Non-blocking Lazy Schema Changes in Multi-Version Database Management Systems

Yangjun Sheng

CMU-CS-19-104
May 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andy Pavlo, Chair
Nathan Beckmann

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Copyright © 2019 Yangjun Sheng

Keywords: DBMS, schema change, multi-version, non-blocking, concurrency control, catalog, index, transaction

For graduation. To my family.

Abstract

The relational schema of a table in a database management system (DBMS) describes its logical attribute information and constraints. Despite the aim of separation between logical schema and physical data storage, in practice, the schema often dictates how a DBMS organizes data on disk or in memory. This tight coupling is because the database's physical schema must match its logical schema. The problem with this is that applications that incur frequent schema changes (e.g., add a column, change column type) may become slower or even unavailable during a change due to data migration. A better approach is to support non-blocking schema changes by storing multiple versions of tables and allow data migration happens lazily.

In this thesis, we introduce multi-version schemas that are based on multi-version concurrency control policies (MVCC) to support fast online schema changes. This approach maintains multiple tables of different schemas and allows transactions to see the correct versions of tuples. It migrates tuples from old schema to new schema lazily on demand. We show that multi-version schemas achieve non-blocking schema changes. We also show that the overhead of maintaining multiple schemas is small and the system can recover from performance degeneration caused by schema change fast when there is hotspot in the database.

Acknowledgments

First and foremost, I would like to give huge thanks to my advisor Andy Pavlo for working on this project with me. Thank you for the opportunity for taking me as your student last year. Thank you for supporting me and giving me helpful technical advice and life advice. Your guidance has been a valuable input for this thesis. Also, thank you for being an amazing and funny advisor and professor.

I gratefully acknowledge the help of John Rollinson, Yashwanth Nannapaneni, and Kiriti Sai with this thesis. I would like to thank all people in the CMU DB group for your hard work on the system. None of this would have been possible without the inspiration and support of you.

I would like to extend my sincere thanks to Tracy Farbacher for keeping everything in order and scheduling my thesis presentation on May 6th. I must also thank Professor Nathan Beckmann for attending my presentation and asking great questions, and Catherine Copetas for the help with thesis formatting.

Last but not least, I am deeply grateful to my girlfriend and my family for your unparalleled support. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am. I would like to give special thanks to my girlfriend for supporting me through the last semester. I don't know what I did to deserve all of you, but I am glad to have you in my life.

This work was supported (in part) by the National Science Foundation (IIS-1846158, SPX-1822933) and an Alfred P. Sloan Research Fellowship.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 4

2 Background **5**

2.1 Multi-Version Concurrency Control 6

2.1.1 Concurrency Control 6

2.1.2 Isolation Levels 7

2.1.3 MVCC with Snapshot Isolation 8

2.1.4 Indexing in MVCC 8

2.2 System Overview 9

2.2.1 Storage Engine 9

2.2.2 Concurrency Control 12

2.2.3 Catalog 12

2.2.4 Transaction Manager 13

2.2.5 Indexes 14

2.2.6 Garbage Collection 14

3 Non-Blocking Schema Change **15**

3.1 Design 15

3.1.1 Multiple DataTables in SqlTable 17

3.1.2 Version Table in Catalog 18
3.1.3 Physical Addresses in Indexes 19
3.2 Implementation 22
3.2.1 Methods in SqlTable 23
3.2.2 SELECT Migration Policies 25
3.2.3 Single-Version Cache 25
3.3 Unsafe Schema Changes 26
3.3.1 Serializable Isolation Level 26
3.3.2 Invariant and Consistency 26
3.3.3 Execution Steps 32
3.3.4 Pending Constraint List 33
3.4 Alternative Designs 33
3.4.1 Global DataTable Approach 33
3.4.2 Multi-Block Approach 35

4 Evaluation **37**
4.1 Experimental Framework 37
4.2 Workloads 37
4.2.1 SQL Operation Performance 37
4.2.2 Performance Overhead 39
4.2.3 High-Frequency Schema Change 41

5 Related Work **43**
5.1 Existing Systems 43
5.2 Research 45

6 Conclusion and Future Work **47**

Bibliography **49**

List of Figures

1.1 Unavailability of the table caused by a non-blocking schema change . . . 2

2.1 The default storage engine in Terrier 10

2.2 An illustration of the structure of a block. The higher 44 bits of a tuple address is the block header address. 11

3.1 Select operation after a lazy schema change 16

3.2 Update operation after a lazy schema change 17

3.3 Multiple DataTables in a SqlTable. DataTables are the physical student tables, and the SqlTable is the logical student table. 18

3.4 An example of a transaction interacting with the catalog and a SqlTable . 19

3.5 A version mismatch between `addr_version` and `txn_version`. `addr_version` is less than `txn_version` in this case. 20

3.6 An example of how indexes avoid return addresses with `addr_version` greater than `txn_version`. Blue fonts and arrows indicate steps occur in Transaction B and red fonts and arrows indicate steps occur in Transaction A. 21

3.7 An illustration of how indexes retrieve the DataTable pointer from a tuple address 22

3.8 Conflicts between a schema update transaction and a concurrent transaction 27

3.9 The pending constraint list 30

3.10 The Global DataTable design 34

3.11 The Multi-Block design 35

4.1 Performance of SQL operations under different states of the system . . . 38
4.2 Change in throughput for Update transactions overtime after one schema change . . . 40
4.3 Average throughput under high-frequency schema changes workload . . . 41

List of Tables

3.1 A potential inconsistent state. Step I solves the issue by having T_1 refresh timestamp to see commits happened before it installs the constraint. . . . 28

3.2 A potential inconsistent state. Step II solves the issue by having T_2 atomically check constraints in the pending list. T_2 sets flags on the constraints if T_2 violates them at the commit phase. T_2 uses the flags to inform T_1 to abort. T_1 sees the flags at the commit phase and aborts. Operations in red-font sections are performed atomically. 29

3.3 A potential inconsistent state. Step III solves the issue by having T_1 enforces the constraint when it commits to abort concurrent invalid modifications. Operations in red-font sections are performed atomically. 31

3.4 Execution steps of unsafe schema-update transactions and concurrent transactions 32

Chapter 1

Introduction

A database is an organized collection of data, generally stored and accessed electronically from a computer system. The database schema of a database system is its structure described in a formal language supported by the database management system (DBMS). The term "schema" refers to the organization of data as a blueprint of how the database is constructed. A database can be divided into database tables in the case of relational databases. A table has rows and columns, where rows represents records and columns represent the attributes. A tuple is a single row of a table, which contains a single record for that relation. A relation schema, also known as table schema, is the logical definition of a table. It defines what the name of the table is, a set of column names, the data types, and constraints associated with each column.

A schema change is an alteration made to a collection of logical structures in a database. Schema changes are generally made using structured query language (SQL) and DBMSs typically implement schema changes during maintenance windows. Application developers design the schemas in DBMSs to suit applications' use cases. The schema design may be optimized for the intended usage during the development phase. However, software evolves, and applications are updated frequently. Application developers are under pressure to change their database schemas and code to fix bugs and add new features.

1.1 Motivation

Many DBMSs (PostgreSQL [33], SQLite [40], and RocksDB[38]) block the table when they change the table schema. At the same time, they block transactions accessing the table until it finishes installing the schema change. A **transaction** is a logical unit that is

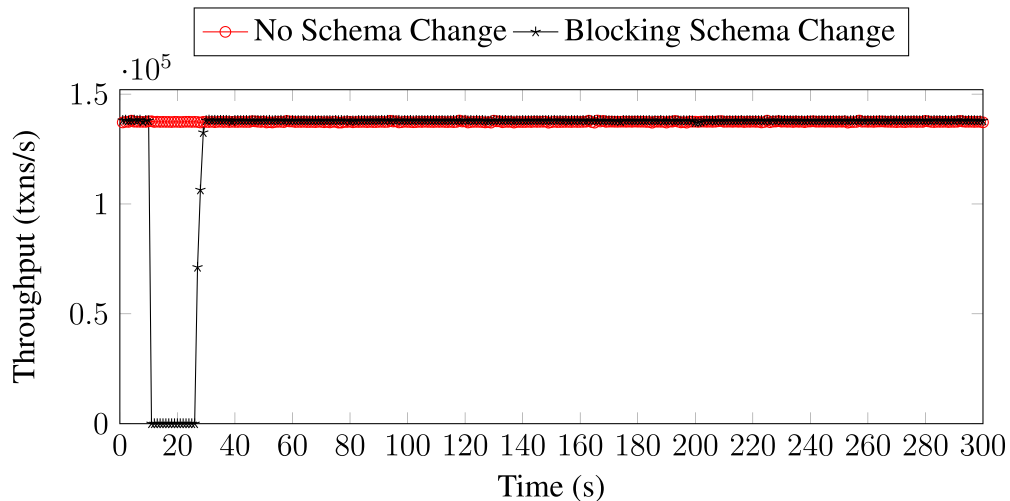

Figure 1.1: Unavailability of the table caused by a non-blocking schema change

independently executed by a DBMS for data retrieval or updates. In relational databases, transactions must be atomic, consistent, isolated and durable – summarized as the ACID acronym [15]. Therefore, transaction throughput is reduced when schema change occurs. During the installation of a schema change, those systems create a new table with the new schema and copy tuples in the old table into the new table. To show how this affects availability, Figure 1.1 shows that a blocking schema change blocks transactions updating the table in one of the systems, Terrier [10]. The table schema consists of two columns of 8-byte integers, and there are 10 million tuples in the table. The red line shows the number of writing transactions the table can process. The black line shows the change in throughput when a blocking schema change occurs at 10 seconds. It takes 17 seconds for the system to copy 10 million tuples to a new location and bring the throughput back to a normal level. As another example, MediaWiki (the wiki platform used by Wikipedia) has seen more than 170 database schema versions [9] in its 55 possible upgrades, and only 5 of out of 55 can be performed online. One notable example was the MediaWiki 1.5 upgrade that caused 22 hours of Wikipedia downtime because the core table schema has changed significantly [12, 24]. In a workload that consists of many read and write transactions, blocking schema changes reduce transaction throughput significantly. Because of this, many applications apply schema changes in software updates during maintenance windows at off-peak times so that they do not affect running transactions.

However, schema changes happen at a high cadence, with a recent study showing that some cloud applications change schemas more than once a week [25]. Unfortunately, the requirement that schema is changed often and applied quickly is at odds with providing a

seamless, uninterrupted service. In particular, Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) providers are bound by Service Level Agreements (SLA) to provide high-availability services, and stopping service to update a database schema might lead to long periods of unavailability which violates the SLA. For example, the G Suite SLA stipulates that G Suite Covered Services web interface must be available 99.9% of the time; otherwise, Google is liable to compensate users for unavailability [11].

We have shown that schemas continue to change after the application development period has ended. Hence, there is a tension between the need to frequently update schemas and high-availability. For high-availability applications, such as web applications and online services, such as SaaS and PaaS providers, stopping and restarting service to update a database schema is unacceptable. It indicates the need for non-blocking schema transformations. For a DBMS to achieve high availability without sacrificing consistency, we believe the system should have the following properties:

- A transaction sees one schema of a table when it starts and can only view and modify data in this schema.
- When the transaction changes a schema, the system verifies that other transactions are not updating that schema.
- Other transactions can still retrieve and update data in the old schema during the installation of a new schema.
- The new schema becomes visible to other transactions only when and if the schema change transaction commits successfully.

Tuples in a DBMS that implements multi-version concurrency control (MVCC) [3] have similar properties as described above. In MVCC, a transaction sees one version of a tuple. It can read the tuple in the old version when another transaction is modifying the tuple. The updated tuple becomes visible to other transactions when the writing transaction commits. MVCC achieves these properties for tuples by maintaining multiple versions of a single logical tuple. In this thesis, we extend the idea of multi-versioning to schemas. We introduce non-blocking schema changes by maintaining multiple versions of logical schemas based on MVCC. The existence of multiple schemas of a table allows transactions to access the table while the table is performing schema update. We introduce three multi-version schema approaches built upon MVCC. Then, we also provide a specific implementation of how to support multi-version schemas in a multi-version DBMS, by maintaining multiple versions of data tables and determines which version a transaction

is allowed to see without latches. In the end, we evaluate our approach under different schema changes and database usage scenarios. We evaluate the overhead of keeping multiple data tables, the cost of version translation, and the change in transaction throughput overtime caused by schema changes. We also compare our results against blocking schema changes. We show that we make schema changes non-blocking, and achieve high throughput and availability with when schema changes are frequent.

1.2 Contribution

We present multi-version relation schemas and non-blocking lazy schema changes for the system. The main contributions of this research are as follows:

1. We discuss three design choices for non-blocking relation schema changes with multi-version schemas.
2. We provide an implementation of non-blocking relation schema changes for a multi-version in-memory DBMS.
3. We compare against blocking schema update to show that our approach improves throughput and availability of the system.
4. We discuss fast recovery from performance degeneration caused by schema updates in a database with hotspot.

Chapter 2

Background

This chapter describes multi-version concurrency control protocol (MVCC) and different components related to multi-version schemas in a relational database system. Section 2.1 explains MVCC and isolation levels. Section 2.2 describes an in-memory MVCC DBMS, Terrier, including the storage engine, the concurrency control system, the catalog, and indexes.

The relational schema of a table in a database management system (DBMS) describes its logical attribute information and constraints. Despite the aim of separation between logical schema and physical data storage from Codd's original relational model [6], in practice, the schema often dictates how a DBMS organizes data on disk or in memory. For example, a disk-oriented DBMS may store tuples in disk pages. The schema determines the size of a tuple, which in turn determines the number of tuples that can be stored in a page. The system may also decide the layout of a page based on the existence of variable-length columns in the schema. Therefore, changing the schema implies changing the physical layout of data.

Schema changes are generally made using `ALTER TABLE` commands. One can write `ALTER TABLE` commands to add or delete columns, create or drop indexes, change the type of existing columns, add or drop constraints on columns, or rename columns or the table itself. Due to time constraints, our research mainly focuses on schema changes that add columns and constraints. We defer the study of other types of schema changes as future work. Many existing systems (PostgreSQL [33], SQLite [40], and RocksDB[38]) implement schema changes by locking tables affected by this change, creating new tables with the new schema, and migrating data over to the new tables. The drawback of this approach is that transactions are blocked during the entire modification. Even though some systems (MariaDB) allow reads to the table during the migration, transactions that

involve writes are blocked. Blocking user transactions is not an viable option in systems with high availability requirements.

Research in this area has proposed several alternative solutions such as query rewriting [8], schema versioning, and temporal querying [20, 29], schema leases [35], schema mapping[41, 45], or schema matching[36]. However, the flexibility offered by these solutions must be weighed against their potential cost and operating constraints. For example, rewritten queries run with a permanent overhead and are on average 4.5 times slower than original queries [8] in some system. Such solutions might have limited applicability for online applications that are performance critical.

Our approach provides non-blocking schema changes by maintaining multiple versions of logical schemas. The idea of multi-versioning has been explored in multi-version concurrency control protocols (MVCC) to maintain multiple versions of tuples in a table. Although many DBMSs (SQL Server, Oracle, PostgreSQL, MySQL with InnoDB) implement MVCC, they either do not have full support for non-blocking schema change or do not utilize the existing multi-version infrastructure already built for tuples to support multi-version schemas. The purpose of this research is to provide a high-performance implementation of non-blocking schema changes by storing multiple versions of data tables for multi-version database systems.

2.1 Multi-Version Concurrency Control

Most of the modern DBMSs implement multi-version concurrency control (MVCC) to achieve higher levels of concurrency. The first mention of MVCC appeared in Reeds 1979 dissertation [37]. We discuss three aspects in the DBMS related to MVCC, namely, concurrency control protocol, isolation levels, and index management.

2.1.1 Concurrency Control

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Without concurrency control, if a transaction is reading from a database at the same time as another transaction is writing to it, it is possible that the reader sees a half-written or inconsistent piece of data. For instance, when making a wire transfer between two bank accounts, if a reader reads the balance at the bank when the money has been withdrawn from the original account, and before it has deposited in the destination account, it would seem that money has disappeared from the bank.

Among these four properties (Atomicity, Consistency, Isolation and Durability) of transactions, Isolation determines how transaction integrity is visible to other users and systems, and it is implemented using a concurrency control protocol. The simplest way is to make all readers wait until the writer is done, which is known as a logical read-write lock. Locks are known to create contention especially between long read transactions and update transactions.

2.1.2 Isolation Levels

Isolation levels define the anomalies that could occur during concurrent transaction execution. A lower isolation level increases the number of different types of anomalies, such as dirty reads, non-repeatable reads, and phantom reads, that users might encounter. Conversely, a higher isolation level reduces the types of concurrency anomalies that users might encounter, but requires more system resources and increases the chances that one transaction blocks another. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level.

The highest isolation level, serializable, only allows execution of the operations of concurrently executing transactions that produce the same effect as some serial execution of those same transactions. A serial execution is one in which each transaction executes to completion before the next transaction begins. The lowest isolation level, read uncommitted, can retrieve data that has been modified but not committed by other transactions. All concurrency side effects can happen in read uncommitted, but there is no read locking or versioning, so overhead is minimized. Snapshot isolation (SI) [2] is an isolation level that guarantees all reads made in a transaction sees a consistent snapshot of the database and the transaction itself successfully commits only if no updates it has made conflict with any concurrent updates made since that snapshot. In a DBMS using SI for concurrency control, reads are never blocked because of concurrent transactions' writes, nor do reads cause delays in a writing transaction. Several major DBMSs, such as Oracle, MySQL, PostgreSQL, MongoDB, and Microsoft SQL Server, have adopted SI. Despite the nice properties of SI Snapshot isolation, it has been known that SI allows non-serializable executions. This occurs when concurrent transactions modify different items that are related by a constraint, and it is called the Write Skew anomaly [2].

It a DBMS enforces that all executions are serializable, the the developers do not need to worry that inconsistencies in the data might appear as artifacts of concurrency. It is well-known how to use strict two-phase locking to control concurrency to produce serializable execution [14]. Serializable Snapshot Isolation (SSI) [4] is a serializable concurrency

control algorithm that makes SI serializable. Under a range of conditions, the overall transaction throughput is close to that allowed by SI, and much better than that of strict two-phase locking [4]. PostgreSQL's new serializable isolation level is based on the SSI technique [30, 32].

2.1.3 MVCC with Snapshot Isolation

Multi-version concurrency control (MVCC), is a concurrency control method commonly used by DBMS to provide concurrent access to the database [3]. MVCC aims at providing concurrent access by keeping multiple copies of each data item. In this way, each user connected to the database sees a snapshot of the database at a particular instant in time. Any changes made by a writer is not visible to other users until the transaction has been committed.

When a MVCC database needs to update a piece of data, it does not overwrite the original data item with new data but instead creates a newer version of the data item. Thus, there are multiple versions stored. The version that each transaction sees depends on the isolation level implemented. The most common isolation level implemented with MVCC is snapshot isolation [2]. A transaction in MVCC with Snapshot Isolation keeps a snapshot view of the database. Read transactions under MVCC typically use a timestamp or transaction ID to determine what state of the database to read, and read these versions of the data. Read and write transactions are thus isolated from each other without any need for locking. Writes create a newer version, while concurrent reads access an older version.

2.1.4 Indexing in MVCC

All MVCC DBMSs keep the databases versioning information separate from its indexes [44]. That is, the existence of a key in an index means that some version exists with that key, but the index entry does not contain information about which versions of the tuple match. We define an *index entry* as a key/value pair, where the *key* is a tuples indexed attribute(s) and the *value* is a pointer to that tuple. The DBMS follows this pointer to a tuples version chain and then scans the chain to locate the version that is visible for a transaction.

Primary key indexes always point to the current version of a tuple. However, how often the DBMS updates a primary key index depends on whether or not its version storage scheme creates new versions when a tuple is updated. For secondary indexes, it is more complicated because an index entrys keys and pointers can both change. The two

management schemes for secondary indexes in an MVCC DBMS differ on the contents of these pointers. The first approach uses *logical pointers* that use indirection to map to the location of the physical version. Contrast this with the *physical pointers* approach where the value is the location of an exact version of the tuple.

The main idea of using logical pointers is that the DBMS uses a fixed identifier that does not change for each tuple in its index entry. This avoids the problem of having to update all of a tables indexes to point to a new physical location whenever a tuple is modified, but since the index does not point to the exact version, the DBMS traverses the version chain from the HEAD to find the visible version.

With this second scheme, the DBMS stores the physical address of versions in the index entries. When updating any tuple in a table, the DBMS inserts the newly created version into all the secondary indexes. In this manner, the DBMS can search for a tuple from a secondary index without comparing the secondary key with all of the indexed versions.

2.2 System Overview

This section describes the physical storage of tuples in a database, MVCC, schema information contained in the catalog, and addresses stored in indexes in a multi-version DBMS. We use Terrier [10] as our example. Terrier is an in-memory DBMS developed at Carnegie Mellon University. Terrier implements lock-free MVCC to support real-time analytics. It uses high-performance, latch-free Bw-Tree for indexing [19, 42].

2.2.1 Storage Engine

The storage engine is separated from execution engine in Terrier so that it is pluggable. The default storage engine is designed to be an in-memory column store, organized in blocks. The system integrates the storage layer with a concurrency control system and a garbage collection mechanism. It implements in-memory MVCC [27], with **delta storage**, **newest-to-oldest** and **in-place** updates [44].

Under MVCC, the DBMS always constructs a new physical version of a tuple when a transaction updates it. The DBMSs version storage scheme specifies how the system stores these versions and what information each version contains [44]. In **delta storage** schema, the DBMS maintains the master versions of tuples in the main table and a sequence of delta versions in separate delta storage. This storage is referred to as the rollback segment

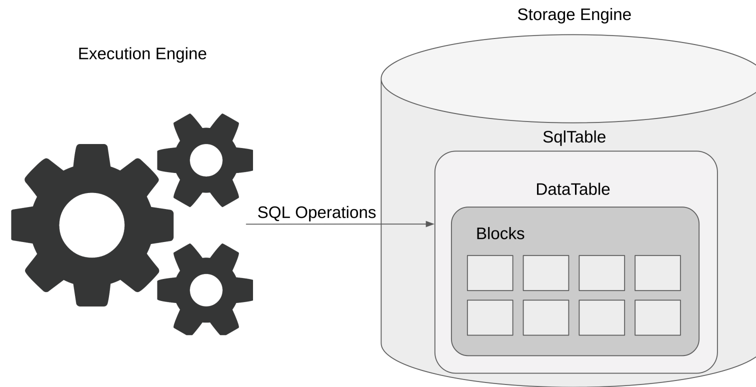

Figure 2.1: The default storage engine in Terrier

in MySQL and Oracle and is also used in HyPer. **Newest-to-oldest** means that the DBMS stores the current version of a tuple in the main table. To update an existing tuple, the DBMS acquires a continuous space from the delta storage for creating a new delta version. This delta version contains the original values of modified attributes rather than the entire tuple. The DBMS then directly performs **in-place** update to the master version in the main table.

We introduce two storage concepts in Terrier, DataTables and SqlTables. Figure 2.1 shows a DataTable and a SqlTable in the storage engine. The DataTable implements MVCC, and it consists of blocks of memory. The SqlTable is a wrapper around DataTable that supports SQL operations.

DataTables

Tuples are stored in 1 MB blocks in Terrier by default. Terrier internally organizes blocks with a data organization model called PAX[1]. More specifically, each block has a fixed layout that corresponds to the relational schema of a table and starts with a block header which contains metadata of its layout. A DataTable in Terrier is a chain of blocks of the same layout, and represents a **physical table** in the system. It returns the tuple values given a tuple address. When a transaction creates a table by giving a relation schema, the system creates a DataTable, and all the blocks in this DataTable are used only for store tuples with that schema. DataTables implement MVCC and support transactions, so transactions can potentially see different values of the same tuple. Given a tuple in a DataTable and a transaction, the DataTable can check if the tuple is visible to the transaction.

The storage layer accesses a tuple using a physical pointer as a key for stored objects

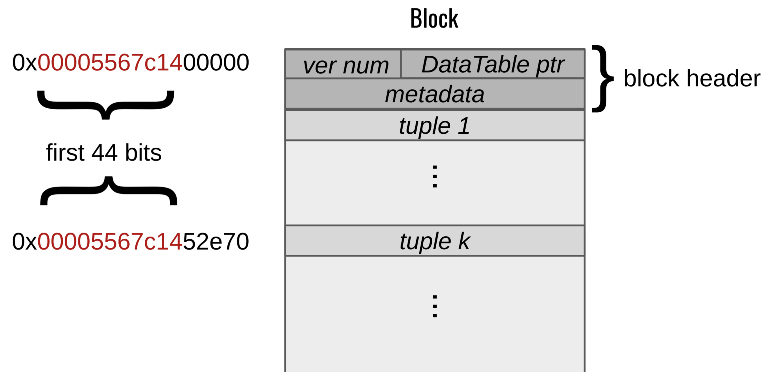

Figure 2.2: An illustration of the structure of a block. The higher 44 bits of a tuple address is the block header address.

in blocks, which always corresponds directly with a physical memory location. Since all blocks are 1 MB in size, it is guaranteed that the heads of two blocks are always 1 MB apart at least, and the system can tell which block any given physical pointer might be in, provided we know the possible starts of blocks. Then, the significant 44 bits inside a 64-bit pointer are sufficient to locate the head of the block as shown in Figure 2.2. The remaining 20 bits are guaranteed to be large enough to hold the possible offset values within a block, assuming any tuple has to be at least 1 byte in size, so there can at most be 2^{20} tuples and unique offsets within a block. Figure 2.2 also shows the information stored in the block header. Terrier stores a version number that is preserved for multi-version schemas in the block header. It also stores a pointer to the DataTable to which the block belongs. The system uses the rest metadata to interpret the layout of the block.

SqlTables

A SqlTable in Terrier is a wrapper around a DataTable that provides SQL-operation interface. It is the entry point for a transaction to access data stored in a table, and it is responsible for communication between the transaction and the DataTable. It serves as a **logical table** existing in the database. SqlTables support the following SQL operations:

- **Select:** Given an address to a tuple and a set of columns, the SqlTable returns the tuple values for these columns.
- **Insert:** Given a vector of values, the SqlTable inserts a tuple with given values to the logical table.

- **Update:** Given an address to a tuple and a vector of new values, the SqlTable updates the existing tuple with new values.
- **Delete:** Given an address to a tuple, the SqlTable removes a tuple from the logical table.
- **Scan:** Given an iterator to the logical table and a size k, the SqlTable return a vector of tuples of size k after the iterator. If there are not enough tuples available, slots in the vector remain empty.

A SqlTable contains only one DataTable in the current implementation. An update on the schema would require locking the whole table and swapping the underlying DataTable with the new schema. We discuss how to change the current design to maintain multiple DataTables in a SqlTable in Chapter 3, so that the system can support multi-version schemas and non-blocking schema changes.

2.2.2 Concurrency Control

DataTables implement in-memory multi-version concurrency control protocols in a way that is similar to HyPer [27]. The concurrency control system is delta based, and updates records in place. The concurrency control system is generally latch-free, except transaction begins and ends, where the system has to update global data structures atomically. For every tuple, the system maintains a singly linked list of undo records, which is the version chain, in a latch-free manner. Every version chain node or an undo record, stores a physical "before image" of the tuple modified, as well as the timestamp of the transaction that modified it. The timestamp is either the transaction's id or the commit timestamp for committed transactions. When the system starts a transaction, the transaction sees a specific version of the tuple based on its timestamp.

On a tuple update, the updating transaction first copies the current version of the tuple into an undo record and then attempts to prepend the record to the version chain atomically. The version chain prepending serves as an implicit "write latch" The transaction aborts if the head of the version chain is not visible to it, so DataTables do not allow write-write conflict.

2.2.3 Catalog

The database catalog of a database instance consists of metadata in which definitions of database objects such as base tables, views (virtual tables), synonyms, value ranges, in-

dexes, users, and user groups are stored. More specifically, it includes schemas of tables, information about columns, constraints on attributes.

The catalog in Terrier is similar to the system catalog in PostgreSQL [31]. The catalog tables in Terrier are SqlTables themselves and hence transnational. Transactions can drop and recreate the tables, add columns, insert and update values into the catalog. While it stores all this information in tables like any other application would, the system fully manage the data in the tables so the data cannot be modified unless an absolute emergency. Some catalog tables are "system-wide" tables, where the data represents the whole systems, no singular database. They store system information such as databases in the system, transaction information, memory and disk usage, timestamps, and statistics. Other catalog tables contain database specific metadata, such as table schemas and index metadata.

Since catalog tables support transactions, transactions see different versions of tuples in the catalog. We discuss how to exploit the transnational property of the catalog to store schema version information in the catalog in Chapter 3.1.2.

2.2.4 Transaction Manager

A transaction is a logical unit that is independently executed by a DBMS for data retrieval or updates. After a transaction begins, it either commits or aborts at the end. Commit phase a coordinating process that takes the necessary steps for either committing or aborting the transaction. A transaction manager is part of the system that is responsible for coordinating transactions across one or more resources. The responsibilities of the transaction manager are as follows:

- Starting and ending transactions using begin, commit, and rollback methods.
- Managing the transaction context. A transaction context contains the information that a transaction manager needs to keep track of a transaction. The transaction manager is responsible for creating transaction contexts and attaching them to the current thread.
- Coordinating the transaction across multiple resources.
- Recovery from failure. Transaction managers are responsible for ensuring that resources are not left in an inconsistent state if there is a system failure.

2.2.5 Indexes

Terrier uses latch-free Bw-Tree [19, 42] for indexing and values are physical addresses to tuples in SqlTables. If columns are indexed, a transaction can ask indexes for the tuple addresses and then pass them to SqlTable to access tuples efficiently. Terrier keeps the databases versioning information separate from its indexes like many other MVCC DBMSs [44]. The existence of a key in an index means that some version exists with that key, but the index entry does not contain information about which versions of the tuple match. We discuss how indexes interact with tables that have multiple schemas in Chapter 3.1.3.

2.2.6 Garbage Collection

While multi-version concurrency control (MVCC) supports fast and robust performance in in-memory, relational databases, it has the potential problem of a growing number of versions over time due to old versions. Although a few TB of main memory is available for enterprise machines, the memory resource should be used carefully for economic and practical reasons. Thus, to maintain the necessary number of versions in MVCC, the system needs to delete versions which will no longer be used. This process is called garbage collection. MVCC uses the concept of visibility to define garbage. A set of versions for each record is identified as candidates if their version timestamps are lower than the minimum value of snapshot timestamps of active snapshots in the system. Garbage collection can safely reclaim all such candidates.

In Terrier, all of the data to clean up resides within the transaction context themselves. The transaction manager essentially keeps a back queue of all the transactions that are finished. In the absence of long-running transactions [18], the garbage collector is then a background thread that periodically polls from the transaction manager's queue and processes the finished transactions.

Chapter 3

Non-Blocking Schema Change

This chapter discusses the behavior of non-blocking schema changes. It focuses on one specific design of non-blocking schema changes in detail and includes a complete implementation of SqlTable operations such as Select, Insert, Delete, Update, and Scan. At the end of this chapter, we also discuss different storage designs and analyze their advantages and disadvantages.

3.1 Design

In this section, we discuss how to achieve non-blocking schema changes in terms of DataTables and SqlTables in multi-version DBMSs. We call our design the **lazy schema change** method.

The lazy schema change method does not copy all tuples from the old schema to a new schema. When a schema change occurs, it only creates an empty chunk of blocks for storing tuples with the new schema and returns without populating them. Hence, the whole schema change operation is cheap. Tuple migration is carried out by subsequent operations as lazily as possible.

We describe how the system executes Select operations after a lazy schema change. Suppose the system has a physical table `student` with two columns `id` and `name`. There are two tuples initially in the table, `(1, abc)` and `(2, xyz)`. At time T_1 in Figure 3.1, a schema-change transaction adds a third column `age` with default value `0`. The system then creates an empty physical table with all three columns. At time T_2, a transaction tries to read `id` and `name` of the tuple with `id=1`. The system detects that `id` and `name` are

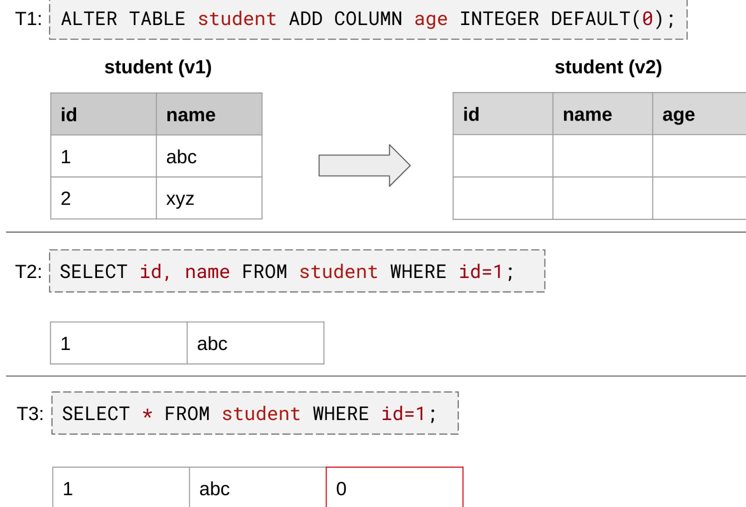

Figure 3.1: Select operation after a lazy schema change

attributes in the previous schema, so it retrieves the values from the old table and returns. At time T_3, a transaction tries to read all columns in `student` table. The system knows the third column has a default value `0`, so it first retrieves the values of the first two column from the old table, appends `0` on the fly, and returns. The second physical table remains empty and untouched.

Now we describe how the system executes Update operations after a lazy schema change. Again, suppose the system has a physical table `student` with two columns `id` and `name`. There are two tuples initially in the table, `(1, abc)` and `(2, xyz)`. At time T_1 in Figure 3.2, a schema change transaction adds a third column `age` with default value `0`. The system creates an empty physical table with all three columns. At time T_2 in Figure 3.2, a transaction updates the `age` of the tuple with `id=2` to be `21`. The system detects that `age` is a column that only exists in the new schema, so it adds the tuple `(2, xyz, 21)` to the new physical table. At the same time, it removes the old tuple from the old physical table. At time T_3 in Figure 3.2, a transaction tries to change the `name` of the tuple with `id=1` to be `bcd`. The system checks and realizes that it can perform the update in the old physical table because the column exist in both schemas. Therefore, the system updates the data in the old physical table.

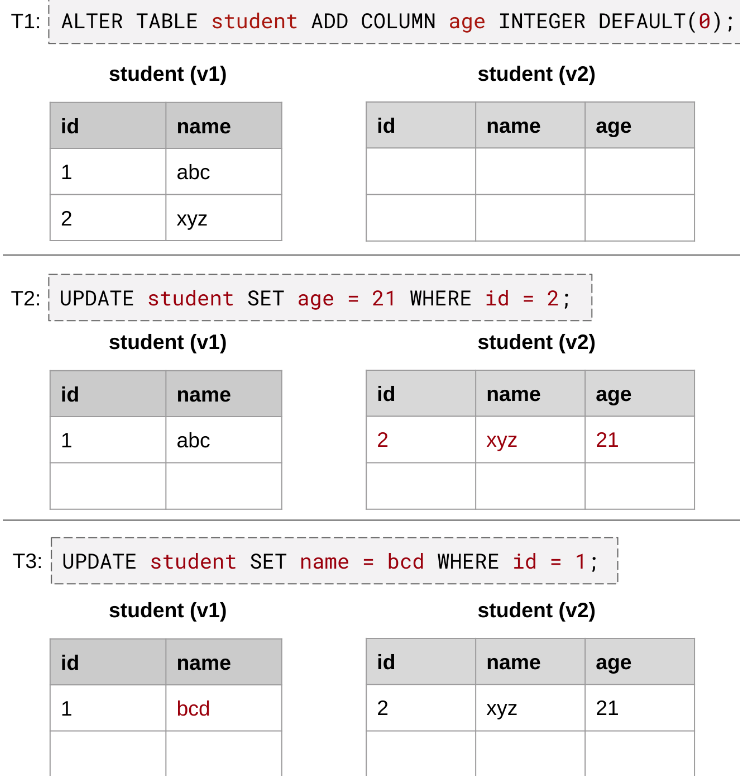

Figure 3.2: Update operation after a lazy schema change

In general, lazy schema change requires the ability to store tuples in the same logical table in different physical locations. Note that the application drives the data migration because the system only migrates tuples touched by transactions.

3.1.1 Multiple DataTables in SqlTable

Most systems (PostgreSQL [33], SQLite [40], and RocksDB[38]) implement blocking schema changes by creating a new table with the new schema, and moving tuples from the old table to the new table. During the migration, these two physical tables with different schemas coexist, but neither can be accessed or is ready to be modified by transactions. The lazy schema change is a relaxation of this migration process. It allows transactions to access multiple physical tables of different schemas at the same time.

Figure 3.3 shows the design in the storage engine. We make DataTables correspond to different versions of physical tables. The SqlTable acts as the unique logical table in

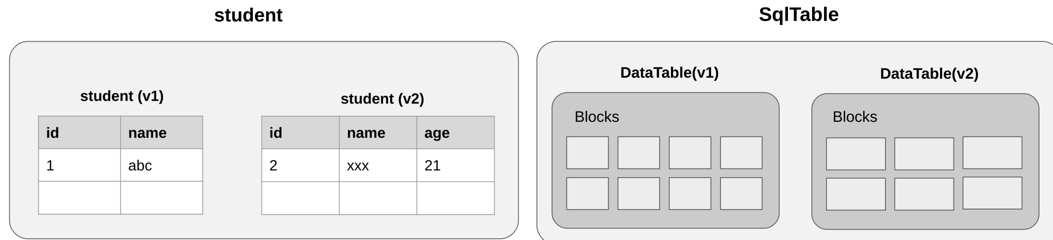

Figure 3.3: Multiple DataTables in a SqlTable. DataTables are the physical student tables, and the SqlTable is the logical student table.

the database. Suppose we have a `student` table with two columns, as shown in Figure 3.3, and change the schema by adding a third column `age`. Then, the system has two versions of physical tables (DataTables) with two complete schemas, but only one logical `student` table (SqlTable).

3.1.2 Version Table in Catalog

In MVCC, different transactions see different versions of a tuple based on their timestamps. Correspondingly, in multi-version schemas, the next step is to develop a mechanism for SqlTables to expose the appropriate version(s) of DataTables to transactions.

The system maintains a new version table in the catalog that stores tables and their version numbers. One of the reasons that the version table is in the catalog is that the catalog is meant to store metadata of the database, and the number of schema versions of a table is metadata that is not exposed to the client. Another reason is that catalog tables are transactional SqlTables that implement MVCC so the system can exploit the existing implementation to make transactions see different version numbers for each table. Our design requires a transaction to obtain its version number for each table it accesses from the version table, and pass it along with every SQL operation to the corresponding SqlTable. However, the version number of a SqlTable does not change for non-schema-change transactions under snapshot isolation. Therefore, transactions can cache their version numbers for future queries.

Figure 3.4 shows the process of obtaining a version number for a table before accessing the table. In Figure 3.4, a transaction tries to execute a Select query on the `student` SqlTable. First, it talks to the version table in the catalog to get its version number for the `student` table, which is `v2` in this example. Then it asks the SqlTable to execute a Select operation and provides its version number. The SqlTable then retrieves the tuple in

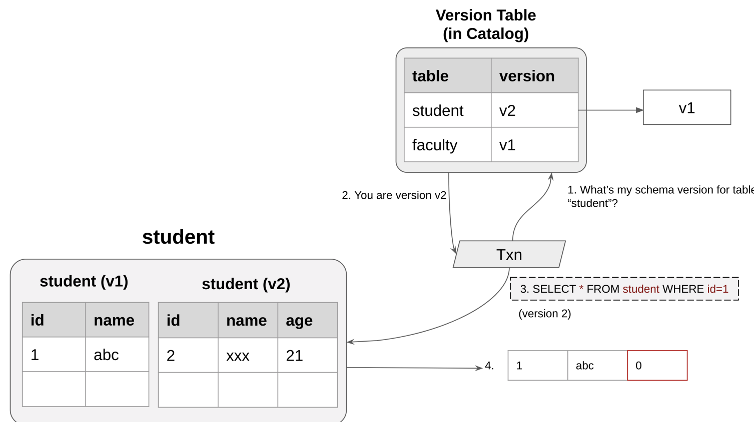

Figure 3.4: An example of a transaction interacting with the catalog and a SqlTable

the schema `v1`, transforms it into the schema `v2` that is expected by the transaction, and returns `(1, abc, 0)`. We call the process of transforming a tuple from one schema version into another schema version **version translation**.

3.1.3 Physical Addresses in Indexes

We next explain how indexes work with multi-version schemas. For online transaction processing (OLTP) [28] systems, most of the time transactions access tuples efficiently via indexing. Some systems, such as Terrier, store physical memory addresses in indexes. To invoke SQL operations like Select, Update in SqlTables, transactions need to provide addresses of the tuples. They obtain these addresses from indexes.

Let R be a table and T be a transaction. We introduce two definitions of versions:

`txn_version`: We define `txn_version` to be the version T receives from the version table in the catalog for R.

`addr_version`: For all addresses in a DataTable, we define `addr_version` of these addresses to be the schema version of the DataTable.

By definition, addresses from the same DataTable have the same `addr_version`. The two types of versions do not need to be the same. A **version mismatch** is the case where a transaction provides an address whose `addr_version` does not match the `txn_version`. We discuss when version mismatch could occur and how SqlTables detect it in the following.

Version Mismatch: `addr_version` less than `txn_version`

We first discuss the case where `addr_version` is less than `txn_version`. It occurs when a tuple is an old version of DataTable.

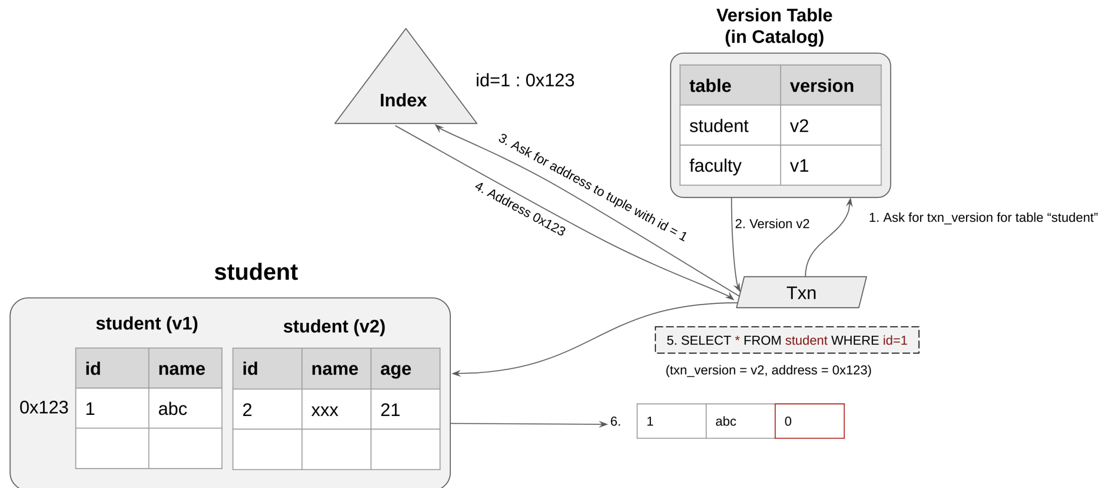

Figure 3.5: A version mismatch between `addr_version` and `txn_version`. `addr_version` is less than `txn_version` in this case.

Figure 3.5 shows an example of this type of version mismatch. In Figure 3.5, we have a `student` SqlTable that have two DataTables and it has an index built on the column `id`. A Select transaction first asks for its `txn_version` for this table, which is `v2`. Then it asks the index for the address of the tuple with `id=1`. The index returns the address `0x123`, which is an address in the old DataTable. Then transaction then calls Select method along with its version number and the address. This is a version mismatch because `txn_version` is `v2` and `addr_version` is `v1`.

Version Mismatch: `addr_version` greater than `txn_version`

We next discuss the case where `addr_version` is greater than `txn_version`. This case cannot happen because our design requires the indexes not to return an address with `addr_version` greater than `txn_version` by checking the visibility of the tuple.

Figure 3.6 shows how the index avoids the case where `addr_version` is greater than `txn_version`. Blue fonts and arrows indicate the steps that happen in transaction B and red fonts and arrows indicates the steps that happen in transaction A. The index initially

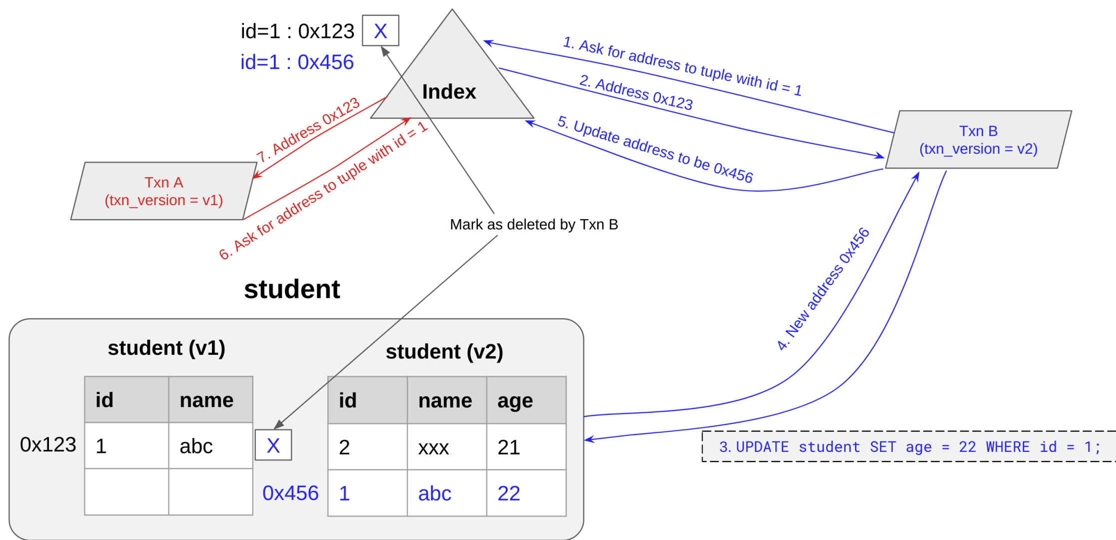

Figure 3.6: An example of how indexes avoid return addresses with `addr_version` greater than `txn_version`. Blue fonts and arrows indicate steps occur in Transaction B and red fonts and arrows indicate steps occur in Transaction A.

stores the address, `0x123` to the tuple with `id=1`. In this example, transaction B is a transaction that starts after the schema change, so it sees the second schema. Transaction A is a transaction that starts before the schema change, so it still sees the first schema. Then, transaction B tries to update the tuple with `id=1`. It first asks the index for the address of the tuple. The index returns `0x123`. Next, transaction B updates the tuple and it causes the SqlTable to migrate the tuple from `v1` to `v2` by logically deleting the tuple in `v1` and inserting a tuple to `v2`. The SqlTable returns the new address `0x456` to transaction B and transaction B updates the index with the new address. The index marks the old key-value pair to be logically deleted by transaction B and inserts a new key-value pair. Next, transaction A asks the address to the tuple with `id=1`. The index detects that the old address is still visible to transaction A and the new address is not visible to transaction A, so it returns the old address `0x123` with `addr_version v1` to transaction A. Then, the transaction A can still access the old tuple even though it is marked deleted by transaction B. Therefore, an index never returns an address with `addr_version` greater than `txn_version`.

Indexes perform visibility checks that happen in step 2 and 7 in Figure 3.6 by asking DataTables. As described in Section 2.2.1, one can obtain the starting address of the block, given a physical address to a tuple in a block, by extracting out the first 44 bits in

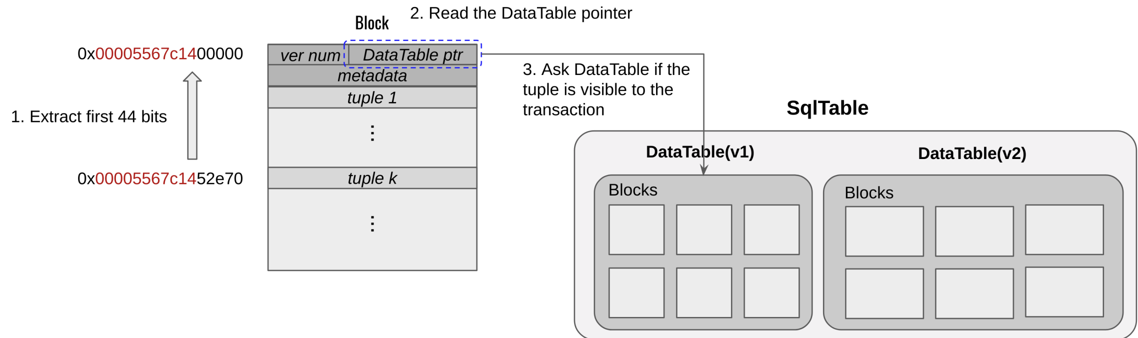

Figure 3.7: An illustration of how indexes retrieve the DataTable pointer from a tuple address

the address because each block is of size 1 MB by default in Terrier. Because the block header contains a pointer to the DataTable, the index jumps to the block header, as shown in Figure 3.7, retrieves the pointer to the DataTable, and asks the DataTable if the address is visible to a transaction.

Version Mismatch Detection

The SqlTable detects a version mismatch by exploiting the fact that the block header contains the version number of DataTable, as described in Section 2.2.1. By definition, the version number is the `addr_version` for all addresses in this block. Given a physical address to a tuple in a block, the SqlTable can jump to the block header, and retrieves the version number in a similar way as indexes retrieve the DataTable pointers in the block header. Next, the SqlTable compares `addr_version` with `txn_version` and detects a version mismatch. Finally, depending on the operation type, the SqlTable may perform a version translation to transform the tuple to the new version before it returns the tuple for Select operations, or migrate the tuple to the latest DataTable for Update operations.

3.2 Implementation

In this section, we describe a specific implementation of various operations of a SqlTable with multi-version schemas. A SqlTable internally maintains a map from schema version numbers to DataTables. A transaction can access a tuple by a physical address obtained from an index. As indicated in the previous section, the address has a `addr_version` and the transaction has a `txn_version`, and indexes guarantee that the `addr_version`

is not greater than `txn_version`. Based on if the two versions match or not, operations go to different code paths. If `txn_version` is equal to `addr_version`, then it indicates that the tuple is stored in a form that is consistent with the transaction's view of the schema. On the other hand, if `txn_version` is greater than `addr_version`, tuples are located in a DataTable with an old schema. The SqlTable performs version translation and moves the tuple to the newer DataTable if necessary.

3.2.1 Methods in SqlTable

SqlTables provide access methods like Select, Insert, Update, Delete and Scan for transactions to store, modify, retrieve, delete and update data in a table. We discuss how SqlTables with multiple DataTables execute these methods.

Select

If `txn_version` is equal to `addr_version`, the SqlTable knows that the address belongs the DataTable that has the expected schema, so it directly retrieves the tuple from that DataTable and returns it to the transaction. If `txn_version` is greater than `addr_version`, then it means that the wanted tuple resides in a DataTable with an old schema. After retrieving the tuple from the old DataTable, the SqlTable performs version translation to transform the tuple into the expected schema before returning to the transaction by filling out values for columns that exist in the new schema, removing values for columns that do not exist in the new schema, and appending default values if necessary.

Update

A transaction updates a tuple by providing the physical address to the tuple and the new values for a subset of the columns. If `txn_version` is equal to `addr_version`, then the SqlTable directly follows the address and updates the tuple in the corresponding DataTable. If `txn_version` is greater than `addr_version`, then it determines whether the set of attributes the transactions modifies is a subset of both the old schema and the new schema. If the transaction only modifies the attributes that are in the intersection of two schemas, then the SqlTable updates the tuple in place in the old DataTable. Otherwise, it deletes the tuple in the old DataTable, copies the tuple over to the new DataTable, and updates the tuple directly in the new DataTable. After the migration, the tuple stays in the DataTable with the correct schema version for the rest of the transaction.

It is important to delete the tuple from the old DataTable before inserting a new version of the tuple into the new DataTable. It is because if two in-flight transactions are updating the same tuple and both insert a new tuple to the new DataTable before deletion, the new DataTable would have two copies of the tuple. On the other hand, if they delete before inserting the tuple, one of them would abort due to deletion failure since the system avoids write-write conflict. Having deletion before insertion avoids inserting duplicates of the tuple to the new DataTable and allows conflicting transactions to abort early.

If the SqlTable migrates a tuple into a new DataTable in an Update operation, it changes the location where the tuple is physically stored. Thus, the SqlTable also returns a new physical pointer, and the transaction is responsible for updating the index with the new address.

Insert

Insertions do not require physical addresses. Given a new tuple, the SqlTable always inserts new tuples into the newest DataTable with the correct schema version (i.e., `txn_version`). Storing the tuple in some old DataTable has no benefits.

Delete

The SqlTable deletes a tuple in an address provided by the transaction. The SqlTable identifies the DataTable with schema version equal to `addr_version` and deletes the tuple from the DataTable. In this case, the SqlTable does not check if `txn_version` is equal to `addr_version`. Whether the tuple was in an old schema or the new schema does not change the behavior.

Scan

Scan is different from Select because it returns a vector of tuples instead of a single tuple. Scan takes an iterator to the logical table, which is internally a physical address in some DataTable. It populates the buffer vector as it iterates through the tuples that are visible to the calling transaction. After scanning all the tuples in a DataTable, it goes to next DataTable and continues until it scans all DataTables.

3.2.2 SELECT Migration Policies

When a SqlTable has multiple versions of schemas (DataTables), if a Select operation with a version mismatch only accesses a subset of the attributes that is in the intersection of the new schema and the old schema, then the SqlTable can retrieve the data from the old DataTable. On the other hand, when it accesses columns only contained in the new schema, there are two migration policies, depending on if the SqlTable move tuple into the newer version DataTable after version translation:

> **Migration on Reads:** The SqlTable moves a tuple to a newer DataTable after it performs version translation on this tuple.

> **Migration on Writes:** The SqlTable only moves a tuple to a newer DataTable if it performs version translation and it is an Update operation.

If the SqlTable moves a tuple to a new location, the transaction must update the index with a new address, which incurs extra performance cost for the transaction. Although *Migration on Reads* policy requires the transaction to update the index, subsequent Select operations on the tuple do not need to perform version translations. In contrast, with the *Migration on Writes* policy, the transaction does not need to update the index for Select operations but the SqlTable needs to translate between versions for all future Select queries. These two policies perform differently on varying workloads, and our design implements the first policy. We defer the investigation on the two policies as future work.

3.2.3 Single-Version Cache

A SqlTable internally maintains a map from version numbers to DataTables. It adds an indirection layer to find the correct DataTable. Depending on the implementation of the data structure, this additional layer incurs extra CPU overhead if the map involves hashing. Since the SqlTable performs a lookup in the map for every SQL operation in the transaction, the aggregated cost is nontrivial even when the SqlTable has only one schema version. To avoid performance penalty for tables that have only one schema, the system caches the DataTable in the SqlTable when there is only one DataTable so that it bypasses the map lookup.

3.3 Unsafe Schema Changes

The schema changes we have discussed so far (i.e., `ADD COLUMN`) can be performed safely without interaction with other concurrent transaction. The other type of schema changes we address in this thesis is `ADD CONSTRAINT`. Unlike adding a column, adding a constraint to a column can be unsafe to perform concurrently. For example, if a transaction adds a constraint to a column while there are in-flight transactions inserting values into the same column, inserting transactions do not know the constraint and may violate it. Then the system is likely to end up in an inconsistent state.

3.3.1 Serializable Isolation Level

If transactions run in serializable isolation level, such as Serializable Snapshot Isolation [4], the database does not result in an inconsistent state described previously. Serializable isolation requires serializability among any transactions. However, to avoid potential inconsistent states of the database created by schema-change transactions, the system only needs serializability between the schema-change transaction and any other concurrent transaction. Non-schema-change transactions can still execute under a lower isolation level previously defined by the system, such as Snapshot Isolation, with respect to each other.

Because the serializable isolation level limits concurrency, our approach implements its own consistency checks to prevent from data inconsistency caused by schema-change transactions. These checks only forces serializability between the schema-change transaction and any other concurrent transaction. Furthermore, these checks only exist when there are running schema-change transactions. Therefore, they preserve high concurrency allowed by Snapshot Isolation with low overhead.

3.3.2 Invariant and Consistency

We discuss how to avoid data inconsistency caused by schema-change transactions. It is necessary for the transaction that installs the constraint to make sure that no existing tuples violate the constraint to preserve data consistency. Suppose T_1 is a transaction that adds a `NOT NULL` constraint to column `age` in table `student`. It has to scan all the existing tuples in `age` and make sure that they already satisfy the constraint to be installed. If not, T_1 has to abort to avoid inconsistency. Nevertheless, even if all existing tuples satisfy the constraint, there can be another concurrent transaction T_2 inserting a tuple to that table

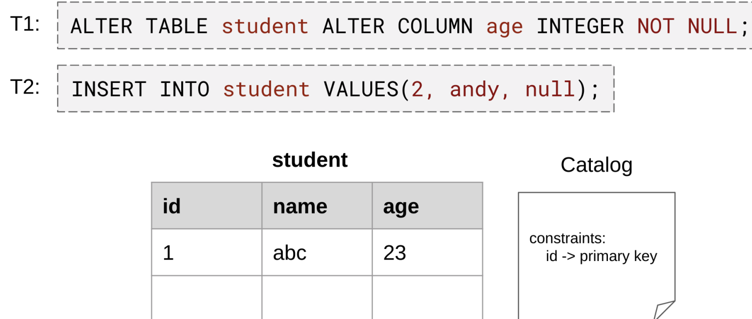

Figure 3.8: Conflicts between a schema update transaction and a concurrent transaction

with the value of that column being `NULL`. As long as T_2 begins before T_1 commits, T_2 does not see the constraint so T_2 can insert `NULL` values to the column under snapshot isolation.

We introduce three steps to resolve inconsistency issue. Each of these three steps resolves a specific inconsistent case and, if combined, they guarantee consistency in the database. In the following discussion, we assume T_1 is a transaction adding a `NOT NULL` constraint on column `age`, and T_2 is a transaction inserting `(2, andy, null)` into the table `student`, as shown in Figure 3.8.

Step I: T_1 Refreshes Its Timestamp

The left side of Table 3.1 shows the inconsistent state where `NULL` value is inserted and constraint is installed. In this example, T_1 begins first. Before T_1 adds `NOT NULL` constraint to the catalog, another transaction T_2 inserts a `NULL` entry into the column and commits. Then T_1 modifies the catalog and scans the table to check if existing tuples violate the constraint. However, T_1 cannot see T_2's modifications since it has newer timestamp under snapshot isolation. It indicates that T_1 needs to catch all the transactions that have begun and committed before the modification to the catalog was completed since they might have violated the constraint.

To solve this problem, T_1 refreshes its timestamp so that it can see those modifications happened before it installed the constraint in the catalog. The right side of Table 3.1 describes the procedure that avoids the problem. After T_1 begins, T_2 inserts a `null` entry and commits. Then T_1 add a `NOT NULL` constraint to the catalog. Next, T_1 refreshes its timestamp so that now it can see the modification made by T_2. When T_1 scans the table, it sees the `null` entry inserted by T_2 and aborts because it finds an existing record that

Before		After	
T_1	T_2	T_1	T_2
BEGIN		BEGIN	
	BEGIN		BEGIN
	Insert `null`		Insert `null`
	COMMIT		COMMIT
Add `NOT NULL`		Add `NOT NULL`	
Scan & check		**Refresh Timestamp**	
COMMIT		Scan & check	
		ABORT	

Table 3.1: A potential inconsistent state. Step I solves the issue by having T_1 refresh timestamp to see commits happened before it installs the constraint.

violates the `NOT NULL` constraint.

Step II: T_2 Check Pending Constraints During Commit

Having T_1 capture committed modifications before constraint installation is not enough to guarantee consistency. Even though the constraint has been installed, T_1 has not committed. A concurrent transaction T_2 can violate the constraint and safely commit under snapshot isolation as long as T_1 has not committed. The left side of Table 3.2 shows how this can also end up in an inconsistent state. First, T_1 begins and adds the constraint to the catalog. Then, T_2 begins. Next, T_1 refreshes its timestamp, scan the table, and do not find any existing tuple violating the constraint. Before T_1 enters its commit phase, T_2 inserts a `null` entry and commits. Finally, T_1 commits and the database is in a inconsistent state.

It is clear that communication between two transactions is unavoidable to achieve consistency. An obvious solution is to make the "Scan & check" step and Commit phase atomic in T_1. However, the "Scan & check" step requires an entire scan of columns, which drastically increases the duration of the critical section. We should make the critical section as short as possible because acquiring a latch to make these two steps atomic blocks other transactions from updating the table for a long time.

Instead, we propose adding a separate data structure, the pending constraint list, to the system as shown in Figure 3.9. The transaction manager described in Section 2.2.4 maintains the pending list that is visible to all transactions. Each entry in the pending list is a pair of a constraint and a boolean flag. We call the flag the **violation flag**. With the pending list, T_1 not only adds constraint to the catalog, but also adds the constraint to a

Before		**After**	
T_1	T_2	T_1	T_2
BEGIN		BEGIN	
Add `NOT NULL`		Add `NOT NULL`	
	BEGIN		BEGIN
		Add to pending list	
Refresh Timestamp		Refresh Timestamp	
Scan & check		Scan & check	
	Insert `null`		Insert `null`
	COMMIT		Check constraint
			Set violation flag
			COMMIT
COMMIT		Check Flag	
		ABORT	

Table 3.2: A potential inconsistent state. Step II solves the issue by having T_2 atomically check constraints in the pending list. T_2 sets flags on the constraints if T_2 violates them at the commit phase. T_2 uses the flags to inform T_1 to abort. T_1 sees the flags at the commit phase and aborts. Operations in red-font sections are performed atomically.

pending list. When T_2 enters its commit phase, it atomically

1. checks the pending constraints list
2. sets the violation flag for each constraint it violates
3. commits

On the other hand, when T_1 enters commit phase, it atomically

1. checks the violation flag for its constraint
2. if the violation flag is set, aborts
3. else commits

Now when T_2 commits, as shown in the right side of Table 3.2, it sets the violation flag that is checked by T_1 when T_1 enters its commit phase. T_1 sees the violation flag and aborts.

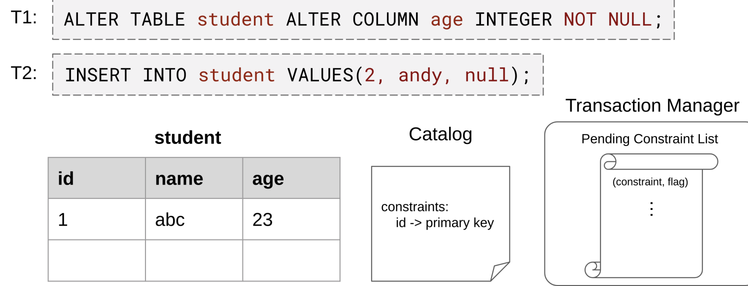

Figure 3.9: The pending constraint list

This design uses first-commiter-wins policy. Since T_2 commits before T_1 finishes, our approach requires T_1 to abort. In this example, T_2 sends a signal to T_1 by setting the violation flag on the constraint in the pending list. The flag indicates that the constraint T_1 is installing will be violated, and hence T_1 should abort.

Step III: T_1 Enforces Constraints During Commit

Step III resolves the inconsistency as a result of T_1 committing before T_2. The left side of Table 3.3 shows the case where T_1 first installs the constraint, both in the catalog and the pending list, and commits before T_2 commits. When T_2 enters its commit phase, it looks at the constraint in the pending list and sets the violation flag as described in step II. In this case, T_2 commits and hopes the schema-change transaction would see the flag and abort, whereas T_1 has already committed.

This scenario occurs because the committed constraint is not enforced on the in-flight transaction. To address the issue that concurrent transactions always try to abort T_1, we add another `enforcing flag` in the pending list. Now each entry in the pending list is a tuple `(constraint, violation flag, enforcing flag)`. The philosophy is that, when T_1 commits, the constraint is no longer pending and should be enforced for all new transactions and concurrent transactions. During its commit phase, T_1 atomically

1. checks the violation flag for its constraint
2. if the violation flag is set, aborts
3. else sets the enforcing flag on the constraint and commits

By enforcing the constraint, T_1 forces concurrent transactions to check against the constraints and abort if they violate the installed constraint. During T_2's commit phase, it

Before		After	
T_1	T_2	T_1	T_2
BEGIN		BEGIN	
Add `NOT NULL`		Add `NOT NULL`	
	BEGIN		BEGIN
Add to pending list		Add to pending list	
Refresh Timestamp		Refresh Timestamp	
Scan & check		Scan & check	
	Insert `null`		Insert `null`
Check violation flag		Check violation flag	
COMMIT		Set enforcing flag	
		COMMIT	
			Check enforcing flag
			ABORT
	Check Constraint		
	Set Violation Flag		
	COMMIT		

Table 3.3: A potential inconsistent state. Step III solves the issue by having T_1 enforces the constraint when it commits to abort concurrent invalid modifications. Operations in red-font sections are performed atomically.

atomically

1. checks the pending constraints list
2. aborts if it violates an enforced constraint
3. else sets the violation flag for each pending constraint it violates
4. commits

The right side of Table 3.3 shows how this approach solves the problem. T_1 first begins and adds a `NOT NULL` constraint to the catalog. Then, T_2 begins. T_1 continues to add its constraint to the pending list, refreshes its timestamp, and does not find any existing tuple violating the constraint. Before T_1 enters commit phase, T_2 inserts a `null` entry. Then, T_1 enters its commit phase and does not see a violation flag for its constraint, so it sets the enforcing flag and commits. When T_2 enters its commit phase, it sees the enforcing flag and aborts.

3.3.3 Execution Steps

When a schema-change transaction begins, it increments the schema version of the table in the version table in the catalog. Then it modifies the constraints in the catalog. Since write-write conflicts are not allowed, other transactions that update the schema of the same table would fail to increment the number in the version table and abort. Therefore, there exists at most one schema-change transaction for each table at any point in the system. We summarize the execution steps of a schema-change transaction and other concurrent transactions in Table 3.4.

Txn Installing Constraints (T_1)	Concurrent Txn (T_2)
1. BEGIN	1. BEGIN
2. Increment version in version table	2. Violate constraints
3. Modify constraints in the catalog	3. Atomically:
4. Add constraints to pending list	a. Check enforcing flag; ABORT if set
5. Refresh timestamp	b. Set violation flags
6. Scan and check constraint violation	c. COMMIT
7. Atomically:	
a. Check violation flags; ABORT if set	
b. Set enforcing flags	
c. COMMIT	

Table 3.4: Execution steps of unsafe schema-update transactions and concurrent transactions

We prove that the three steps established above are sufficient to avoid data inconsistency caused by schema-change transactions. Firstly, such inconsistency can only happen when there are concurrent transaction violating the constraints being installed. Next, a concurrent transaction can enter its commit phase in one of the following cases:

- before step 4 in T_1
- after step 4 but before step 7 in T_1
- after step 7 in T_1

If T_2 commits before step 4 in T_1, T_1 would see T_2's modification after T_1 refreshes its timestamp and abort. If T_2 commits after step 4 but before step 7, then T_2 would set the violation flag that causes T_1 to abort. Finally, if T_2 commits after step 7, then T_2 would see the enforcing flag and abort.

3.3.4 Pending Constraint List

When an unsafe schema-update transaction aborts, modifications to the version table and constraints in the catalog are rolled back automatically because catalog tables are SqlTables themselves, which implement rollback semantics. In contrast, changes made to the pending list should be rolled back manually since it is a separate data structure.

When a schema-change transaction aborts, it removes its constraints in the pending list so that new transactions do not need to check against them. When a schema-change transaction commits, it receives a commit timestamp in MVCC, and writes the timestamp on the constraints as the enforcing flag in the pending list. It does not remove constraints from the pending list since current transactions need to check if they violate an enforcing constraint. A concurrent transaction only checks constraints in the pending list with no enforcing flag or with commit timestamps after its start timestamp. Therefore, new transactions that begin after the schema-change transaction that has committed do no need to check its constraints. When there are no running transactions with start timestamps before the commit timestamp, the system returns to the normal state and is ready for next schema update. These enforcing constraints become obsolete and the garbage collection eventually cleans them up.

3.4 Alternative Designs

We discuss two alternative designs of the lazy schema change. We analyze their advantages and disadvantages compared to the current approach. The two designs differ in the storage layer. Both of them require a pending list to resolve data inconsistency caused by unsafe schema changes.

3.4.1 Global DataTable Approach

Figure 3.10 shows the storage design of the Global DataTable approach. Since DataTables in Terrier provide a snapshot of tables for each transaction, this approach maintains a global internal DataTable which has two columns: `table` and `dt_address`, which is pointers to DataTables. Given a transaction, the SqlTable can look up this global table and retrieve the pointer to the DataTable with the correct schema version. To retrieve tuples stored the previous DataTables, it traverses the version chain to find the pointers to old DataTables since DataTables in Terrier is implemented with newest-to-oldest delta chain as described in Section 2.2.1. This global table does not need to be durable. If the system

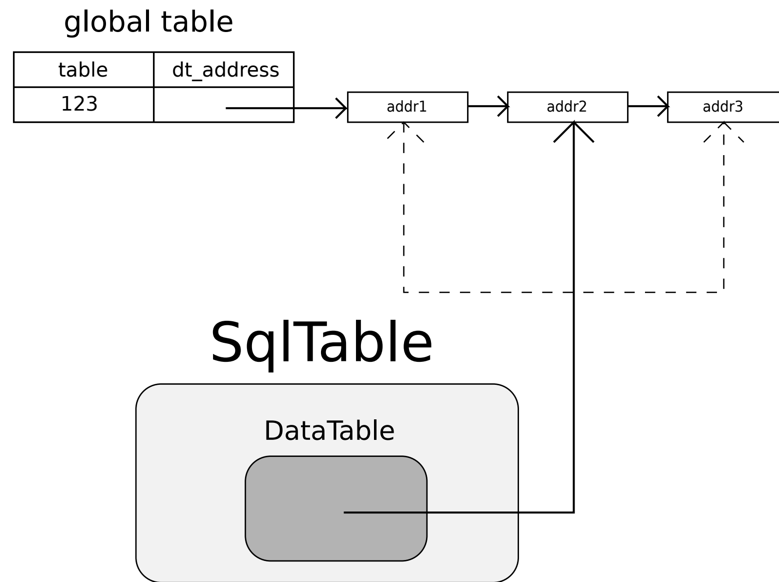

Figure 3.10: The Global DataTable design

reboots, all the address information is invalid, and the system needs to populate it with the new addresses.

Pros

This approach uses DataTables as the abstraction of physical tables and SqlTables as the abstraction of logical tables. Instead of getting the version numbers from the version table in the catalog, transactions directly receive pointers to DataTables.

Cons

Because a single transaction can potentially need more than one DataTable pointers if some tuples are stored in old schemas, the design requires that transactions can see multiple versions and traverse the version chain. MVCC with snapshot isolation does not support this functionality. Even if the SqlTable can walk through the chain, it can incur performance overhead when the version chain is long.

Another drawback of this approach is that a delta record is collected and freed by garbage collection when no transaction needs to access it. Once the record is recycled, the system loses the address to the old DataTable that may still contain valid tuples. It requires

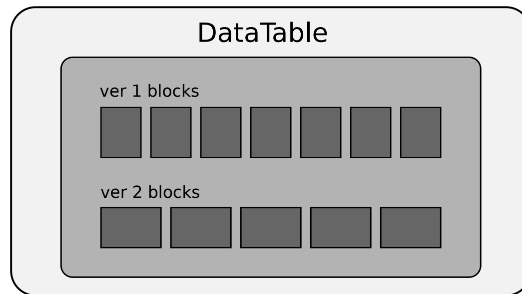

Figure 3.11: The Multi-Block design

the system to mark a special bit in the delta record and inform the garbage collector to treat this bit as a special case.

3.4.2 Multi-Block Approach

A second design option, as shown in Figure 3.11, allows a DataTable to have multiple blocks of different schema headers. Originally, a DataTable maintains a list of blocks used to store tuples in the same schema. Contrast with this design where a DataTable maintains several lists of blocks each of which stores tuples in separate schemas. As in our approach, the system assigns a transaction a version number for each table so that the DataTable can identify the correct set of blocks.

Pros

This approach pushes the logic of multi-versioning down to the physical table level, so the SqlTable does not need to maintain a map. Also, it is more space-efficient since the system only creates a block instead of a new table for a schema change.

Cons

It breaks the assumption that each DataTable represents a physical table of one schema. Pushing all the logic down to block-level may potentially slow the system down since much optimization can be placed in the DataTable when there is only one type of blocks. For example, if there is only one list of blocks, a DataTable can use the compare-and-swap

instruction to append a block at the beginning. Otherwise, the DataTable needs a latch to protect multiple lists of blocks, which can result in performance penalty.

Chapter 4

Evaluation

We next evaluate our approach for performing online schema changes. This chapter evaluates the lazy schema change method in Terrier. We use microbenchmarks with a set of mixed types of transactions. We compare against blocking schema changes.

4.1 Experimental Framework

The experimental framework focuses on isolating external factors from the system to highlight differences in performances caused by workloads and implementation. The machine used for the experiment contains 6 cores from Intel Xeon CPU E5-2420 v2 @ 2.20GHz with 32 GB of memory and hyper-threading. We run transactions as stored procedures in the system. All experiments and benchmarks run on the Terrier system. The systems uses a concurrent map from Intel Threading Building Blocks library [17], and all experiments and benchmarks use Google Benchmark library [13]. We run all microbenchmarks multiple iterations and report the average results.

4.2 Workloads

4.2.1 SQL Operation Performance

We measure the performance of a SqlTable by running 6 different types of operations on a table. The table is initially empty and contains two columns of 8-byte long integers.

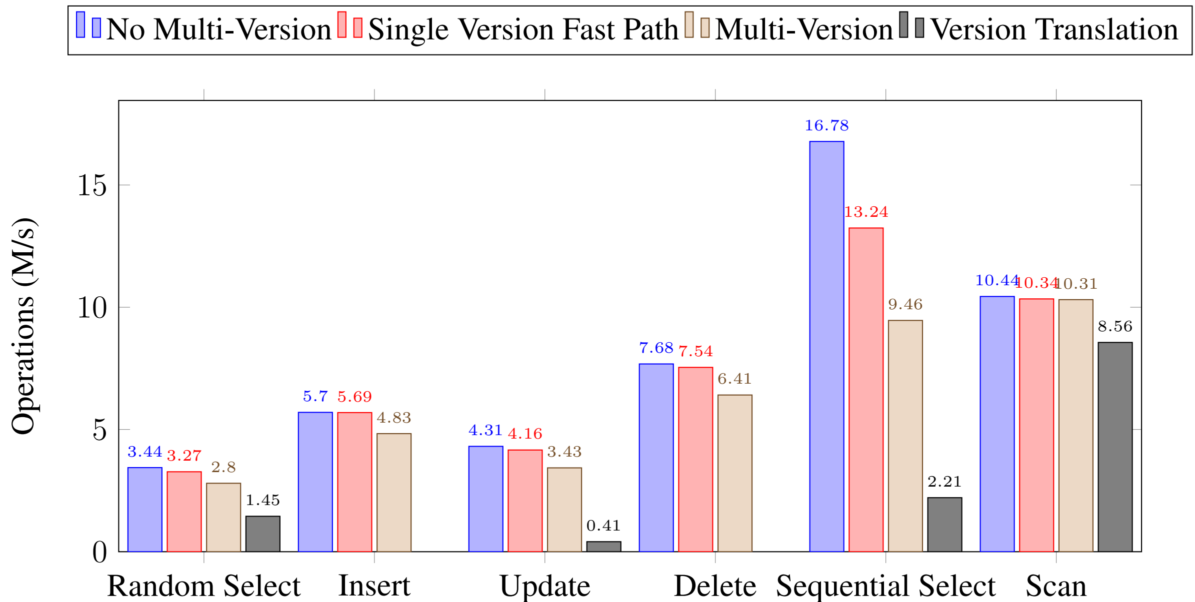

Figure 4.1: Performance of SQL operations under different states of the system

Random Select: It inserts 10 million tuples into the table. Then it starts the benchmark, randomly chooses a valid physical address of a tuple, and Selects all attributes of this tuple.

Insert: It continuously inserts a pre-generated tuple into the table 10 million times.

Delete: It inserts 10 million tuples into the table. Then it starts the benchmark and sequentially deletes all 10 million tuples from the beginning.

Update: It inserts a tuple in the table. Then it starts the benchmark and updates the tuple using pre-generated values 10 million times.

Sequential Select: It inserts 10 million tuples into the table. Then it starts the benchmark and sequentially Selects all attributes of a tuple from the beginning.

Scan: It inserts 10 million tuples into the table. Then it starts the benchmark and scan blocks of 1000 tuples sequentially from the beginning.

For each type of operations except for Insert and Delete, we report four results. Figure 4.1 shows the performance of six types of operations under four different states of the system.

No Multi-Version: This is the performance when the system does not have the lazy schema change implementation.

Single Version Fast Path: The is the performance when the system implements the lazy schema change and the table has only one schema version.

Multi-Version: This is the performance when the table has two schema versions, and all the tuples have been migrated to the newest version. The table still contains two columns of 8-byte long integers in the second schema.

Version Translation: This is the performance when the table has two schema versions, and all the tuples are still in the old DataTable. The table still contains two columns of 8-byte long integers in the second schema. Both Selecting and Updating a tuple require version translation in this scenario. We do not report numbers for Insert and Delete since they do not require version translation.

Figure 4.1 shows that the lazy schema change implementation added to the system produces little overhead for most of the SQL operations except for Sequential Select. When schema updates are rare, our lazy schema change implementation has little overhead on tables whose schemas are never changed. However, comparing performance on the single version table and the multi-version table, we observe that going through the concurrent map in the SqlTable introduces 14-28% overhead for each operation. It indicates that a schema update on a table permanently slows down operations by more than 20% if the system does not clean up the chain of different versions of DataTables. Figure 4.1 also shows that the primary cost of the lazy schema change occurs when a transaction access tuples in the new schema that do not exist in the new DataTable. Those tuples are still in the old table with the old schema. Reading and writing to such tuples can potentially cause version translation between two schemas. However, these costs occur only once for each tuple if the system migrates the tuple when a transaction accesses it. The migration cost is unavoidable. In our lazy schema change design, the migration is driven by the workload.

4.2.2 Performance Overhead

This experiment measures the impact on the throughput of Update transactions when a schema change occurs. The table is initially populated with 10 million tuples containing ten columns of 8-byte long integers. An Update transaction performs one Update operation on a tuple, which updates all ten columns. The transaction allocates memory space and generates new data on-the-fly. A schema update occurs 10 seconds after the experiment begins. The new schema still contains ten columns of 8-byte long integers. When

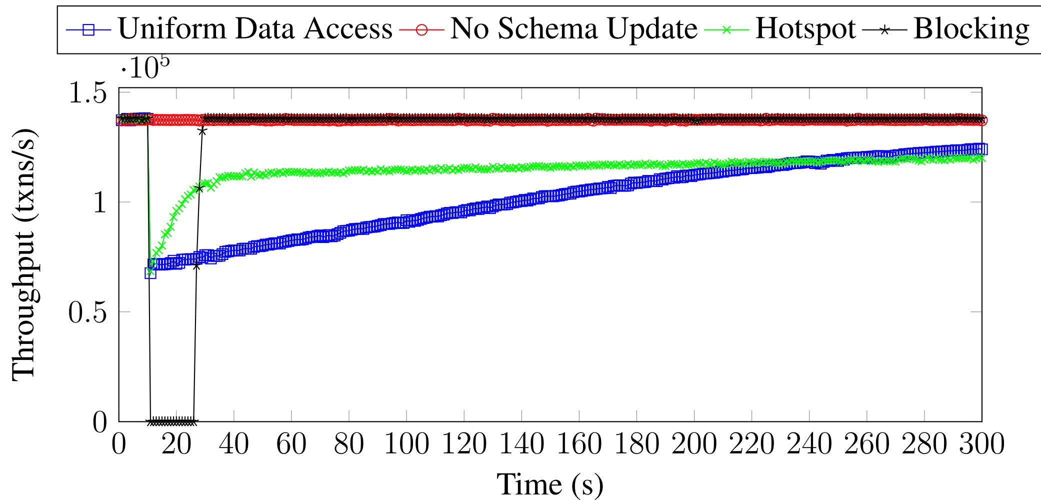

Figure 4.2: Change in throughput for Update transactions overtime after one schema change

the experiment begins, a single thread executes Update transactions in a closed loop for 5 minutes. We measure the performance in four different cases. Figure 4.2 shows the throughput of Update transactions under the four scenarios.

No Schema Update: No schema update occurs in the experiment in this case.

Uniform Data Access: Each Update transaction uniformly updates a tuple among 10 million tuples.

Hotspot: Each Update transaction has 80% chance to pick a random tuple in the hotspot and 20% chance to update a random cold tuple. The hotspot is 5% of tuples in the table in a contiguous range.

Blocking: The system performs a blocking schema update in this case. When the schema update occurs, it locks the table blocks all running transactions. It copies tuples from the old table to a new table and releases the lock.

The blue and green lines in Figure 4.2 show that our implementation supports non-blocking schema changes. Moreover, a schema update initially drops the throughput since Update operations on tuples in old DataTable require version translation. As more and more tuples are moved to the latest DataTable, we observe that the throughput gradually increases. It shows that tables with hotspot recover from performance degeneration at a

much faster speed. After 5 minutes, the system has moved approximately 94% tuples to the new DataTable in the Uniform Data Access case, and 52% tuples in the Hotspot case.

This experiment also shows the problem that the table suffers from permanent performance penalty after a schema update because of the loss of single version cache, which is a disadvantage of the current implementation of the lazy schema change. Comparing the throughput against the blocking schema change mechanism, we believe that tuples should be moved to a new DataTable in a background thread to make the multi-version penalty transient and under the system's control. We defer this problem as future work.

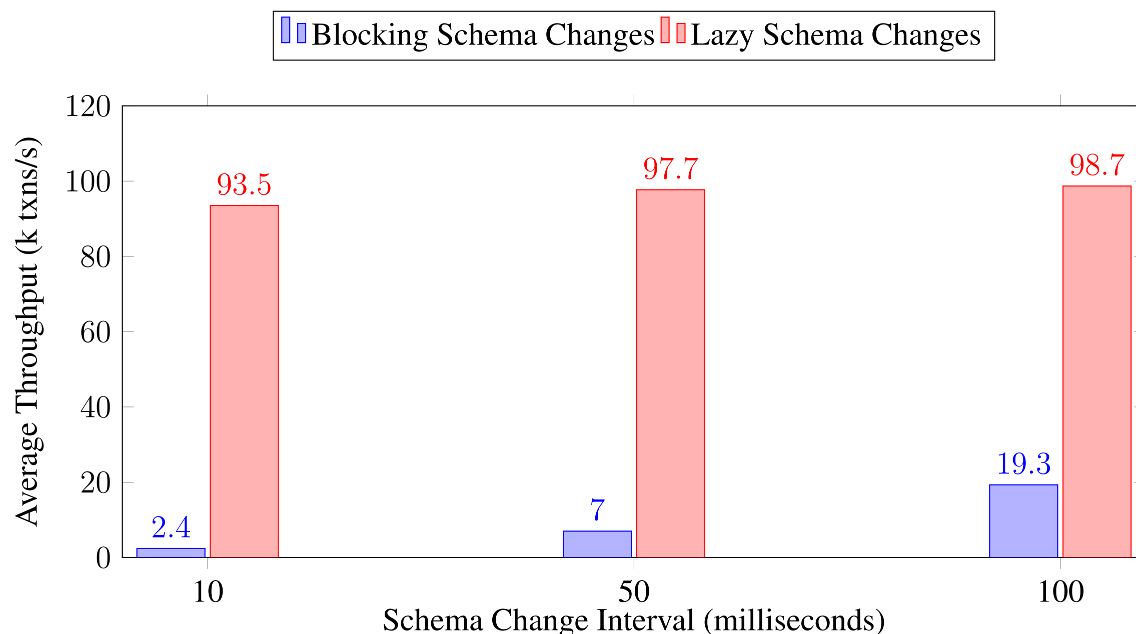

Figure 4.3: Average throughput under high-frequency schema changes workload

4.2.3 High-Frequency Schema Change

In this benchmark, we measure the transaction throughput under frequent schema changes. The table is initially populated with 10 million tuples of two columns of 8-byte long integers, with 5% hotspot. The workload consists of three types of transaction, Select transactions, Insert transactions, and Update transactions. We only report committed transactions. The benchmark triggers a schema change every 10/50/100 milliseconds and measures the average throughput over 120 seconds. It compares lazy schema changes with blocking schema changes.

Select Transactions: Select transactions make up 70% of all transactions. Each transaction performs one Select operation. It has 80% chance to pick a random tuple in the hotspot and 20% chance to select a random cold tuple.

Insert Transactions: Insert transactions make up 20% of all transactions. Each transaction inserts one tuple into the table with no null values.

Update Transactions: Update transactions make up 10% of all transactions. Each transaction performs one Update operation. The transactions generate new data on-the-fly. It has an 80% chance to update a random tuple in the hotspot and 20% chance to update a random cold tuple.

The results in Figure 4.3 show that our implementation of lazy schema change can handle highly frequent schema changes compared to the blocking schema change. The frequency of schema changes has a significant impact on the transaction throughput of the blocking schema changes, while the performance of lazy schema changes is relatively immune to the frequency of schema changes. We achieve approximately 40x higher average throughput compared to the blocking schema change when a schema change is triggered every 10 milliseconds.

Chapter 5

Related Work

This section describes how existing commercial, open source DBMSs support schema changes and relevant research on non-blocking schema changes.

5.1 Existing Systems

MySQL 8.0

Dictionary tables in MySQL 8.0 contain metadata required to execute SQL queries [23]. MySQL 8.0 stores all dictionary information in transactional storage in InnoDB storage engine. It implements catalog as views on dictionary tables, allowing optimization of catalog queries [22]. It eliminates costs such as the creation of temporary tables for each catalog query during execution on-the-fly and scanning file-system directories. MySQL with InnoDB engine supports non-blocking schema changes for ADD COLUMN, Change index option, Rename table, SET/DROP DEFAULT, MODIFY COLUMN, and Add/drop virtual columns by specifying INSTANT algorithm in the queries [21]. Non-blocking schema changes are made in the data dictionary, so they do not require acquiring locks as there is no change to the underlying data. Each record has a flag that is stored in **info_bits**. It uses the **info_bits** to track if the record was created after first instant ADD COLUMN or not. With this extra information, it is now possible for the ADD COLUMN operation to be executed instantly, without modifying any of the rows in the table. After an instant ADD COLUMN is issued, any update to the table writes rows in the new format along. The default values are looked up from the data dictionary. Contrast this approach with our approach where updates go to the new format only when they update some attribute in the

new schema; otherwise, the table still writes updates in the old schema.

Non-blocking operations are limited in MySQL 8.0. It only supports adding columns in one statement, that is if there are other non-INSTANT operations in the same statement, it cannot be done instantly. Also, it only supports adding columns at last, not in the middle of existing columns, whereas our approach allows adding columns in the middle or rearranging columns since it creates a new table with a new schema.

PostgreSQL 11

PostgreSQL 11 stores schema information in pg_tables in the catalog. In PostgreSQL, all schema changes are blocking. PostgreSQL uses table-level locks stored in the pg_locks table [33]. There are various types of locks in PostgreSQL, including ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE. Depending on the type of schema updates, schema updates acquire different types of locks. Most of Data Definition Language (DDL) operations, which define the structures in a database, require an exclusive lock, which blocks other data manipulation language (DML) operations, which modify data in a database. There is no concurrency among DDL and DML operations. The table is not available while the DDL operations are carried out and it causes downtime. The use of explicit locking can increase the likelihood of deadlocks. PostgreSQL 11 automatically detects deadlock situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete.

MemSQL 6.7

MemSQL is a distributed, in-memory, relational DBMS. Schema information is stored at every node. MemSQL 6.7 supports online ALTER TABLE as it does not require doubling the disk or memory use of the table while executing, does not lock the table or prevent querying it for long periods, and does not use excessive system resources [34]. It changes only the metadata of the table on every node in the cluster with a short-lived table lock. The distributed lock is required to synchronize all nodes in the cluster so they can start displaying the new column at the same time. New memory space is set up at the time of the metadata change to allocate rows for the new table schema. It serves SELECT queries by generating the default values on the fly. Write queries insert new rows into the new table and the old table. In addition, a separate thread begins transferring the row data from the old format to the new format. MemSQL's approach is similar to our approach as both create a separate space with a new schema. One difference is that our approach can write

updates in the old memory space. The other difference is that the transferring process in our approach is driven by workloads, not by a separate thread.

MemSQL does not support for rollbacks once ALTER TABLE starts. Because it requires distributed locks to change metadata, it may need to wait for long running queries to finish to acquire locks.

5.2 Research

Schema Evolution Benchmarks

Prior work has developed schema evolution benchmarks for both soft and hard schema change [39] to demonstrate its importance and challenges. The paper aims to introduce two benchmarks. The first benchmark is for systems that support soft schema change, which allow new transactions and old transactions (i.e., transactions based on an old schema version) to access the same database concurrently [9]. The benchmark is derived from real database schemas across different versions of MediaWiki, which is widely used for web information systems like Wikipedia. It also contains real and synthetic queries that access the database using any schema version. The second benchmark is for systems that support hard schema change, which abort any old transactions after the schema change transaction commits [43]. It is based on the TPC-C [7] workload implemented as stored procedures. The goal of the second benchmark is to run Data Definition Language (DDL) and Data Manipulation Language (DML) operations concurrently while minimizing the performance impact of schema changes on TPC-C transactions. Each schema change transaction includes DDL operations that modify the schema as well as updates to stored procedures to adapt TPC-C transactions to the new schema.

Online Schema Evolution Through Query Rewriting

PRISM: The PRISM workbench [8] automatically rewrites the legacy query that accesses an old version of the schema to adapt to the latest version of the schema. It does not keep old data from the previous schema version. In other words, every time the user updates the schema, there is data migration, but it does keep all schema changes in a separate metadata DB. However, it does not support read history records or a temporal query, and rewritten queries run with a permanent overhead and are on average 4.5 times slower than original queries.

PRIMA: Moon et al. [20] describes the PRIMA system, which is a transaction-time DBMS that supports schema evolution. It has a different architecture than that of PRISM. It stores historical data and historical schema info in a separate document-oriented database to support temporal query and maintains a current or snapshot database in the relational model at the same time to support regular query. A user expresses their query based only on the current schema version. Then PRIMA automatically translates this input query into equivalent queries against all applicable schema versions and executes them against the databases underlying such schemas. The system uses the same technique for query rewriting in PRISM, including schema mapping with changes. One drawback of this approach is that the system can exhibit response lags of up to 30 seconds per schema change while computing mappings between schema versions [20, 26].

Updateable Views: InVerDa creates schema versions as views and uses triggers to allow queries to update using any schema version [16]. Like PRISM and PRIMA, the system also allows accesses to data any valid schema. InVerDa does not need to copy data when it creates a new schema version (only creates view and triggers). Instead, a new table is created, if necessary, to contain newly created columns in the new schema version, whereas the old data do not need to be moved or copied.

Online Schema Evolution Through Copying

Lazy Copying: Neamtiu et al. [26] allows schema change transactions to commit before actual schema change has completed. Although the system may not have moved the data to the new schema, queries that access the data are not blocked. These queries transform the data into the new format on the fly. The system generates a thread that transforms tuples from the old schema to the new schema in the background. One disadvantage of this approach is that each schema update causes migration of the entire table in the background. This is too inefficient when the schema of a table frequently changes.

External Tools: Ronstrom [39] creates a tool that operates on top of DBMSs that creates new tables and copies all data from the old tables in a single transaction when migrating to a new schema version. It adds triggers on the old table to avoid missing any concurrent updates to the data. Callaghan [5] developed a tool on top of MySQL to allow online schema changes.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented an implementation of non-blocking schema change, the lazy schema change approach, via schema multi-versioning. This approach migrates tuples lazily on demand from an old table to a new table, and it relies on maintaining multiple DataTables in a SqlTable and keeping a version table in the catalog. Furthermore, we study isolation levels regarding schema-change transactions. We show that the system needs serializability between schema change transactions and any other concurrent transactions to avoid inconsistency cased by unsafe schema changes, but it allows concurrent non-schema-change transactions continue to run under a lower isolation level previously set in the system. Moreover, we show that lazy schema changes can handle frequent schema changes. The performance penalty after a schema change can reduce fast on tables that have hot tuples. We learn that, although we desire non-blocking schema changes via multi-versioning, we should be careful that they need to maintain the performance for the single-version common case.

In the future, we would like to integrate the lazy schema change with other existing table migration techniques. We want to implement version compaction that migrates tuples from old DataTables to the newest DataTable and shrink the schema version chain so that a multi-version table can be cleaned up and return to a single version state. Then, operations on the table can remain efficient. Version compaction can happen concurrently in the background like in most other systems, or it can manually start and block concurrent transactions until it finishes. Another interesting direction is to use Machine Learning to determine when to conduct version compaction intelligently. Also, when the workloads drive data migration, we would like to evaluate different migration policies.

Bibliography

[1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, November 2002. ISSN 1066-8888. doi: 10.1007/s00778-002-0074-9. URL `http://dx.doi.org/10.1007/s00778-002-0074-9`. 2.2.1

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL `http://doi.acm.org/10.1145/568271.223785`. 2.1.2, 2.1.3

[3] Philip A Bernstein and Nathan Goodman. Multiversion concurrency controltheory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983. 1.1, 2.1.3

[4] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, December 2009. ISSN 0362-5915. doi: 10.1145/1620585.1620587. URL `http://doi.acm.org/10.1145/1620585.1620587`. 2.1.2, 3.3.1

[5] Mark Callaghan. Online schema change for mysql. `https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932`, 2010. 5.2

[6] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks*, pages 263–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-642-59412-0. doi: 10.1007/978-3-642-59412-0_16. URL `https://doi.org/10.1007/978-3-642-59412-0_16`. 2

[7] The Transaction Processing Council. Tpc benchmark c. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`, 2010. 5.2

[8] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453939. URL `http://dx.doi.org/10.14778/1453856.1453939`. 2, 5.2

[9] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *In ICEIS*, 2008. 1.1, 5.2

[10] CMU DB. Terrier (it's a temporary name). `https://github.com/cmu-db/terrier`, 2018. 1.1, 2.2

[11] Wikimedia Foundation. Mediawiki 1.5 upgrade. `https://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade`, 2005. 1.1

[12] Google. G suite service level agreement. `https://gsuite.google.com/intl/en/terms/sla.html`, 2018. 1.1

[13] Google. Google benchmark. `https://github.com/google/benchmark`, 2018. 4.1

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902. 2.1.2

[15] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL `http://doi.acm.org/10.1145/289.291`. 1.1

[16] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1101–1116, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3064046. URL `http://doi.acm.org/10.1145/3035918.3064046`. 5.2

[17] Intel. Intel threading building blocks documentation. `https://software.intel.com/en-us/node/506171`, 2018. 4.1

[18] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in sap hana. In *Proceedings of the 2016 International*

Conference on Management of Data, SIGMOD '16, pages 1307–1318, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2903734. URL http://doi.acm.org/10.1145/2882903.2903734. 2.2.6

[19] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313, April 2013. doi: 10.1109/ICDE.2013.6544834. 2.2, 2.2.5

[20] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453952. URL http://dx.doi.org/10.14778/1453856.1453952. 2, 5.2

[21] MySQL. Innodb now supports instant add column. https://mysqlserverteam.com/mysql-8-0-innodb-now-supports-instant-add-column/, 2018. 5.1

[22] MySQL. Improvements to information_schema. https://mysqlserverteam.com/mysql-8-0-improvements-to-information_schema/, 2018. 5.1

[23] MySQL. A new data dictionary for mysql. http://mysqlserverteam.com/a-new-data-dictionary-for-mysql/, 2018. 5.1

[24] Priya Narasimhan. No downtime for data conversions: Rethinking hot upgrades, 2009. 1.1

[25] Iulian Neamtiu and Tudor Dumitras. Cloud software upgrades: Challenges and opportunities. *2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10, 2011. 1.1

[26] Iulian Neamtiu, Jonathan Bardin, Md. Reaz Uddin, Dien-Yen Lin, and Pamela Bhattacharya. Improving cloud availability with on-the-fly schema updates. In Bipin V. Mehta, Nikos Mamoulis, Arnab Bhattacharya, and Maya Ramanath, editors, *19th International Conference on Management of Data, COMAD 2013, Ahmedabad, India, December 19-21, 2013*, pages 24–34. Computer Society of India, 2013. URL http://comad.in/comad2013/pdfs/neamtiu.pdf. 5.2, 5.2

[27] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2749436. URL http://doi.acm.org/10.1145/2723372.2749436. 2.2.1, 2.2.2

[28] Oracle. Online transaction processing (oltp). https://docs.oracle.com/cd/A87860_01/doc/server.817/a76992/ch3_eval.htm#2680, 2018. 3.1.3

[29] Christian Plattner, Andreas Wapf, and Gustavo Alonso. Searching in time. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 754–756, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142578. URL http://doi.acm.org/10.1145/1142473.1142578. 2

[30] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367523. URL http://dx.doi.org/10.14778/2367502.2367523. 2.1.2

[31] PostgreSQL. System catalogs. https://www.postgresql.org/docs/current/catalogs.html, 2018. 2.2.3

[32] PostgreSQL. Serializable snapshopt isolation. https://wiki.postgresql.org/wiki/SSI, 2018. 2.1.2

[33] PostgreSQL. Explicit locking. https://www.postgresql.org/docs/current/explicit-locking.html, 2018. 1.1, 2, 3.1.1, 5.1

[34] Adam Prout. Making painless schema changes. https://www.memsql.com/blog/painless-schema-changes/, 2015. 5.1

[35] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. Online, asynchronous schema change in f1. In *VLDB*, 2013. 2

[36] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001. ISSN 1066-8888. doi: 10.1007/s007780100057. URL http://dx.doi.org/10.1007/s007780100057. 2

[37] D.P. Reed. *Naming and Synchronization in a Decentralized Computer System*. AD A061. Massachusetts Institute of Technology, Laboratory for Computer Science, 1978. URL https://books.google.com/books?id=mUKvQgAACAAJ. 2.1

[38] RocksDB. Myrocks limitations. https://www.percona.com/doc/percona-server/LATEST/myrocks/limitations.html, 2018. 1.1, 2, 3.1.1

[39] M. Ronstrom. On-line schema update for a telecom database. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 329–338, Feb 2000. doi: 10.1109/ICDE.2000.839432. 5.2, 5.2

[40] SQLite. Alter table. https://www.sqlite.org/lang_altertable.html, 2018. 1.1, 2, 3.1.1

[41] Yannis Velegrakis, Rene J. Miller, and Lucian Popa. - mapping adaptation under evolving schemas. In Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer, editors, *Proceedings 2003 VLDB Conference*, pages 584 – 595. Morgan Kaufmann, San Francisco, 2003. ISBN 978-0-12-722442-8. doi: https://doi.org/10.1016/B978-012722442-8/50058-6. URL http://www.sciencedirect.com/science/article/pii/B9780127224428500586. 2

[42] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 473–488, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196895. URL http://doi.acm.org/10.1145/3183713.3196895. 2.2, 2.2.5

[43] L. Wevers, Matthijs Hofstra, Menno Tammens, Marieke Huisman, and Maurice van Keulen. A benchmark for online non-blocking schema transformations. In *Proceedings of 4th International Conference on Data Management Technologies and Applications, DATA 2015*, pages 288–298. SCITEPRESS, 7 2015. ISBN 978-989-758-103-8. doi: 10.5220/0005500202880298. eemcs-eprint-26142. 5.2

[44] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10 (7):781–792, March 2017. ISSN 2150-8097. doi: 10.14778/3067421.3067427. URL https://doi.org/10.14778/3067421.3067427. 2.1.4, 2.2.1, 2.2.5

[45] Cong Yu and Lucian Popa. Semantic adaptation of schema mappings when schemas evolve. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1006–1017. VLDB Endowment, 2005. ISBN 1-59593-154-6. URL `http://dl.acm.org/citation.cfm?id=1083592.1083708`. 2