Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 56

Systems Group, Department of Computer Science, ETH Zurich

Distributed Snapshot Isolation on RamCloud

by

Thomas Etter

Supervised by

Prof. Dr. Donald Kossmann
Simon Loesing
Markus Pilman

April 2012 - September 2012

inf | Informatik
Computer Science

Abstract

In the last few years, many researchers have come to the conclusion that traditional relational database management systems (RDBMS) can no longer scale to the requirements of today. The traditional setup of RDBMSs relies on having a central processing server which stores all data on disk.

We will replace the disk by a distributed storage system which uses RAM to achieve faster transfer rates and shorter access times. RAM and fast networks have gotten much cheaper over the last years, while at the same time, CPU speed improvements per year especially for single-thread performance have been much smaller than before. We also do not know yet how we can use additional threads of multi-core CPUs to efficiently accelerate DB systems. This moves the bottleneck from the disk to the processing node. In this thesis we want to explore ways to distribute transactions across multiple servers and different layers.

We use as the base a MySQL storage-engine developed at ETH Zurich which runs on top of RAMCloud, a distributed key-value store with very short access times.

The goal of this master thesis is to implement a multi-version concurrency control algorithm and evaluate a system that uses snapshot isolation (SI) as a mean of isolating transactions. The main challenges are the adaption of a B-tree with garbage collection to store the data in and the adaption of the DB system (MySQL) to work with multiple nodes executing transactions at the same time. For persistence and fault-tolerance, we implement Write-Ahead-Logging (WAL). We will also evaluate the influence of different concurrency control algorithms on performance.

Contents

1 Introduction **2**
1.1 Motivation . . . 2
1.2 Problem Statement . . . 2
1.3 Structure of the Thesis . . . 3

2 Background **4**
2.1 Concurrency Control in Relational Databases . . . 4
2.1.1 Locking . . . 4
2.1.2 Snapshot Isolation (SI) . . . 4
2.2 Distributed Database and Cloud Storage Systems . . . 5
2.2.1 Systems Using Sharding . . . 5
2.2.2 Shared-Disk Approaches . . . 6

3 Distributed Snapshop Isolation **8**
3.1 The Big Picture . . . 8
3.2 Components . . . 8
3.2.1 Query Processing Nodes . . . 8
3.2.2 Storage Nodes . . . 9
3.2.3 Commit Manager . . . 9
3.2.4 Lock Server . . . 10
3.3 Transaction Processing . . . 10
3.3.1 Workflow . . . 10
3.3.2 Versioning . . . 11
3.3.3 Commit Strategy . . . 11

4 Implementation **12**
4.1 Architecture . . . 12
4.1.1 MariaDB . . . 12
4.1.2 RAMCloud . . . 12
4.2 Data Structures . . . 13
4.2.1 Time Information . . . 13
4.2.2 Row Data . . . 13
4.2.3 Indexes . . . 14
4.2.4 Write Ahead Log (WAL) . . . 15
4.3 Caching . . . 16

5 Experimental Evaluation **18**
5.1 Methodology . . . 18
5.2 Experiments and Results . . . 20
5.2.1 Write-Intensive Workload . . . 20
5.2.2 Read-Intensive Workload . . . 21

6 Conclusion **22**
6.1 Summary . . . 22
6.2 Future Work . . . 22

1 Introduction

1.1 Motivation

Over the last few decades, relational database management systems (RDBMS) have become the dominant form of structured storage in various application domains. Because they are based on tables, the basic concepts are easy to understand.

They also typically give ACID guarantees for transactions which facilitate reasoning about correctness of programs and are essential for building fail-safe and mission-critical systems. These guarantees have led to a design with one master server, as networks have been plagued by low bandwidth and high latencies. Therefore the use of one machine proved faster.

To run the database distributed on multiple servers, many newer systems sacrifice one or more of the ACID properties. Most NoSQL databases offer weaker transactional guarantees, while some clustered SQL databases only provide isolation as long as all data touched by a transaction is on the same server. These weaker constraints allow systems to provide an increased throughput, as well as the ability to scale out to very large datasets and clusters consisting of hundreds of machines.

For workloads that can tolerate their limitations, these solutions offer great scalability and elasticity. There are however workloads that cannot make these trade-offs and which are often currently run using expensive hardware.

We suggest a new solution that is both scalable and elastic while still providing ACID guarantees and supporting standard SQL queries.

1.2 Problem Statement

Previous work [12] has demonstrated that it is possible to implement a RDBMS on top of a key-value store.

Current in-memory key-value stores can provide great scale-out behaviour and excellent access times, making them well-suited as storage systems.

In traditional single-server databases, multi-version concurrency control (MVCC) often performs better than pessimistic strategies (locking). We want to implement snapshot isolation on a key-value store. To do that, we implement a RDBMS storage engine that uses MVCC on top of RAMCloud [10], a in-memory key-value store designed for low latency networks.

Such a setup allows to operate multiple database servers on the same data while still providing the same isolation level between transactions. This allows database servers to be elastically added and removed. As snapshot isolation has fewer non-parallelizable components than other approaches, we expect a well-tuned implementation to scale better than a locking strategy.

Contributions of this thesis:

- A distributed implementation of the snapshot isolation protocol.
- This protocol is integrated into a MariaDB storage engine running on top of an in-memory key-value store, namely RamCloud [10].
- An experimental evaluation comparing our implementation to traditional approaches.

1.3 Structure of the Thesis

The thesis is structured as follows:

- Chapter 2 gives an overview over the two most used solutions for concurrency control and over some currently existing distributed database systems.
- Chapter 3 discusses the components needed for building our system and how they fit together.
- Chapter 4 provides details on architecture and the data structures used in the system.
- Chapter 5 evaluates the system in a benchmarking environment and compares its performance to a system using another approach to concurrency control.
- Chapter 6 summarizes the thesis and gives an overview over possible future enhancements.

2 Background

2.1 Concurrency Control in Relational Databases

To achieve isolation (the I in ACID), any DBMS must provide some kind of concurrency control. In the following sections, we will focus on row-level isolation.

2.1.1 Locking

This is a pessimistic approach to the problem of concurrency control. In a locking DBMS, a lock must be acquired for any operation on the data. There are several levels of locks that can be employed in DBMSs, however this discussion will focus on a simple model with two types of locks: shared locks and exclusive locks.

When reading data, a read-lock is acquired. Because multiple transactions can read the same row without conflicts, there can also be multiple read-locks on the same row. When writing data, write-lock, an exclusive lock, is acquired (only one transaction can have a write-lock on a row). This ensures that no other transaction can read while the row is written and that changes are not overwritten by another transaction.

For complete isolation (full serializability), locks must be held until the end of a transaction. If they are released earlier, transactions could read uncommitted results. E.g. the transactions T_0 and T_1 operate concurrently on the data x. T_0 does the sequence $w_0(x), r_0(x)$, while T_1 does $w_1(x)$. If locks are not held until the end of the transaction, the global history $w_0(x), w_1(x), r_0(x)$ can be produced. The result of T_0's read in this history is not the value written by itself, but the value written by T_1, which would be impossible in a serializable history. This collecting of locks and releasing them at the end is also called two-phase locking (2PL).

A problem with locking are deadlocks. If transaction T_2 first locks x and then y and transaction T_3 first locks y and then x, it can happen that T_2 has the lock on x and T_3 has the lock on y. To progress, each transaction needs the other's lock. The deadlock can only be solved if one of them is aborted. This complicates the logic of locking based databases because there need to be timeouts or cycle detectors.

2.1.2 Snapshot Isolation (SI)

This is an optimistic approach for transaction isolation. Instead of having just one version in the database, also older versions can be retrieved. This implies that multiple versions must be saved or old versions must be reconstructable using logging information.

Each transaction gets a "snapshot" of the database and reads only from this snapshot. A snapshot consists of the set of transactions that have been completed before the current transaction started. When reading, all versions newer than the ones in the snapshot are ignored.

Read-only transactions can therefore run without ever waiting on other transactions. When writing, conflicts can occur when two transactions try to write to the same row. Isolation could also be discussed on a cell level, however we will limit the discussion here to row level conflicts as it is the granularity that

is most common for detecting conflicts. In case of a conflict, one transaction needs to be aborted and restarted to resolve this conflict. To further decrease the possibility of having It would also be possible to implement Strategies used for this can be found in section 3.3.3.

As conflicts are only detected when two transactions write to the same row, anomalies can occur that do not exist in a strictly serializable history. The anomaly that occurs with SI is "write skew". If we have two transactions $T_0 = r_0(x), r_0(y), w_0(x)$ and $T_1 = r_1(x), r_1(y), w_1(y)$, the global history $r_0(x), r_0(y), r_1(x), r_1(y), w_0(x), w_1(y)$ can be produced. It is obvious that this history is not serializable because on x the ordering is $T_1 < T_0$ while on y the ordering is $T_0 < T_1$.

To solve the non-serializability of standard snapshot isolation, transactions can be rewritten before execution to materialize conflicts [7]. This of course only works for transactions which are known before execution time. Detecting cycles or dangerous structures in the dependency graph during execution can also make SI fully serializable [6].

We chose not to implement serializable snapshot isolation because we believe the overhead needed to detect cycles is too big. The isolation level offered by snapshot isolation is sufficient for many workloads and workloads can be statically analyzed and modified to prevent write skew. Databases using the snapshot isolation level have already been widely deployed.

2.2 Distributed Database and Cloud Storage Systems

2.2.1 Systems Using Sharding

Sharding databases partition tables horizontally to improve parallelism. Data is partitioned into "shards", with one shard assigned to each server.

MySQL Cluster [3] is an example of a database that partitions data horizontally to increase throughput. When doing queries that can be handled by a single shard, the scale-up is almost linear. It also offers two-phase locking for queries running on multiple servers.

This provides transparent access to the database, hiding the fact that the database is sharded.

Windows Azure SQL [4] does not offer any facilities for partitioning. Microsoft recommends to implement sharding in the middleware and run multiple independent databases. This is also a popular approach to scale, using most traditional databases (e.g. Facebook and Twitter with MySQL).

H-Store [13] takes a very interesting approach to the problem of partitioning data: All transactions must be known before runtime and the user can give hints on their execution frequencies. This information allows optimization of the partitioning/replication schema so that most transactions can be executed on just one node. If the analysis fails to optimize the schema to have mostly single server executions, performance degrades. Every CPU core is a node in H-Store and single node transactions run sequentially without locking or logging. In order to execute transactions spanning multiple nodes, the whole node is

locked. This makes H-Store unsuited for analytical workloads and it is therefore marketed as an OLTP database.

When most transactions can be handled on a single server, all previously mentioned solutions can offer great scalability and good isolation. However, for transactions spanning multiple servers, they either only guarantee isolation on the individual server or performance degrades.

2.2.2 Shared-Disk Approaches

Google Megastore [5] is a shared-disk data store that has full ACID transactions. A major difference between this and other approaches is that the user can choose which data is placed close together.

An important element are "entity groups". These denote a group of data entries which are located close together and are always updated with full ACID semantics. For reads, the application can specify which isolation level is wanted and also retrieve older values. Two-phase commit is available spanning multiple entity groups, but it is much slower than transactions on just one entity group.

Google Percolator [11] is a shared-disk database that uses a combination of locking and snapshot isolation. A transaction first gets a read timestamp from the "timestamp oracle" and then creates the write set. Locks are stored in the row entries, so the transaction tries writing a lock on all row entries in the write set. If it sees other locks or newer versions than its read timestamp, it aborts. If all locks are successfully acquired, it gets a commit timestamp from the oracle. The updated rows are then written using the commit timestamp, while at the same time deleting the lock that it acquired before.

This design provides good performance on workloads with few write conflicts. On the other hand, it is not well suited for workloads with hot spots, because two writes are needed for each update and the granularity of the versioning is rather coarse, leading to many aborts.

3 Distributed Snapshop Isolation

3.1 The Big Picture

Our architecture is composed of the following components:

- RAMCloud: the storage layer, consisting of storage nodes.
- Query Processing Nodes: each processing node runs a database instance.
- Lock Server: manages the locks for index writes.
- Commit Manager: provides versioning information for the query processing nodes.

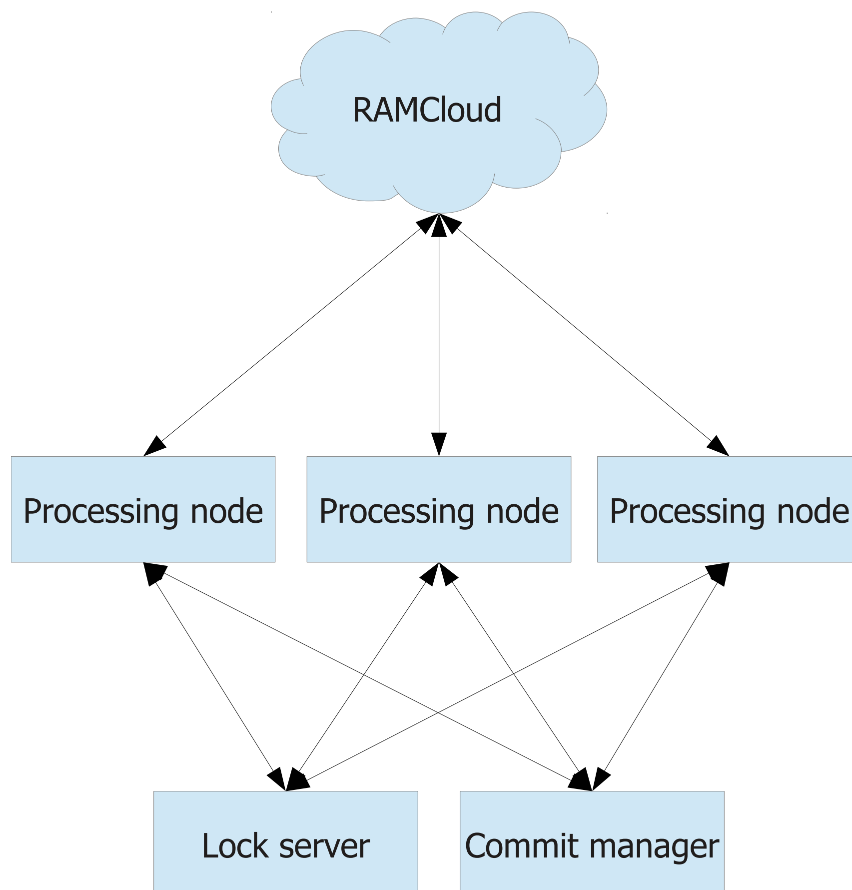

Figure 1: A typical setup

3.2 Components

3.2.1 Query Processing Nodes

The query processing nodes run instances of MariaDB with the modified storage engine. Clients send their queries to one of the query processors. In a production

system, a load balancer could handle this distribution of queries to the nodes, based on the load of the query processors. This would also allow the system to handle the addition and removal of processing nodes.

They also write the status of each transaction into a Transaction Control Block (TCB). The TCB is stored in the transaction control table in the RamCloud.

3.2.2 Storage Nodes

Storage nodes run instances of a key-value store that provides reliable and durable storage with low latency. All data on the storage nodes is kept in memory.

3.2.3 Commit Manager

The commit manager is a server maintaining the set of currently running transactions. It is responsible for keeping track of a global time, so that every transaction gets a unique timestamp. Every transaction also receives a read-set, which consists of the transactions that have committed before it has started. Information about garbage collection is also given to transactions. It has roughly this API:

```
struct SnapshotDescriptor {
    //all transactions up to this have committed
    TransactionTime base_version;

    //transactions that have committed with
        TransactionTime > base_version
    //also contains the transaction's own time
    set<TransactionTime> newly_committed;
};

struct TransactionInfo {
    TransactionTime transaction_id;
    TransactionTime lowest_active_base_version;
    SnapshotDescriptor snapshot;
};

class CommitManager {
public:
    TransactionInfo start();
    void setCommitted(TransactionTime transaction_id);
    void setAborted(TransactionTime transaction_id);
}
```

See also section 4.2.1 for more information about times. A failure of the commit manager cannot be recovered yet. The changes needed to enable recovery are outlined in section 6.2.

3.2.4 Lock Server

Because the current implementation of the B-tree (see: 4.2.3) uses locking to update nodes, a lock server is needed. This lock server would not be needed in a completely lock-free B-tree as suggested in 6.2.

3.3 Transaction Processing

3.3.1 Workflow

The basic states of a transaction are illustrated in Figure 2

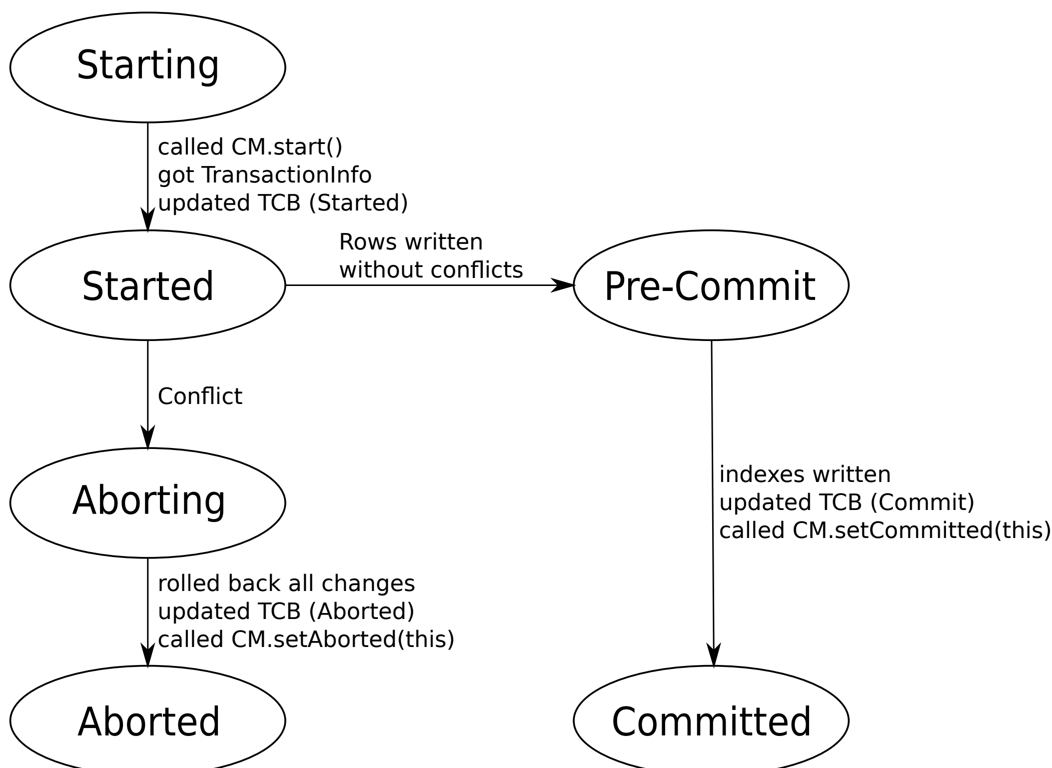

Figure 2: The states of a transaction

- **Starting**: This is the initial state. The transaction fetches a `TransactionInfo` from the commit manager and writes Started to its transaction control block (TCB). This will advance it to the **Started** state.

- **Started**: The transaction executes (reads and writes data on its snapshot). If conflicts occur, it changes to **Aborting**. If no conflicts occur, it changes to **Pre-Commit**.

- **Pre-Commit**: All rows have been written, no more conflicts can occur. The transaction just needs to insert the updated/inserted entries into the indexes.

- **Committed**: The TCB has been written and the commit manager has been notified of our commit. The transaction is done.
- **Aborting**: The transaction roll-backs its writes. It updates its TCB and notifies the commit manager that it has aborted.
- **Aborted**: The commit manager has been notified of our abort. The transaction is done.

A transaction first gets a unique time and a snapshot to read from the commit manager. Next it starts reading entries and creating a list of updates. After the transaction has advanced to a state where it is ready to commit, it tries to apply the updates. If it encounters a conflict while creating the list of updates or while applying an update, it will roll back. Once it has successfully committed or aborted, it will notify the commit manager that it is finished.

3.3.2 Versioning

Versioning ensures that a transaction sees a consistent snapshot of the data and gets the same result if a read is repeated. All version information is managed by the commit manager. When a transaction reads an entry, it searches for the highest version that is still contained in its `SnapshotDescriptor`.

A write-write conflict exists, if when trying to write, a version of a row exists that is not in the snapshot.

Versioning is also relevant for garbage collection as explained in section 4.2.2.

3.3.3 Commit Strategy

There are mainly two strategies of dealing with write conflicts:

FUW (First Updater Wins): If there already is a newer version than the ones contained in the snapshot written by another transaction, abort and rollback the current one.

FCW (First Committer Wins): If there is a version not in the snapshot that is written by another transaction which started earlier, abort the current one. If the current transaction is older, try to abort the other transaction and overwrite its entry. If the newer one has already committed, this will fail and the current transaction will abort. This strategy requires that an older transaction can abort a newer one.

The advantage of FCW is that it is less likely to trigger a deadlock situation where two transactions always restart and rollback. This can be prevented by first creating the write set and then applying it in an ordered way.

Both of these strategies cannot un-commit a transaction in a system using snapshot isolation. Care must be taken, that no long-running transaction reads a lot and then writes to one entry which is often written by short-running transactions. Failure to do so can lead to starvation of the long-running transaction, as when it tries to write it will encounter a newer, committed version and has to be aborted.

4 Implementation

4.1 Architecture

4.1.1 MariaDB

Our implementation is based on MariaDB [1], a community-developed branch of the well-known MySQL database [2]. MariaDB integrates multiple bug fixes and performance enhancements not yet included into the MySQL core and thus provides more performance and stability. The MariaDB server, like the MySQL server it originates from, is built in a modular fashion and provides a defined API to implement custom storage engines. To implement the processing instance, we used MariaDB and implemented our own storage engine. Running a MariaDB process with this new storage engine on a server represents a query processing node as described in section 3.2.1.

The MariaDB core takes care of query parsing and optimization, and calls the storage engine as soon as data is required. For example, to retrieve a range of tuples or even scan an entire table, MariaDB will use an iterator pattern, asking the storage engine to deliver tuples one by one. It is left to the storage engine to efficiently access the data using index structures and buffer pools, as well as manage concurrency control.

4.1.2 RAMCloud

RAMCloud is a key-value store that offers very fast response times and high throughput. It achieves better performance than most other key-value stores by keeping all data in the RAM. It can also use an Infiniband network which has much better throughput and delay than traditional Ethernet networks.

It supports multiple independent key value sets through "tables". A table is identified by its name. When the table is created a 32 bit integer is generated. This integer is then used in all reads/writes to specify the wanted table. It is omitted in samples for better readability. Keys are byte sequences of any length up to $2^{16} - 1$ and values are byte sequences with a maximum length of 1MB.

It also offers durability by replicating the data to multiple nodes and can write data to disk in case of power failure. In case of a node failure, it has very short recovery times [9].

All operations on RAMCloud are atomic. The most important feature of RAMCloud for implementing optimistic concurrency control is its support of CAS (Compare And Swap) style operations. All data in RAMCloud is versioned on a per-key basis. When an entry is read, its version (a 32 bit integer) can be retrieved with it. When writing, a condition can be given that the version number must be the same as the one that was read. This ensures that no updates are lost.

```
//Pseudo-code for updating data with compare-and-swap
   semantics
(version, data) = ramcloud.read(key);
//change data
ramcloud.write_if_same_version(version, data);
```

4.2 Data Structures

This section covers the discussion of the data's storage layout and the data structures needed to manage the version information.

4.2.1 Time Information

The basic data structures for time information can be found in section 3.2.3.

There are three timestamps which are essential to do correct versioning and garbage collection:

- `transaction_id`: the version assigned to the current transaction and with which writes will be marked.
- `base_version`: the version where all transactions with `transaction_id` $\leq$ `base_version` must have committed.
- `lowest_active_base_version`: this value is needed for garbage collection. It denotes the version which is visible to all running transactions or, in other words, the `base_version` of the oldest transaction that is still running.

For these timestamps, the following property must always hold:

`lowest_active_base_version` $\leq$ `base_version` $<$ `transaction_id`

`lowest_active_base_version` can only be equal to `base_version` if there was no transaction running after all versions up to `base_version` had committed.

The `newly_committed` set is not necessary for correctness and the system would also run without it. It is an optimization that allows to include updates of committed transactions that are newer than the `base_version` into the snapshot while an older one is still running. This decreases the number of conflicts, but increases the load on the commit manager which has to maintain this set.

The following table shows an example of snapshots:

TID	base_version	newly_committed	lowest_active_base_version
1	0	-	0
2	0	-	0
3	1	-	0
4	2	-	1
5	2	-	1
6	2	4	1
7	2	4,6	1
8	4	6	2

Using this table, the commit order can be inferred: 1,2,4,6,3

4.2.2 Row Data

Usually the most storage consuming part of a relational database is the data itself, which is stored in rows. The primary key of the row is used as the key for the key-value store. As our key-value store supports variable-length data, both the row data and the blobs are stored together as one value in the store.

A row consists of a time, a state and the row data:

1	VALID	(117,Makrus)

To support access to older versions, there are two strategies:

- Store multiple versions of a row.
- Store only the most current version and reconstruct old versions from the log.

We store multiple versions of a row in one entry as it allows us to have a much smaller log (see also section 4.2.4). Storing only one version would also require a second read for any transaction accessing an older value than the most current one.
When updating a row, we just append a new version:

1	VALID	(117,Makrus)
2	VALID	(117,Markus)

When deleting a row, we also append a new version, but mark it as deleted. This ensures that all transactions starting after we have committed will not return this row.

1	VALID	(117,Makrus)
1	VALID	(117,Markus)
11	DELETED	empty

Garbage Collection Of course, all these additional rows need to be removed again later if we do not want the storage to grow to infinite size, hence garbage collection. To identify which versions can be safely removed, the lowest version which can still be invisible to a transaction needs to be known.

When there are two or more versions which are `<=lowest_active_base_version`, all but the newest version `<=lowest_active_base_version` can be collected. This version always needs to be kept, as a transaction could still be reading it, while the others are invisible to all transactions. See also section 4.2.1 and 4.2.3 for an example.

4.2.3 Indexes

Unlike in sequentially stored data as in many disk-based databases, it is impossible to iterate over all data in a table in RamCloud without using an index. Even for full table scans, the primary key index must be used to identify the location of entries. Therefore indexes are involved in every transaction.

A B^{Link}-Tree is used because it allows for reads to be lock-free, therefore allowing more transactions to operate in parallel on the tree. In the non-leaf nodes, only the indexed key is stored. In the leaf nodes, we additionally store the time of the transaction that inserted it.

As the key for the B-tree, combined keys consisting of the indexed key, a primary key and a time are used. The inclusion of the primary key and time avoids special duplicate handling but needs more space. A future version could use techniques to avoid this overhead (see also section 6.2).

To insert, we operate as on any other B^{Link}-Tree: First the leaf to insert into is searched and locked. If it has changed in the meantime, the correct node to insert can be found by following the right pointer. Once the correct node is

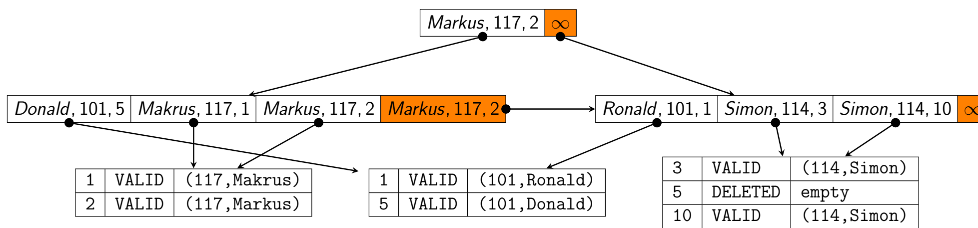

Figure 3: Example of an index

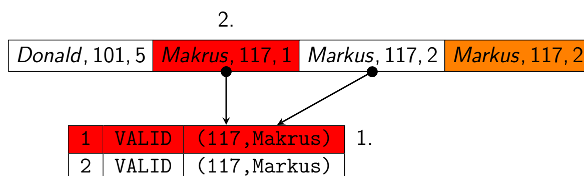

Figure 4: Example of a garbage collection, 1. and 2. denote the two steps of the collection.

found and locked, the insert is executed. If a split happens, the parent is also locked and the pointer is updated. For a more thorough discussion, see [8].

In deletes (deletes only happen in garbage collection), the entry is just deleted from the leaf node. We do not do node merges.

When a row is deleted, no change happens on the tree. Because it cannot be known upon reading an entry whether it is still up-to-date, an entry must always be validated before it is returned as a result of a query. If an old node is encountered that cannot be garbage collected, it is skipped.

Garbage Collection When the index has been read and the validation with the row versions fails, the transaction checks if garbage collection can be done. If the version with the timestamp of the index entry no longer exists in the row versions, it can be safely removed from the tree. That leaf is then removed and no further action is taken as tree nodes are not merged.

As an example, let's assume we're running transaction 8 from the table in section 4.2.1. When the transaction encounters the version 1 of the data for key 117, the data for version 1 can be safely removed. Because 1 and 2 are lower than or equal to the `lowest_active_base_version`, the older version is removed. This is marked as step 1. in figure 4.

If the same or another transaction then reads the entry "Makrus" in the index, it will not find a corresponding entry in the data and will remove the entry from the tree. This is marked as step 2. in figure 4.

4.2.4 Write Ahead Log (WAL)

Every processing node keeps a pool of tables for logging. When a transaction starts, it is assigned one of these tables. The address of that table is stored in the transaction's state in the TCB entry. When the transaction commits, the table is returned to the pool so that it can be reused by other transactions.

Only the addresses (table, primary key) of the modified rows are saved to the logging table. This allows to have a very space efficient WAL.

Rollback When rolling back a transaction, the row versions written by that transaction are deleted. Invalid entries in the index are not removed during rollback. Future transactions will remove them while accessing the index because there is no corresponding row version. The invalid index entries will be removed later when another transaction reads the index as part of the normal garbage collection.

4.3 Caching

To improve throughput on the system, read rows are cached. As conflicts are discovered when writing, old reads can be used as long as they occured after the start of the transaction.

```
//Pseudo-code for caching reads
Time newest_tid;//holds the newest transaction that is
    running on this node
void start_transaction(Time transaction_id) {
    newest_tid = max(newest_tid, transaction_id);
}

Data* read(Key key, Time tid) {
    CacheEntry entry = cache.read(key);
    if (entry.valid() && entry.tid >= tid)
        return entry.data;//there is a version in the
            cache and it is valid for tid
    entry.data = ramcloud.read(key);
    entry.tid = newest_tid;
    cache.write(entry);
    return entry.data;
}
```

Writes are cached similarly.

5 Experimental Evaluation

As of writing this thesis, the lock server is not yet finished which is why the work can not be distributed onto multiple processing nodes yet. The experimental evaluation is therefore limited to running one instance of MariaDB. This also means that the locking service runs in the database engine itself and is faster than the locking server will be as there is no network overhead. Because we have configured the database engine to cache heavily when there is only one node active, the throughput per processing node will also be higher than in a distributed setup.

5.1 Methodology

We use the TPC-W benchmark to test the throughput of our system. The TPC-W benchmark simulates an online bookstore and is a reference that allows to compare full web server stacks. In its original form, the benchmark consists of emulated browsers (EBs) sending requests to application servers which start transactions consisting of queries and updates on the database. There are fourteen request types, each triggering some database queries. The throughput of the system is measured in the number of web interactions per second (WIPS).

We have modified the TPC-W benchmark by removing the application servers and we are executing transactions directly on the database. Also, unlike the original implementation, we increase the issued load linearly over time until the throughput of the system under test (SUT) is reached. In addition, we modified the bestseller request included in the TPC-W benchmark so that it does not go through recent transactions to look for the most often sold items. Instead it just looks up 50 random items. The bestseller request is a very unrealistic workload as in actual deployments, such queries do not get executed more if there are more clients using the system.

As TPC-W is an industry standard, its transactions offer a great workload to compare database systems. Because users will not use web applications that take too long to respond, there are upper limits on the response time in which the transactions must be handled. If the transaction takes longer than that limit it is not considered valid.
For our experiments, we setup the SUT as follows:

- One server runs the workload generator which runs EBs and executes transactions on the processing node.
- A second server (the only processing node) runs both the commit manager and the MariaDB instance. The commit manager generates only very little load so should not impact the MariaDB performance.
- The RAMCloud is deployed on three servers with one running both the coordinator and a RAMCloud server while the other two just run a RAMCloud server each. They are configured without replication so writes are faster than they would be on a production system. The RAMCloud servers are connected over Infiniband for better speed.

So the whole setup for running the benchmarks consists of five servers. Each server has two sockets with an Intel Xeon E5-2609 CPU clocked at 2.40GHz and

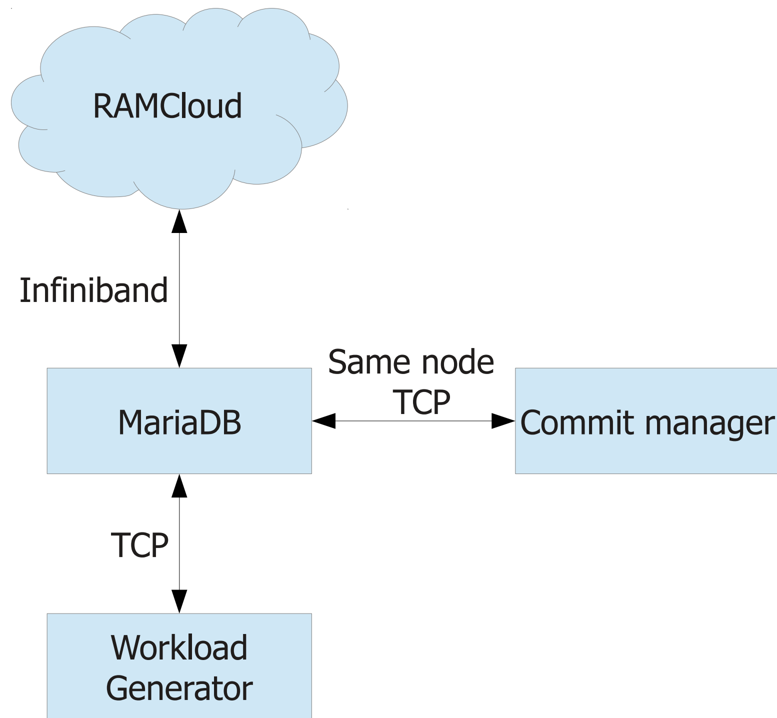

Figure 5: The simplified benchmark setup

128 GB of RAM. The operating system is Debian unstable running on a Linux kernel with version 3.4.4. Figure 5 shows how the components are connected.

We want to measure the throughput in WIPS and compare the storage engine using snapshot isolation to a storage engine using locking which we have also developed. For this we run the benchmark with the load intensifying over time. The workload generator increases the load continually by creating a new EB every 0.2 seconds.

5.2 Experiments and Results

We measure the throughput by counting the valid WIs in windows of 30 seconds. The scale factor (which is equal to the number of items in the store) for the TPC-W benchmark is 10'000. We have chosen the number of emulated browsers for calculating the database size to be constant at 100 so we can just measure throughput without worrying about adjusting it as it would be required for TPC-W compliance.

5.2.1 Write-Intensive Workload

We use the ordering mix of TPC-W that simulates clients ordering items from the store. This is an update intensive workload as the inventory needs to be updated so that no items can be ordered which are not available. 36.55 percent of all requests in this workload contain an update operation.

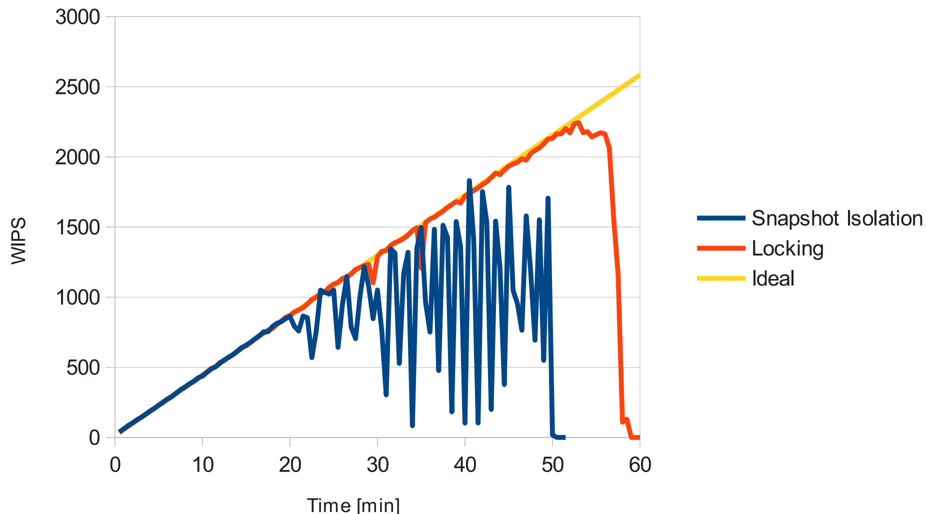

Figure 6: The results with the write-intensive workload

In figure 6 we can see that up to 800 WIPS, both snapshot isolation and locking perform equally well. Then SI starts to variate very much between different time intervals. When looking at detailed logs, we found that the response time for the "Admin Confirm" request is much longer under SI than with locking. The intervals where SI performs bad correlate with the intervals where there are many "Admin Confirm" requests.

The reason for this performance gap is that locking implements also backward scans on indexes, while left-traversals are not yet implemented for SI. This causes MariaDB to create a different query plan for SI than for locking. While the query just traverses the index from right to left in locking, it has to do a full table scan in snapshot isolation.

5.2.2 Read-Intensive Workload

In the read-intensive workload, we use the shopping mix of TPC-W with the same modifications as before. Additionally we also excluded the "Admin Confirm" request which caused SI to perform badly because of a feature not yet implemented instead of showing conceptual differences between the approaches. 15.4 percent of all requests in this workload contain an update operation.

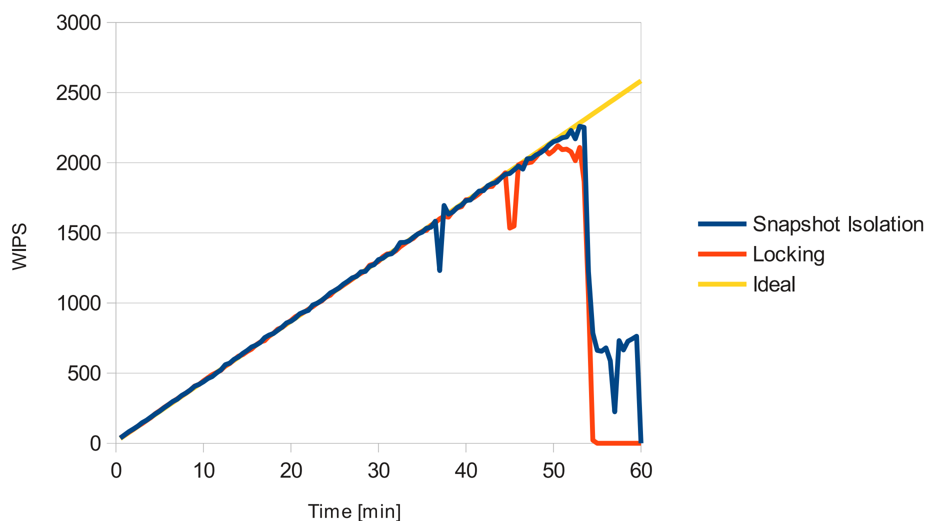

Figure 7: The results with the read-intensive workload

As we can see in figure 7, when we measure the WIPS using the read-intensive workload, snapshot isolation performs about the same as locking. The bottleneck of the system is not on the concurrency control, showing that the SI implementation can easily compete with locking implementations.

6 Conclusion

6.1 Summary

We have shown that it is possible to implement Snapshot Isolation on top of RamCloud. Transactions running on a system using SI run no slower than transactions that run on a system using locking. When thinking about scalability, a system that uses as few centralized components as possible is desirable. In a system using locks, even reading rows requires contacting a central locking service. This is where SI really shines: transactions only have to contact the commit manager once when starting the transaction and once when committing. The central locking service only needs to be contacted for updates that are actually executed and could even be fully eliminated.

6.2 Future Work

No Lock Server The current B-tree implementation still requires that a locking service is present. Contacting this service means additional round-trips and more overhead, especially in low-conflict environments. Replacing it with a lock-free implementation would remove a potential bottleneck and at the same time increase the throughput and scalability for many workloads.

Less Tree Updates Currently, when updating a row, all indexes are updated even if there was no change to the corresponding columns just to have a new timestamp in the tree. This is an overhead that could be eliminated by changing the way that the tree is read and how entries are validated.

Smaller Row Data Entries In this implementation, the whole new version of a row is appended to the row data entry when updating. Instead, just the changes could be written, leading to smaller entries that need to be read and written. This would decrease the load on the network, especially for hot spots where many versions can be active at the same time.

Recovering Commit Manager State The system currently cannot recover from a failure of the commit manager as the it never writes its state to the RAMCloud. While information about committed transactions can be read in the cloud in the form of TCBs reading all TCBs is not feasible. Therefore the commit manager should write the `base_version` to the cloud. Also the maximum timestamp given to a transaction has to be written to the cloud. In order to avoid having to update the persistent state every time a transaction is started, timestamp ranges could be reserved in advance. A similar approach is also taken in [11] which enables them to scale to 2 million timestamps per second.

Other Processing Nodes Because the system runs on a shared disk, there can be different kinds of processing nodes operating on the data. There could for example be specialized nodes optimized for data warehousing queries which would use multiple threads or even multiple servers to cooperatively process a request.

References

[1] MariaDB. http://mariadb.org/, 2012. [Online; last accessed 2012-09-20].

[2] MySQL. http://www.mysql.com/, 2012. [Online; last accessed 2012-09-20].

[3] MySQL Cluster. http://www.mysql.com/products/cluster/, 2012. [Online; last accessed 2012-09-20].

[4] Windows Azure SQL. http://www.windowsazure.com/en-us/home/features/data-management/, 2012. [Online; last accessed 2012-09-20].

[5] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[6] Cahill, M. J., Röhm, U., and Fekete, A. D. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 729–738.

[7] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. Making snapshot isolation serializable. *ACM Transactions on Database Systems 30*, 2 (June 2005), 492–528.

[8] Lehman, P. L., and Yao, s. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6* (December 1981), 650–670.

[9] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[10] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review 43* (January 2010), 92–105.

[11] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010).

[12] Pilman, M. Running a transactional Database on top of RamCloud. Master's thesis, ETH Zurich, 2012.

[13] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB '07, VLDB Endowment, pp. 1150–1160.