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 Con icts 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 de nition of a table. It de nes 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 x 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 nishes installing the schema change. A transaction is a logical unit that is

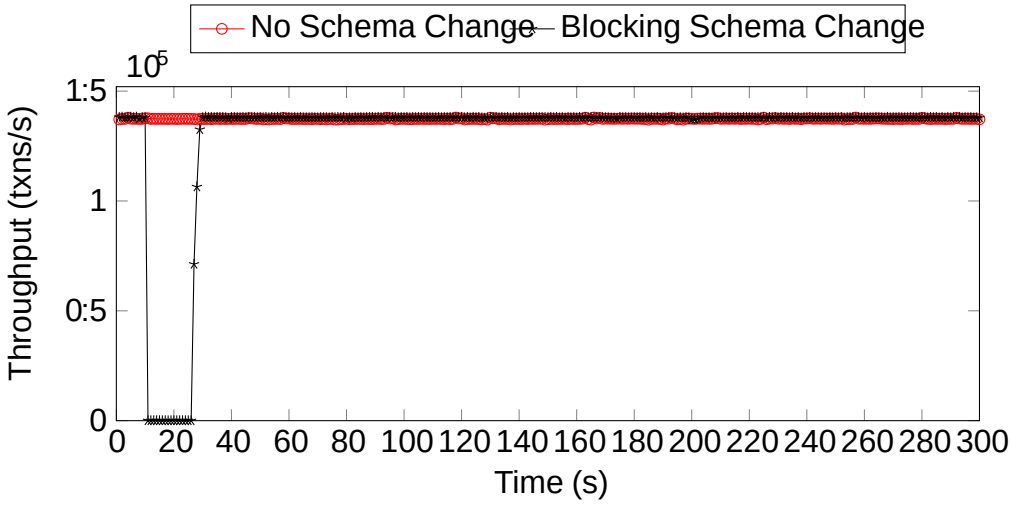

Figure 1.1: Unavailability of the table caused by a non-blocking schema change

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 signi cantly [12, 24]. In a workload that consists of many read and write transactions, blocking schema changes reduce transaction throughput signi cantly. 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 sacri cing 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 veri es 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 speci c 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 modi cation. 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 exibility 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 rst 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 con icting 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 de ne 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 modi ed 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 con ict 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 DBMS's 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 modi ed 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 speci cally, each block has a xed 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 signi cant 44 bits inside a 64-bit pointer are suf cient 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 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 modi ed, as well as the timestamp of the transaction that modi ed 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 speci c version of the tuple based on its timestamp.

On a tuple update, the updating transaction rst 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 con ict.

2.2.3 Catalog

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

dexes, users, and user groups are stored. More speci cally, 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 modi ed 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 speci c 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 ef ciently. 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 de ne garbage. A set of versions for each record is identi ed 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 nished. 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 nished transactions.

Chapter 3

Non-Blocking Schema Change

This chapter discusses the behavior of non-blocking schema changes. It focuses on one speci c 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 y, 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 modi ed 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. *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.