         `t1`.`section` ASC,
         `t1`.`valuetoupdate` ASC,
         `t1`.`clobpayloade` ASC
```

The main problems of this query is the subselect in the FROM part containing the union. We thought about how this issue could be overcome:

- If no data is in the historized table, it could be omitted

- If the minimum *starttimetamp* in the actual table is bigger than the query time, the actual table could be omitted
- Using Common Table Expressions (CTEs, or WITH Queries) [6]
- Querying both separately and UNION the results in .Net
- Using Views (normal or materialized)
- Pushing the WHERE clauses into the subselects.

The first two optimization approaches cover corner cases. They therefore do not improve the performance in general. CTEs are, with the execption of recursive ones, syntactic sugar. They can not generate any performance gains. Creating the union of both queries in our code is feasible, if only one table is queried. However if multiple tables are queried, it will get messy with resolving the joins. One argument against this "optimization" is: RDMBS are optimized for retrieving and joining data; attempts to resolve the joins in our own framework will probably be significantly slower. Views do not improve performance and therefore fall into the same category as CTEs. If materialized views would be used, the materialized view would be the same as the history table of the separated approach, therefore the seperated approach should be used. Pushing the WHERE clauses into the subselects showed great performance improvements, nevertheless, as soon as JOINs come into play again, the HYBRID approach falls behind again significantly. This has been tested with the query above, by pushing the WHERE clause into the subselect statemetns, duplicating the table $t1$ again as $t2$ and joining both via $t1.id = t2.id$. This SELECT statement was issued in MySQLWorkbench and took around 1.5 seconds to complete. The same modified query could be answered by the separated approach in 0.000 seconds by MySQL Workbench. Changing the WHERE clause to retrieve values of a given $SECTION$, this modified query ran in around 3.5 seconds in the hybrid approach and 1.2 in the separated approach.

We conclude, that the Hybrid Historization Approach has inherent design problems, which result in infeasible completion time of the re-execution of stored queries.

Why is re-execution at separated faster than execution? As the result is only a few milliseconds difference, we suspect that no unknown caches are in place. As no rewriting needs to be done, and the rewritten query can be reused as it is (as it targeted the history table), some time is saved. Also, no locks etc. need to be acquired. Those few bits could add up for the few milliseconds that this approach is faster at re-execution. A detailed examination of all parts would be needed to answer this question with certainty.

[6]https://www.mysql.com/why-mysql/presentations/mysql%2D80%2Dcommon%2Dtable%2Dexpressions/

Why is the re-execution of integrated slower than the execution? When we re-execute a statement for the integrated approach, we have to check if the actual table still exists in the current schema. If it does not, the query needs to be rewritten to target the history table. This check is not needed for the separated approach (there, the rewriting is already done at the initial execution time, thus the initial execution time is higher than the time of the integrated approaches).

The last question that arises from the values in Table 4.21 is: *Why is the separated execution time slower than it is for integrated or hybrid?* The separated approach rewrites the query to target the history table. Therefore, before execution, a query for the actual schema needs to be done (it is later ensured, after locks have been acquired, that the schema has not changed). The rewriting for the integrated or hybrid approach does not need this rewriting, therefore the call is omitted which is reflected in the increased performance. Note, however, that the separated rewriting could be changed to target the actual table for the execution, thus reducing the call and getting it on par with the other two approaches. We want to note here, that we would not advice to optimize this in practice as the time for this call (around 50ms) should not be significant compared to the sorting needed on the resulting dataset, as we will see in the next evaluation.

4.6.3 Querystore operations filtering on Section - without data changes

Setup To measure the querystore operation performance of executing a select statement that returns multiple rows and re-executing it, we select all rows of a given *Section* from a table with 2000 rows, where the value for *Section* is between 0 and 9. We issue 10 statements, one for each section, and then we re-execute each statement. Indexes have been applied as described in Subsection 4.4.6. This evaluation shows, compared to the first evaluation, the impact of the applied sorting.

Storage Size We omitted the storage size as it showed the same result as our last query.

Run Time The results of our evaluation can be seen in Table 4.22. The observed execution times show the impact of the added *ORDERBY* clause. When one row was returned in our last evaluation, the overhead of our execution was about 50-60ms, now it nearly doubled. The overhead of the sorting is also seen at the hybrid approach which takes an additional second to complete. As we have mentioned in Section 3.7, we suggest tracking primary key integrity constraints and sorting by them instead of all columns. This should lower the overhead here.

We observed some system hickups in the separated and integrated runs, resulting in a high max value. Subsequent runs of our setup showed that those occurred randomly and are not associated with any section.

Table 4.22: Querystore select run time of extracting all rows of a section and re-execution - all values in ms

Measured Values	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Exec.**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**
Avg Time	1237.2	/	2622.6	3458.7	2832.8	9138.9	2986.0	3013.0
Min Time	1174.0	/	2571.0	2600.0	2516.0	8696.0	2325.0	2853.0
Max Time	1316.0	/	2707.0	6893.0	4971.0	9652.0	6735.0	3236.0

4.6.4 Querystore operations filtering on Section - all rows deleted before re-execution

Setup To measure the impact of delete operations on querystore re-execution performance, we issue a select statement that returns multiple rows, delete all rows and afterwards re-executing it. We select all rows of a given *Section* from a table with 2000 rows, where the value for *Section* is between 0 and 9. We issue 10 statements, one for each section, then delete all rows, and afterwards we re-execute each select statement. Indexes have been applied as described in Subsection 4.4.6.

Storage Size We omitted the storage size as it showed the same result as our last query.

Run Time The results of our evaluation can be seen in Table 4.23. They basically line up with the results in Table 4.22. Deleting rows sets the *endtimestamp* in the integrated and separated approaches. Deleting rows in the hybrid approach moves them from the actual table to the history table. This did not influence the extraction time.

Table 4.23: Querystore select run time of extracting all rows of a section and re-execution after all rows were deleted - all values in ms

Measured Values	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Exec.**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**
Avg Time	1269.4	/	2950.6	3090.0	2578.5	9418.5	2440.1	2850.6
Min Time	1175.0	/	2621.0	2684.0	2394.0	8877.0	2268.0	2714.0
Max Time	1401.0	/	5002.0	4526.0	2746.0	10592.0	2578.0	3083.0

4.6.5 Querystore operations filtering on Section - some sections updated multiple times before re-execution

To measure the impact of update operations on querystore re-execution performance, we issue a select statement that returns multiple rows, update 5 sections 5 times and afterwards re-execute the select statements. We select all rows of a given *Section* from a table with 2000 rows, where the value for *Section* is between 0 and 9. We issue 10 statements, one for each section, then update 5 different sections 5 times, and afterwards we re-execute the select statements. Indexes have been applied as described in Subsection 4.4.6. This evaluation shows, compared to the last one, the impact of a bigger history table on the re-execution time.

Storage Size We omitted the storage size as it showed the same result as our last query.

Run Time The results of our evaluation can be seen in Table 4.24. The history table of separated now contains 10090 rows and the actual table contains 2000 rows. The values for history and integrated can be inferred from that. We see again, why the hybrid approach should not be used for re-execution as the operations took around 1 minute compared to the 3.5 seconds of the other approaches. It is quite interesting comparing the values of the last evaluation in Table 4.23 with our current ones. Although 10090 rows have been added, which is around 5 times of the original size, the query time only increased by 1/6th for separated and integrated.

Table 4.24: Querystore select run time of extracting all rows of a section and re-execution after 5 sections have been updated 5 times - all values in ms

Measured Values	**Simple**		**Separated**		**Hybrid**		**Integrated**	
Stage	**Exec.**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**	**Exec**	**Reexec**
Avg Time	1154.1	/	2480.5	3443.2	2319.5	59007.1	2914.7	3340.3
Min Time	1048.0	/	2176.0	2802.0	2235.0	51603.0	2529.0	3059.0
Max Time	1247.0	/	2642.0	5744.0	2385.0	65217.0	4677.0	3789.0

4.7 Evaluation Summary

We summarize the most important results of our evaluation. The guidelines for choosing which historization approach to use in which scenario are based on this summaries.

Hybrid querystore re-execution time is unsustainable for a real world scenario As we have seen in the results in Table 4.24 and Table 4.21, *hybrid* performs terribly at re-execution. This is an inherent problem of the approach and we therefor do not advice to use this approach in practice. Therefore, for the rest of this summary, we do not consider the *hybrid* approach.

Insert performance *Separated* can perform inserts at about 2.5 times the speed of a system without historization and *integrated* needs around double the time.

Update performance Single row updates can be performed nearly at the same speed for *integrated* and *separated*. Those updates take around double the time of the *simple* approach. The more rows are updated, the slower it can be performed by *separated* and *integrated*. For example, updating 1000 rows 20 times could be done by *integrated* in 76.4 seconds, by *separated* in 84.3 seconds and by *simple* in 12.5 seconds (cf. 4.14).

Delete performance As the *integrated* approach only needs to set the value for the column *endtimestamp*, it is as fast as the *simple* approach. The separated approach needs around double the time of the simple approach for single row deletes. The more rows are deleted, the worse does *separated* perform. For example, deleting 2000 rows (the whole table) takes around 4 seconds compared to 1.3 for the *simple* approach.

Indexes and Primary Keys We suggest extending our system to at least track primary keys and to always add an index on the column *endtimestamp*. Tables that contain a *starttimestamp* column need to include this in their primary key. Without those indexes, our approaches become nearly unusable. Tracking this information also allows to implement more efficient sorting (compared to sorting by all columns).

Querystore Performance Currently, the *integrated* approach and the *separated* approach perform identical as they target a historized table. However, our *separated* approach could be changed to target the actual table in the initial select execution. As this table does not grow by updates, it should perform better than the historized table giving the *separated* approach an advantage. The initial queries take around double the time of queries issued by *simple*. This performance could be improved by employing better sorting strategies, exploiting primary key constraints.

Querystore Size Currently, we use around 27 KiB per query. This can be improved dramatically as described in the paragraph Storage Size in Subsection 4.6.2.

Database Size We can sum up the database size by employing the historization approaches as follows:

- DB size of separated approach is: $2 * DBSize(Simple) + HistorizedTuples$
- DB size of integrated approachn is: $DBSize(Simple) + HistorizedTuples$

Suggested improvements of our prototype

We advice the following improvements based on our evaluation to our prototype before using it in production:

- Implement Primary Key Tracking
- Implement automatically generating an index on *endtimestamp* columns

- Improve sorting by using primary keys
- Reduce querystore row size

4.8 Guidelines for Real World Applications

We want to point out a few guidelines that help to decide when to use which approach.

Hybrid We do not suggest to use the hybrid approach as the re-execution time of queries of the querystore is unacceptable.

Integrated The integrated approach performs nearly as good as the separated approach. The big advantage of it is, that it uses less space. Therefore, if space consumption is an issue, for example at mobile phones or applications running on Raspberry PIs, we suggest using this approach. If performance issues occur with SELECT statements interested in the current data, a materialized view could be employed with more indexes to serve the current active data faster.

Separated The big advantage of the separated approach is, that the actual data is separated from the historized data. Therefore, different indexing strategies can be employed on the table holding the actual data and the one holding the historized data. Consider the following scenario: the inserted data is updated very often. This results in a very long historized table compared to the actual table. Therefore, the indexes on the historized table will be bigger than the one on the actual table. At some point they might get too big and become slow. The ones at the actual table are still small and fast, and potentially more indexes could be added. This is the big advantage we see for the separated approach.

4.9 Summary

In this chapter, we evaluated the different historization approaches with respect to the additional storage size and query time they need, compared to a simple approach without data historization or query storage. The key findings were performance issues in the hybrid approaches query reexecution and the need for primary key tracking to provide good indexes to enable improved sorting and fixity calculation.

We ended with the usage guidelines for our historization approaches that can be summarized as: do not use the hybrid approach; use the integrated approach on systems that are sensitive to storage space such as mobile applications or embedded systems; use the separated approach for everything else.

We continue with our conclusions about the designed system and what future work we propose to address.

CHAPTER 5

Conclusions and Future Work

We have implemented a working solution for a data citeable, schema evolvable system for RDBMS that supports three different historization approaches.

The supported historization approaches using tuple-based-timestamping are: separated (storing all tuples in a historization table), hybrid (storing only already deleted data in historization table) and integrated (using one table with all tuples historized). Those models guarantee, that we know for each tuple when it was generated and deleted, allowing us to reconstruct the result of queries issued in the past. Schema evolution is supported by ten different schema modification operators that either modify columns, by either adding, dropping or renaming them, and table modification operators that handle creating, dropping and renaming a table, as well as vertical/horizontal merging or partitioning. By employing different locking mechanisms, we can guarantee that all issued queries can be reproduced at a later time.

The designed systems historization approaches have been evaluated on their query performance and storage size, compared to a system without historization, for the following use cases: selecting the current active data, reconstructing a historic query, INSERT/UPDATE/DELETE statement performance.

The key lessons we learned from our implementation and evaluation can be summarized as follows: Implementing the correct locking mechanisms to ensure that queries can be correctly reproduced depends on the SQL capabilities of the database. A mix of global locks, read/write locks on tables and optimistic locks were required in MySQL. Knowledge about the primary key for a data citeable system is critical to use as an index on the historized table, as well as using an index on the endtimestamp column. Having the information on which columns form the primary key, allows the query rewriting to use more efficient stable sorting as well as more efficient fixity calculation. The hybrid approach needs to UNION the tuples of the current active table and the historized table to reproduce the result of historic queries. Having to rely on UNION introduces a big performance hit on historic query re-execution.

Based on our evaluation, we developed usage guidelines on when to use which historization approach. The hybrid approach should not be used as historic query execution time is significantly slower than those of the integrated and the separated approach. The integrated approach should be used in scenarios where storage size is important, for example, applications that run on mobile phones or Raspberry PIs. The separated approach should be used in all other scenarios. In simple terms, if there are no good reasons to use the hybrid or integrated approach, one should default to the separated approach.

The implemented system differs from the reference implementation for data citation[PR13] by the support of different historization approaches (integrated, separated and hybrid) and by the native support of schema evolution through SMOs. The main questions that were raised by the reference implementation, schema evolvability and when to use which historization approach, have been answered by our system and evaluation. The big difference of our implemented systems to the data historization approaches PRIMA[MCD+08] and AIMS[MCZ10] is the added query store facility and the required locking mechanisms that ensures isolation and atomicity of query execution and storage. The design of those systems would not allow them to be easily extended with such a locking mechanism. Due to those differentiations, we see that the developed system and the proposed one for big database as an advancement of the current state of the art of data citeable and schema evolvable systems.

We would love to see our system improved by tracking primary keys and having the sorting and fixity calculations improved. Also, our system is probably not suitable for databases with tables that contain million of rows due to the added historization storage size of SMOs. A solution has been proposed in Section 3.8, that is based on historic tuple migration between historized tables as well as the query migrations that need to be employed to guarantee reproducability. The problem with the outlined system is, that historic query rewriting is mainly based on UNIONs, which have a considerable performance loss as seen in our evaluation. We are looking forward to see the outlined system thoroughly designed, implemented and evaluated against our current solution. Based on this, the usage guidelines can be improved to help database owners decide which historization approach (hybrid/separated/integrated) combined with which tuple historization approach (duplicating, as we have implemented it vs. tuple migration as outlined) they should use.

List of Figures

List of Tables

List of Algorithms

Bibliography

[CMDZ13] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *Very Large Databases Journal*, 22(1):73–98, 2013.

[CMHZ09] Carlo Curino, Hyun Jin Moon, MyungWon Ham, and Carlo Zaniolo. The PRISM workwench: Database schema evolution without tears. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 1523–1526, 2009.

[CMZ08] Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772, 2008.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[JS99] Christian S. Jensen and Richard T. Snodgrass. Temporal data management. *IEEE Trans. Knowl. Data Eng.*, 11(1):36–44, 1999.

[KM12] Krishna G. Kulkarni and Jan-Eike Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[MCD$^+$08] Hyun Jin Moon, Carlo Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proceedings of the VLDB Endowment*, 1(1):882–895, 2008.

[MCZ10] Hyun Jin Moon, Carlo Curino, and Carlo Zaniolo. Scalable architecture and query optimization fortransaction-time dbs with evolving schemas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 207–218, 2010.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008.

[PR13] Stefan Pröll and Andreas Rauber. Scalable data citation in dynamic, large databases: Model and reference implementation. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 307–312, 2013.

[RAvP16] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Proell. Identification of reproducible subsets for data citation, sharing and re-use. *Bulletin of IEEE Technical Committee on Digital Libraries (TCDL)*, 12, 5 2016.

[RAvUP15] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Pröll. Data citation of evolving data recommendations of the working group on data citation (wgdc). `https://www.rd-alliance.org/system/files/documents/RDA-DC-Recommendations_151020.pdf`, 2015.

[Rod95] John F. Roddick. A survey of schema versioning issues for database systems. *Information & Software Technology*, 37(7):383–393, 1995.

[SA86] Richard T. Snodgrass and Ilsoo Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

Appendix A - Code Examples

The following code samples have been extracted from the automated tests of our implementation. The schemata used in them alter between the examples as each test uses an automatically generated unique schema.

```sql
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Separated4355776dda15402aa23054f430f9e197.employees
 ↪  WRITE,`separated4355776dda15402aa23054f430f9e197`.`employees_metadata`
 ↪  WRITE,`separated4355776dda15402aa23054f430f9e197`.`employees_1`
 ↪  WRITE,`Separated4355776dda15402aa23054f430f9e197`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Separated4355776dda15402aa23054f430f9e197`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @insertTime = NOW(3)
INSERT INTO `separated4355776dda15402aa23054f430f9e197`.`employees` (
`id`,
`name`,
`job`) VALUES (
1,
'John',
'Developer')
INSERT INTO `separated4355776dda15402aa23054f430f9e197`.`employees_1`
 ↪  (
`id`,
`name`,
`job`,
`startts`,
`endts`) VALUES (
1,
'John',
'Developer',
@insertTime,
null)
```

```
UPDATE
 ↪  `Separated4355776dda15402aa23054f430f9e197`.`employees_metadata`
 ↪  SET lastUpdate = @insertTime;
COMMIT;
UNLOCK TABLES;
```

Listing 2: Separated Approach - Insert Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Separated2e1075aa55e04641b26efa06b96c3062.employees
 ↪  WRITE,`separated2e1075aa55e04641b26efa06b96c3062`.`employees_metadata`
 ↪  WRITE,`separated2e1075aa55e04641b26efa06b96c3062`.`employees_1`
 ↪  WRITE,`separated2e1075aa55e04641b26efa06b96c3062`.`employees_1`
 ↪  AS `updtTable`
 ↪  WRITE,`Separated2e1075aa55e04641b26efa06b96c3062`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Separated2e1075aa55e04641b26efa06b96c3062`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @updateTime = NOW(3)

CREATE TEMPORARY TABLE IF NOT EXISTS
 ↪  `separated2e1075aa55e04641b26efa06b96c3062`.`tmptable` AS (SELECT
`updtTable`.`*`
FROM `Separated2e1075aa55e04641b26efa06b96c3062`.`employees_1` AS
 ↪  `updtTable`
WHERE ((`updtTable`.`endts` IS NULL) AND ((`updtTable`.`ID` = 1)))
);

UPDATE `separated2e1075aa55e04641b26efa06b96c3062`.`employees_1` SET
 ↪  `endts` = @updateTime WHERE ((`employees_1`.`endts` IS NULL) AND
 ↪  ((`employees_1`.`ID` = 1)));

UPDATE `separated2e1075aa55e04641b26efa06b96c3062`.`tmptable` SET
 ↪  `Name` = 'McJohn',
`startts` = @updateTime ;
INSERT INTO `separated2e1075aa55e04641b26efa06b96c3062`.`employees_1`
 ↪  SELECT
`tmp`.`*`
FROM `Separated2e1075aa55e04641b26efa06b96c3062`.`tmpTable` AS `tmp`



DROP TEMPORARY TABLE
 ↪  `separated2e1075aa55e04641b26efa06b96c3062`.`tmptable`;
```

```
UPDATE `separated2e1075aa55e04641b26efa06b96c3062`.`employees` SET
 ↪  `Name` = 'McJohn' WHERE ((`employees`.`ID` = 1))
UPDATE
 ↪  `Separated2e1075aa55e04641b26efa06b96c3062`.`employees_metadata`
 ↪  SET lastUpdate = @updateTime;
COMMIT;
UNLOCK TABLES;
```

Listing 3: Separated Approach - Update Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Separated74d431c4997d4e14afb22f2c726560bf.employees
 ↪  WRITE,`separated74d431c4997d4e14afb22f2c726560bf`.`employees_metadata`
 ↪  WRITE,`separated74d431c4997d4e14afb22f2c726560bf`.`employees_1`
 ↪  WRITE,`Separated74d431c4997d4e14afb22f2c726560bf`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Separated74d431c4997d4e14afb22f2c726560bf`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @deleteTime = NOW(3)
DELETE FROM `separated74d431c4997d4e14afb22f2c726560bf`.`employees`
 ↪  WHERE ((`employees`.`ID` = 1) AND (`employees`.`Name` = 'John'))
UPDATE `separated74d431c4997d4e14afb22f2c726560bf`.`employees_1` SET
 ↪  `endts` = @deleteTime WHERE ((`employees_1`.`endts` IS NULL) AND
 ↪  ((`employees_1`.`ID` = 1) AND (`employees_1`.`Name` = 'John')))
UPDATE
 ↪  `Separated74d431c4997d4e14afb22f2c726560bf`.`employees_metadata`
 ↪  SET lastUpdate = @deleteTime;
COMMIT;
UNLOCK TABLES;
```

Listing 4: Separated Approach - Delete Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Integrated571d327f028b4dc6b254866d6132a435.employees
 ↪  WRITE,`integrated571d327f028b4dc6b254866d6132a435`.`employees_metadata`
 ↪  WRITE,`integrated571d327f028b4dc6b254866d6132a435`.`employees_1`
 ↪  WRITE,`Integrated571d327f028b4dc6b254866d6132a435`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Integrated571d327f028b4dc6b254866d6132a435`.`employees_metadata`
```

```
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @insertTime = NOW(3)
INSERT INTO `integrated571d327f028b4dc6b254866d6132a435`.`employees`
 ↪  (
`id`,
`name`,
`job`,
`startts`,
`endts`) VALUES (
1,
'John',
'Developer',
@insertTime,
null)
UPDATE
 ↪  `Integrated571d327f028b4dc6b254866d6132a435`.`employees_metadata`
 ↪  SET lastUpdate = @insertTime;
COMMIT;
```

Listing 5: Integrated Approach - Insert Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;

LOCK TABLES Integratedce63e84e332546f096ad3ef1a4dc7b90.employees
 ↪  WRITE,`integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees_metadata`
 ↪  WRITE,`integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees_1`
 ↪  WRITE,`Integratedce63e84e332546f096ad3ef1a4dc7b90`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees_metadata`

-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @updateTime = NOW(3)

CREATE TEMPORARY TABLE IF NOT EXISTS
 ↪  `integratedce63e84e332546f096ad3ef1a4dc7b90`.`tmptable` AS
 ↪  (SELECT
`employees`.`*`
FROM `Integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees` AS
 ↪  `employees`

WHERE ((`employees`.`startts` < @updateTime) AND (`employees`.`endts`
 ↪  IS NULL) AND ((`employees`.`ID` = 1)))
```

```sql
);


UPDATE `integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees` SET
 ↪  `endts` = @updateTime WHERE ((`employees`.`startts` <
 ↪  @updateTime) AND (`employees`.`endts` IS NULL) AND
 ↪  ((`employees`.`ID` = 1)));
INSERT INTO `integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees`
 ↪  SELECT
`tmp`.`*`
FROM `Integratedce63e84e332546f096ad3ef1a4dc7b90`.`tmpTable` AS `tmp`


UPDATE `integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees` SET
 ↪  `Name` = 'McJohn',
`startts` = @updateTime WHERE ((`employees`.`startts` < @updateTime)
 ↪  AND (`employees`.`endts` IS NULL) AND ((`employees`.`ID` = 1)));

DROP TEMPORARY TABLE
 ↪  `integratedce63e84e332546f096ad3ef1a4dc7b90`.`tmptable`;

UPDATE
 ↪  `Integratedce63e84e332546f096ad3ef1a4dc7b90`.`employees_metadata`
 ↪  SET lastUpdate = @updateTime;

COMMIT;
UNLOCK TABLES;
```

Listing 6: Integrated Approach - Update Handling

```sql
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Integratedce324cfdad544ab5aed71e7d6d5d7e44.employees
 ↪  WRITE,`integratedce324cfdad544ab5aed71e7d6d5d7e44`.`employees_metadata`
 ↪  WRITE,`integratedce324cfdad544ab5aed71e7d6d5d7e44`.`employees_1`
 ↪  WRITE,`Integratedce324cfdad544ab5aed71e7d6d5d7e44`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Integratedce324cfdad544ab5aed71e7d6d5d7e44`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @deleteTime = NOW(3)
UPDATE `integratedce324cfdad544ab5aed71e7d6d5d7e44`.`employees` SET
 ↪  `endts` = @deleteTime WHERE ((`employees`.`ID` = 1) AND
 ↪  (`employees`.`Name` = 'John'))
```

```
UPDATE
 ↪  `Integratedce324cfdad544ab5aed71e7d6d5d7e44`.`employees_metadata`
 ↪  SET lastUpdate = @deleteTime;
COMMIT;
UNLOCK TABLES;
```

Listing 7: Integrated Approach - Delete Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Hybrid_3f7432ee280f45db98efd75e635a0123.employees
 ↪  WRITE,`hybrid_3f7432ee280f45db98efd75e635a0123`.`employees_metadata`
 ↪  WRITE,`Hybrid_3f7432ee280f45db98efd75e635a0123`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Hybrid_3f7432ee280f45db98efd75e635a0123`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @insertTime = NOW(3)
INSERT INTO `hybrid_3f7432ee280f45db98efd75e635a0123`.`employees` (
`id`,
`name`,
`job`,
`startts`) VALUES (
1,
'asdf',
null,
@insertTime)
UPDATE `Hybrid_3f7432ee280f45db98efd75e635a0123`.`employees_metadata`
 ↪  SET lastUpdate = @insertTime;
COMMIT;
UNLOCK TABLES;
```

Listing 8: Hybrid Approach - Insert Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Hybrid_45895aee57ff48ef97552279ee130f99.employees

     ↪  WRITE,`hybrid_45895aee57ff48ef97552279ee130f99`.`employees_metadata`
    WRITE,`hybrid_45895aee57ff48ef97552279ee130f99`.`employees_1`
    WRITE,`Hybrid_45895aee57ff48ef97552279ee130f99`.`QubaDCSMOTable`
     ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
```

```
SELECT canBeQueried FROM
 ↪  `Hybrid_45895aee57ff48ef97552279ee130f99`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @updateTime = NOW(3)
INSERT INTO `hybrid_45895aee57ff48ef97552279ee130f99`.`employees_1`
 ↪  SELECT
`employees`.`*`, @updateTime AS `ut`
FROM `Hybrid_45895aee57ff48ef97552279ee130f99`.`employees` AS
 ↪  `employees`

WHERE ((`employees`.`startts` < @updateTime) AND ((`employees`.`ID` =
 ↪  1)))

UPDATE `hybrid_45895aee57ff48ef97552279ee130f99`.`employees` SET
 ↪  `Name` = 'asdfxyz',
`startts` = @updateTime WHERE ((`employees`.`startts` < @updateTime)
 ↪  AND ((`employees`.`ID` = 1)));
UPDATE `Hybrid_45895aee57ff48ef97552279ee130f99`.`employees_metadata`
 ↪  SET lastUpdate = @updateTime;
COMMIT;
UNLOCK TABLES;
```

Listing 9: Hybrid Approach - Update Handling

```
SET autocommit=0;
SET SQL_SAFE_UPDATES=0;
LOCK TABLES Hybrid_fbeab32e76be4f389a16e6949736b820.employees
 ↪  WRITE,`hybrid_fbeab32e76be4f389a16e6949736b820`.`employees_metadata`
 ↪  WRITE,`hybrid_fbeab32e76be4f389a16e6949736b820`.`employees_1`
 ↪  WRITE,`Hybrid_fbeab32e76be4f389a16e6949736b820`.`QubaDCSMOTable`
 ↪  READ;
-- C# ensuring hist table has not changed since statement was
 ↪  rewritten
SELECT canBeQueried FROM
 ↪  `Hybrid_fbeab32e76be4f389a16e6949736b820`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if it cannot be
 ↪  queried
SET @updateTime = NOW(3)
INSERT INTO `hybrid_fbeab32e76be4f389a16e6949736b820`.`employees_1`
 ↪  SELECT
`employees`.`*`, @updateTime AS `ut`
FROM `Hybrid_fbeab32e76be4f389a16e6949736b820`.`employees` AS
 ↪  `employees`

WHERE ((`employees`.`startts` < @updateTime) AND ((`employees`.`ID` =
 ↪  1) AND (`employees`.`Name` = 'asdf')))
```

```
DELETE FROM `hybrid_fbeab32e76be4f389a16e6949736b820`.`employees`
 ↪  WHERE ((`employees`.`startts` < @updateTime) AND
 ↪  ((`employees`.`ID` = 1) AND (`employees`.`Name` = 'asdf')))
UPDATE `Hybrid_fbeab32e76be4f389a16e6949736b820`.`employees_metadata`
 ↪  SET lastUpdate = @updateTime;
COMMIT;
UNLOCK TABLES;
```

Listing 10: Hybrid Approach - Delete Handling

```
--Input: Create Table Statement
{
   "TableName":"employees",
   "Schema":"HybridTests_bea80e0a53a14bc091b28b343886d859",
   "Columns":[
      {
         "ColumnName":"ID",
         "DataType":" INT",
         "Nullable":false,
         "AdditionalInformation":null
      },
      {
         "ColumnName":"Name",
         "DataType":" MediumText",
         "Nullable":false,
         "AdditionalInformation":null
      },
      {
         "ColumnName":"Job",
         "DataType":" MediumText",
         "Nullable":true,
         "AdditionalInformation":null
      }
   ],
   "PrimaryKey":[

   ]
}
```

Listing 11: Hybrid Approach - Create Table SMO as JSON

Input JSON from Listing 11, used as serialized SMO.

```
SET autocommit=0;
SELECT GET_LOCK('SMO UPDATES',10);
SET @updateTime = NOW(3);
CREATE TABLE
 ↪  `HybridTests_bea80e0a53a14bc091b28b343886d859`.`employees` (
`ID`  INT NOT NULL ,
`Name`  MediumText NOT NULL ,
`Job`  MediumText NULL ,
`startts` DATETIME(3) NOT NULL

);

CREATE TABLE
 ↪  `HybridTests_bea80e0a53a14bc091b28b343886d859`.`employees_1` (
`ID`  INT NOT NULL ,
`Name`  MediumText NOT NULL ,
`Job`  MediumText NULL ,
`startts` DATETIME(3) NOT NULL ,
`endts` DATETIME(3) NULL

);

 CREATE TABLE
  ↪  `HybridTests_bea80e0a53a14bc091b28b343886d859`.`employees_metadata`
  ↪  (
  `lastUpdate` datetime(3) NOT NULL,
  `canBeQueried` BOOL NOT NULL
);
INSERT INTO
 ↪  `HybridTests_bea80e0a53a14bc091b28b343886d859`.`employees_metadata`
(`lastUpdate`,
`canBeQueried`)
VALUES
(@updateTime,
true);
INSERT INTO
 ↪  `HybridTests_bea80e0a53a14bc091b28b343886d859`.`qubadcsmotable`
(`Schema`,
`SMO`,
`Timestamp`)
VALUES(
'serialzed JSON Schema - ommited as too long',
'serialized SMO'
@updateTime
);
COMMIT;
SELECT RELEASE_LOCK('SMO UPDATES');
```

Listing 12: Hybrid Approach - Create Table SMO Execution Script

```
{
   "TableName":"employees",
   "Schema":"Hybrid_b1df1726cb3240ab9b05a630c981b5df",
   "Column":{
      "ColumnName":"NewSchema",
      "DataType":" MediumText",
      "Nullable":false,
      "AdditionalInformation":null
   },
   "InitalValue":"CONCAT('new',`Name`)"
}
```

Listing 13: Hybrid Approach - Add Column SMO as JSON

Input JSON from Listing 13, used as serialized SMO.

```
SET autocommit=0;
SELECT GET_LOCK('SMO UPDATES',10);
LOCK TABLES
 ↪  `hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_metadata`
 ↪  WRITE;
UPDATE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_metadata`
 ↪  SET canBeQueried = false;
COMMIT;
UNLOCK TABLES;
SET @updateTime = NOW(3);
RENAME TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees` TO
 ↪  `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_old`
CREATE TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees` AS
 ↪  SELECT
`t1`.`ID`, `t1`.`Name`, `t1`.`Job`
FROM `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_old` AS
 ↪  `t1`

WHERE (1 = 2)
;
INSERT INTO `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_1`
 ↪  (`ID`, `Name`, `Job`, `startts`, `endts`) SELECT
`t1`.`*`, @updateTime AS `ut`
FROM `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_old` AS
 ↪  `t1`


;
ALTER TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees` ADD
 ↪  `NewSchema`  MediumText NOT NULL
ALTER TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees` ADD
 ↪  `startts` DATETIME(3) NOT NULL
INSERT INTO `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees`
 ↪  (`ID`, `Name`, `Job`, `NewSchema`, `startts`) SELECT
`t1`.`ID`, `t1`.`Name`, `t1`.`Job`, CONCAT('new',`Name`) AS
 ↪  `NewSchema`, @updateTime AS `ut`
```

```
FROM `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_old` AS
 ↪  `t1`


;
CREATE TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_2`
 ↪  AS SELECT
`t1`.`ID`, `t1`.`Name`, `t1`.`Job`
FROM `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees` AS `t1`

WHERE (1 = 2)
;
ALTER TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_2`
 ↪  ADD  `NewSchema`  MediumText NOT NULL
ALTER TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_2`
 ↪  ADD  `startts` DATETIME(3) NOT NULL
ALTER TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_2`
 ↪  ADD  `endts` DATETIME(3) NULL
DROP TABLE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_old`
UPDATE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_metadata`
 ↪  SET lastUpdate = @updateTime;
INSERT INTO
 ↪  `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`qubadcsmotable`
(`Schema`,
`SMO`,
`Timestamp`)
VALUES(
'serialzed JSON Schema - ommited as too long',
'serialized SMO',
@updateTime
);
UPDATE `Hybrid_b1df1726cb3240ab9b05a630c981b5df`.`employees_metadata`
 ↪  SET canBeQueried = true;
COMMIT;
SELECT RELEASE_LOCK('SMO UPDATES');
```

Listing 14: Hybrid Approach - Add Column Table SMO Execution Script

```
{
   "ResultTableName":"mergedtable",
   "ResultSchema":"Hybrid_ff014d9c06254471985ccca6d5ee4617",
   "FirstTableName":"employees",
   "FirstSchema":"Hybrid_ff014d9c06254471985ccca6d5ee4617",
   "SecondTableName":"basictable2",
   "SecondSchema":"Hybrid_ff014d9c06254471985ccca6d5ee4617"
}
```

Listing 15: Hybrid Approach - Merge Table SMO as JSON

Input: JSON from Listing 15, used as serialized SMO.

```
SET autocommit=0;
SELECT GET_LOCK('SMO UPDATES',10);
LOCK TABLES
 ↪  `hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees_metadata`
 ↪  WRITE,`hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2_metadata`
 ↪  WRITE;
UPDATE `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees_metadata`
 ↪  SET canBeQueried = false;
UPDATE
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2_metadata`
 ↪  SET canBeQueried = false;
COMMIT;
UNLOCK TABLES;
SET @updateTime = NOW(3);
CREATE TABLE `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`mergedtable`
 ↪  LIKE `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees`;
CREATE TABLE
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`mergedtable_3` LIKE
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees_1`;
 CREATE TABLE
  ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`mergedtable_metadata`
  ↪  (
  `lastUpdate` datetime(3) NOT NULL,
  `canBeQueried` BOOL NOT NULL
);
INSERT `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`mergedtable`
 ↪  SELECT `ID`, `Name`, `Job`, @updateTime FROM
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees` ;
INSERT `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`mergedtable`
 ↪  SELECT `ID`, `Name`, `Job`, @updateTime FROM
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2` ;
INSERT INTO `hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees_1`
 ↪  SELECT
`t1`.`*`, @updateTime AS `ut`
FROM `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees` AS `t1`



INSERT INTO `hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2_2`
 ↪  SELECT
`t1`.`*`, @updateTime AS `ut`
FROM `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2` AS `t1`



DROP TABLE `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees`
DROP TABLE `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2`
DROP TABLE
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`employees_metadata`
DROP TABLE
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`basictable2_metadata`
INSERT INTO
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`mergedtable_metadata`
(`lastUpdate`,
`canBeQueried`)
VALUES
(@updateTime,
true);
INSERT INTO
 ↪  `Hybrid_ff014d9c06254471985ccca6d5ee4617`.`qubadcsmotable`
(`Schema`,
`SMO`,
`Timestamp`)
VALUES(
'serialzed JSON Schema - ommited as too long',
'serialized SMO',
@updateTime
);
COMMIT;
SELECT RELEASE_LOCK('SMO UPDATES');
```

Listing 16: Hybrid Approach - Add Column Table SMO Execution Script

```
{
 "Tables":[
  {
   "Table":{
    "Name":"employees",
    "Schema":"HybridTests_9cd3ca6515f14b7dafc62f59a1161632",
    "Columns":[
     "ID",
     "Name",
     "Job"
    ],
    "ColumnDefinitions":[
     {
      "ColumnName":"ID",
      "DataType":" INT",
      "Nullable":false,
      "AdditionalInformation":null
     },
     {
      "ColumnName":"Name",
      "DataType":" MediumText",
      "Nullable":false,
          "AdditionalInformation":null
        },
        {
          "ColumName":"Job",
          "DataType":" MediumText",
          "Nullable":true,
          "AdditionalInformation":null
        }
      ],
      "AddTimeSetGuid":"0fb19c71-73e8-4d74-8adb-3924058d4dd7"
    },
    "HistTableName":"employees_1",
    "HistTableSchema":"HybridTests_9cd3ca6515f14b7dafc62f59a1161632",
    "MetaTableName":"employees_metadata",
    "MetaTableSchema":"HybridTests_9cd3ca6515f14b7dafc62f59a1161632"
  }
],
"HistTables":[
  {
    "Name":"employees_1",
    "Schema":"HybridTests_9cd3ca6515f14b7dafc62f59a1161632",
    "Columns":[
      "ID",
      "Name",
      "Job",
      "startts",
      "endts"
    ],
    "ColumnDefinitions":[
      {
        "ColumName":"ID",
        "DataType":" INT",
        "Nullable":false,
        "AdditionalInformation":null
      },
      {
        "ColumName":"Name",
        "DataType":" MediumText",
        "Nullable":false,
        "AdditionalInformation":null
      },
      {
        "ColumName":"Job",
        "DataType":" MediumText",
        "Nullable":true,
        "AdditionalInformation":null
      },
      {
```

```
      "ColumnName":"startts",
      "DataType":"DATETIME(3)",
      "Nullable":false,
      "AdditionalInformation":null
    },
    {
      "ColumnName":"endts",
      "DataType":"DATETIME(3)",
      "Nullable":true,
      "AdditionalInformation":null
    }
  ],
  "AddTimeSetGuid":null
 }
]
}
```

Listing 17: Example Schema serialized as JSON

```
SET autocommit=0;
SET sql_safe_updates=0;
LOCK tables
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees` AS
 ↪  employees_ref READ,
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees_metadata` READ,
`hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`qubadcsmotable` READ;

SELECT canbequeried
FROM   `hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees_metadata`
-- C# Code checking canBeQueried, throwing exception if one is false
SET @updateTime = Now(3);
SELECT @updateTime
-- C# Selecting Updatetime as it will be renderd into the select
 ↪  statements
SELECT   `id` ,
         `SCHEMA` ,
         `smo` ,
         `timestamp`
FROM     `hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.qubadcsmotable
ORDER BY id DESC limit 0,
         1
-- C# ensuring that the schema has not changed since generating this
 ↪  script
SELECT   md5(
                group_concat(
                Concat_ws(
                 '#',Md5(`employees_ref`.`id`),
                  ↪  Md5(`employees_ref`.`NAME`),
                  ↪  Md5(`employees_ref`.`job`)
                 ) separator '#'
                 )
                 )
FROM      `hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees` AS
 ↪  `employees_ref`
WHERE     (( `employees_ref`.`startts` <= timestamp '2017-10-18
 ↪  00:18:01.074'))
ORDER BY `employees_ref`.`id` ASC,
         `employees_ref`.`NAME` ASC,
         `employees_ref`.`job` ASC

SELECT    `employees_ref`.`id`,
          `employees_ref`.`NAME`,
          `employees_ref`.`job`
FROM      `hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees` AS
 ↪  `employees_ref`
WHERE     ((
          `employees_ref`.`startts` <= timestamp '2017-10-18
           ↪  00:18:01.074'))
ORDER BY `employees_ref`.`id` ASC,
         `employees_ref`.`NAME` ASC,
         `employees_ref`.`job` ASC

unlock tables;

INSERT INTO `hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`querystore`
            (
            `query`,
            `queryserialized`,
            `rewrittenquery`,
            `rewrittenqueryserialized`,
            `timestamp`,
            `hash`,
            `hashselect`,
            `hashselectserialized`,
            `guid`,
            `additionalinformation`
            )
            VALUES
            (
'SELECT `employees_ref`.`ID`, `employees_ref`.`Name`,
 ↪  `employees_ref`.`Job`
 FROM `Hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees` AS
 ↪  `employees_ref`
 ORDER BY `employees_ref`.`ID` ASC, `employees_ref`.`Name` ASC,
 ↪  `employees_ref`.`Job` ASC' ,
'JSON serialized version of the select statement' ,
'SELECT `employees_ref`.`ID`, `employees_ref`.`Name`,
 ↪  `employees_ref`.`Job`
FROM `Hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees` AS
 ↪  `employees_ref`
WHERE ((`employees_ref`.`startts` <= TIMESTAMP \'2017-10-18
 ↪  00:18:01.074\'))
ORDER BY `employees_ref`.`ID` ASC, `employees_ref`.`Name` ASC,
 ↪  `employees_ref`.`Job` ASC' ,
'JSON serialized version of the rewrittenquery`' ,
timestamp '2017-10-18 00:17:47.000' ,
'98dec3754faa19997a14b0b27308bb63' ,
'SELECT MD5( GROUP_CONCAT( CONCAT_WS(\'#\',MD5(`employees_ref`.`ID`),
 ↪
MD5(`employees_ref`.`Name`), MD5(`employees_ref`.`Job`)) SEPARATOR
 ↪  \'#\' ) )
FROM `Hybrid_1965bfa0ed4a4d519a49338e1d9d956e`.`employees` AS
 ↪  `employees_ref`
WHERE ((`employees_ref`.`startts` <= TIMESTAMP \'2017-10-18
 ↪  00:18:01.074\'))
ORDER BY `employees_ref`.`ID` ASC, `employees_ref`.`Name` ASC,
 ↪  `employees_ref`.`Job` ASC' ,
'JSON serialized version of rewritten hashquery' ,
'b67e2545-7d23-4c93-a0e4-c4ea07c76f05' ,
'JSON serialized Dictionary of GUIDs of queried tables '
            );
COMMIT;
```

Listing 18: Example Query Store handling of a query on employees