Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 32

Systems Group, Department of Computer Science, ETH Zurich

Running a transactional Database on top of RamCloud

by

Markus Pilman

Supervised by

Prof. Donald Kossmann
Simon Loesing

August 2011 - February 2012

inf | Informatik
Computer Science

Abstract

In today's big scale applications, the database system has more and more evolved to the most limiting bottleneck. Unlike application servers and web servers, database systems are hard to scale. During the last few years a new generation of simpler but more scalable storage systems, often refered to as NoSQL (Not only SQL), recieved more attention. While these system often solve scalability issues, they sacrifice some fundamantal properties of traditional database systems like consistency or the data model.

This thesis presents a relational and transactional database system which uses a key-value store instead of a hard drive disk as storage. This systems provides better elasticity and scalability than traditional disk based architecures without making any compromises in the consistency guarantees. In the first part we show that a transactional database can run on top of a key-value store without any performance or scalability penalties. We implemented such a system based on MySQL and RamCloud and provide benchmarking results of this system. In a next step we explain how this system can be made scalable and how the database system needs to be changed in order to be able to run several database instances on the same storage.

Contents

1 Introduction

1.1 Motivation

With the rise of the world wide web and cloud computing, today's database systems reach the border of their abilities. In large web applications database systems are often the bottleneck and companies like Google, Amazon, and Facebook put a lot of resources into the optimization and customization of their data storage technologies. One answer to the scalability problems of traditional database systems can be abstracted to the often used term NoSQL (Not only SQL). NoSQL databases often have the goal of providing better scalability, elasticity, and availability than traditional database systems. To achieve these goals, NoSQL databases makes big compromises, often one or several of these:

- *Transactions*: ACID style transactions are not always important for web applications. It is an extremely important feature for most business applications where non-serializable transactions can result in wrong results. But obviously transactions have their price, which is mostly achieved with the sacrifice of performance and scalability.
- *Consistency*: While consistency was always considered crucial for a database system, NoSQL database systems often lower the consistency guarantees. NoSQL Storage systems like Apache Cassandra[1] or Amazon SimpleDB[2] provide weaker forms of consistency, like eventual consistency. Eventual consistency only guarantees that an update will *eventually* be readable by every client.
- *Usability*: Most programmers today understand SQL and are able to use it. The relational model is powerful enough for most use-cases, formalized (with the relational algebra), and easy to use. The problem is that relational databases are difficult to scale. Therefore most NoSQL storage engines implement a much simpler (but easier to scale and more elastic) data model, like key-value pairs.

A big challenge in the research community is to conflate the benefits of a simple NoSQL storage engine (scalability, fault tolerance) with the benefits of relational databases (transactions, usability, consistency guarantees), where possible.

1.2 Problem statement

Most of today's transactional database systems use a hard drive disk as their underlying storage. This makes them unelastic, so migrating the database from one machine to another is difficult and often even requires a physical interaction with a machine. So, for example, if the machine has a defect, the disks must

[1] Apache Cassandra: http://cassandra.apache.org/
[2] Amazon SimpleDB: http://aws.amazon.com/simpledb/

be taken out and put into another machine. However, for cloud computing elasticity is a key feature for efficiency.

In a first step we want to substitute the storage layer of a traditional database system with a modern key-value store. To achieve this, the key-value store needs to provide strong consistency guarantees and atomic operations. One key-value store that provides these features and promises high-availability and excellent performance is RamCloud [12]. The relational database system we chose for this task is MySQL[3]. MySQL provides a plug-in system to change the underlying storage engine at run time. The first contribution of this master thesis is a fully functional storage engine for MySQL that stores and retrieves all data in a key-value store. The storage engine itself is plugable, so the used key-value store can be changed by implementing a very simple interface. We provide two plugins for this storage engine: one for RamCloud and one for Redis[4] - a widely used in-memory key-value store.

A transactional database on top of a distributed key-value store has a lot of benefits, one of them being elasticity. But it does not solve one of the main problems: scalability. So in a next step we want to be able to run several MySQL instances on several machines which all use the same data. This thesis discusses how this can be achieved and we've done a few experiments on a benchmarking system developed at ETH. The main contribution of this Master Thesis in this area is a new algorithm to do distributed locking which uses the key-value store itself to store all information needed for the algorithm.

We also present a detailed experimental evaluation of the first single node architecture.

1.3 Structure of the Thesis

Section 2 reiterates the design and implementation of transactional databases. The next section explains RamCloud, the used key-value and also explains why it is interesting for our work. In section four we briefly discuss three considered architectures that could be used to execute transactional workloads on top of RamCloud and the adjacent section explains how two of these architectures have been implemented. Section 6 presents the benchmark setup and results. The last section discusses future work.

[3]MySQL: http://www.mysql.com/
[4]Redis: http://redis.io/

2 Background: Relational Databases

This chapter gives an introduction on how relational databases are typically implemented. It only tries to cover the main ideas and is not meant as an introduction to the design and implementation of database systems. For a more complete coverage of this very large topic, the reader is referred to one of the many books (for example [5]) about the architecture of database systems.

2.1 Data Storage

Nearly all relational databases used today store their data on a hard disk drive. The data is stored in blocks, usually of the size of 4 kilobytes.

In SQL, most data types have a fixed length (exceptions are the *TEXT* and the *BLOB* type). This allows a row to have a fixed length that can be precomputed when the database schema is known (TEXT and BLOB fields are often handled separately, one example how it can be done as explained later in the chapter about implementation). If a row is smaller than the block size, several rows are stored in one block. A row can then be addressed by a block number and an offset (for example: third row in block $\#n$).

The bottleneck of a hard drive disk is not its speed but its seek time. A disk provides random access, but random access is expensive since the arms need to readjust and some time is needed until the magnetic disk has rotated to the right position.

Therefore database implementers usually put a lot of effort into minimizing random access to the storage and maximizing sequential read and write operations. So it is more important to reduce the number of disk accesses than to reduce the amount of data that is written or read with every disk access.

There are several techniques used to keep the number of disk accesses minimal. Some of them are:

- Trying to keep as much data as possible in DRAM and periodically flushing large amounts of data sequentially to disk.
- For each read access, read more data than needed and cache it, hoping that the data read will be needed soon.
- Trying to achieve a good data locality: if some data is needed, the possibility that data saved in a neighboring block will be needed later should be high - which will increase the effect of caching.
- Using data compression to be able to read more data in main memory.
- On a higher level, the query processor has to try to optimize queries in a way that reduces data access. A simple example how to achieve this would be to always execute a selection before an order by (if possible).

2.2 Indexing

Looking up data fast is important. If the query processor can not use an index for a table, it needs to perform a full table scan. This might be good if nearly all rows need to be inspected anyway (for example in a data warehouse). But if only a few rows are needed, a full table scan is not only slow, it also introduces a lot of locking. However, indexes have one big drawback: they slow down the performance for updating SQL queries. For each update or insert, all indexes need to be changed. So adding the right indexes is a non-trivial problem and is therefore mostly done manually by the user.

An index is a data structure which saves a key and a value. The key is the indexed row and the value is the row itself (or to be more precise: a pointer to the row, so a block number and an offset). Several data structures can be used to implement indexing in a database system. Usually a B-Tree [2] or one of its variants is used (often a B+-Tree, a B-Tree where all values are stored in the nodes). Reasons why B-Tress are so popular as index structures for database systems include:

- *Performance*: The run time of each operation (find, insert, delete) on a B-Tree is $O(log(n))$. While this holds also for other data structures like AVL-Tress [1], B-Trees perform notably well on block storage's, since every node can have the size of one block and therefore the number of disk accesses is reduced.

- *Concurrency Control*: If locking is implemented correctly, several threads can access one B-Tree concurrently. This can be achieved, for example, with a variant called B-Link-Tree, presented in [11].

- *Range queries*: Unlike hash maps, B-Trees allow the execution of range queries.

Most database systems implement more than one index type. For indexes which do not need range queries, hash maps are often more efficient. While B-Trees can be used for multi-dimensional indexes (with the keys just concatenated), R-Trees ([7] and [8]) are often a better choice there.

2.3 Transaction processing

A database system is often the most critical part of a company's IT infrastructure. The worst thing that can happen to a database is that it loses data or holds incorrect/corrupted data. If a database crashes, it is crucial that it can be brought up back quickly, crashes happen very rarely, and a crash must not result in any data loss or data corruption.

To ensure consistency and data safety, a database system implements a transaction manager. Most databases have a configurable isolation level, the highest one is often called serializable. Serializable means that all transactions running concurrently on one database read the same rows and write the same updates

as if they would have been executed one after another. Intuitively speaking this means, that from a users point of view, every transaction behaves exactly the same as if it would have exclusive access to the database system. If a conflict of any kind occurs, the database system rolls back the transaction and leaves the database in a state as if the transaction never would have happened (all or nothing semantic) - the client can then try to execute the transaction again or it can report an error message to the user. The part of the database system which conducts transactions is called the transaction manager and the line of actions a database system executes during transactions is called transaction processing.

There are several ways how transaction processing can be implemented. The most simple way is to just execute all transactions sequentially. This obviously guarantees that the transactions are serializable. The obvious drawback is the lack of parallelism, so if one transaction takes a very long time to run (which is usually the case in OLAP systems), all other incoming transactions have to wait until the transaction is committed or aborted. Therefore most database systems implement a more complicated approach which allows them to execute transactions in parallel. The most often implemented approaches are explained in [14].

A pessimistic approach for transaction processing is locking as described in [4]. The main idea is the following: Every transaction obtains read locks and write locks as needed. A read lock prevents a row from being written, while a write lock has the semantics of an exclusive lock. Every transaction acquires locks and the transaction manager keeps track of all locks held by each transaction. When a transaction tries to acquire a lock, it checks all transactions that keep the lock for whether one of them waits for a lock that is held by this transaction. If that is the case, a deadlock accrued and the transaction aborts, rolling back and releasing all held locks.

Instead of locking, some database systems implement a more optimistic approach (also explained in [14]). The main idea here is to "hope" that no conflict will occur during the execution of a transaction. If a conflict occurs, the transaction will roll back when the user tries to commit. Often, optimistic approaches can lead to higher performance than pessimistic locking. The two methods are compared in [3].

2.4 Distributed and Parallel Databases

In large applications, the database system becomes a bottleneck. While web servers and application servers are easy to distribute over several computers, scaling a database system is a non-trivial problem and still the topic of a lot of current research. There are mainly two ways of scaling out a database system: replication and sharding. Replication means that a database is replicated across several nodes. This increases read performance nearly linearly while it reduces write performance. When writing, the nodes have to agree on a commit protocol (some of them are explained in [14]) and each write needs to be executed on each node. Another benefit of replication is fault-tolerance. Sharding increases the performance of read operations less dramatically than replication, but writing is

faster since each node only has to execute parts of the write operations. But in a shareded system, the nodes still rely on a commit protocol which often forms a bottleneck of distributed database systems.

Another way of distributing databases is presented in this thesis: shared disk. The main idea is to provide a central storage and all processing nodes access the same storage.

3 A small overview of RamCloud

3.1 Motivation

RamCloud [12] is a scalable high-performance key value store. RamCloud provides low latency and high-performance by keeping all data in main memory.

The RamCloud store provides a very simple interface to store and retrieve data. In the following sections, we present the important properties of RamCloud (important in this context means: important for our use-case).

3.2 Features

3.2.1 API

RamCloud is a key value store, where the keys are unsigned 64 bit integers and the values are arbitrary byte arrays. In RamCloud, data is organized in tables. A client can create tables, write entries, read entries, delete entries and tables. Furthermore the API provides a method for test and set: A client can read an entry and gets its version (an unsigned 64 bit integer). When the client writes back the value, it can instruct RamCloud to only do so if the version did not change. All RamCloud requests are done atomically.

The API of the RamCloud client is described in Listing 1.

Listing 1: A simplified version of the RamCloud API

```
/**
 * Used in conditional operations to specify conditions under
 * which an operation should be aborted with an error.
 *
 * RejectRules are typically used to ensure consistency of updates;
 * for example, we might want to update a value but only if it hasn't
 * changed since the last time we read it.  If a RejectRules object
 * is passed to an operation, the operation will be aborted if any
 * of the following conditions are satisfied:
 * - doesntExist is nonzero and the object does not exist
 * - exists is nonzero and the object does exist
 * - versionLeGiven is nonzero and the object exists with a version
 *   less than or equal to givenVersion.
 * - versionNeGiven is nonzero and the object exists with a version
 *   different from givenVersion.
 */
struct RejectRules {
    uint64_t  givenVersion;
    uint8_t   doesntExist;
    uint8_t   exists;
    uint8_t   versionLeGiven;
    uint8_t   versionNeGiven;
};

/**
 * This is a simplified version of the RamCloud API.
 * This class in the RAMCloud namespace of the client
```

```
 * library provides all methods needed to access a
 * RamCloud.
 */
class RamCloud {
public:
  /**
   * Creates a new table or throws an exception if the table
   * already exists.
   */
  void createTable(const char* name);
  /**
   * Drops an existing table and deletes all its entries.
   * Throws an exception if the table does not exist.
   */
  void dropTable(const char* name);
  /**
   * Each table has an associated 32 bit unsigned integer.
   * The openTable method asks for this integer and returns
   * it to the client.
   * All access to a table has to be done with this integer.
   */
  uint32_t openTable(const char *name);
  /**
   * Reads the value of the given table id and id into value.
   * Optionally the current version number can be retrieved.
   */
  void read(uint32_t tableId, uint64_t id, void* value,
            size_t length, uint64_t* version = NULL);
  /**
   * Removes a key value pair from the table with the table id
   * tableId.
   */
  void remove(uint32_t tableId, uint64_t id,
              const RejectRules* rejectRules = NULL,
              uint64_t* version = NULL);
  /**
   * Writes the value of the given table id and id into value.
   * Optinally a RejectRules object can be given, to prevent
   * overwriting new values, values which do exists etc (see
   * the declaration of the RejectRules class above).
   */
  void write(uint32_t tableId, uint64_t id, const void* buf,
             uint32_t length, const RejectRules* rejectRules = NULL,
             uint64_t* version = NULL, bool async = false);
};
```

3.2.2 Consistency

RamCloud guarantees full atomicity for all its basic commands. Furthermore it allows the user to implement a test and set feature: each entry in the store has an associated version number. A user can decide to write a value only in case the version did not change since the last read.

Whenever a value is written, the next read will return the newly written value (unlike in a database that only provides eventual consistency).

	RamCloud	Disks
Cost	High	Low
Power Consumption	High	Low
Latency	Low Infiniband: $5 - 10\mu s$ Ethernet: $\sim 150\mu s$	High ($\sim 10ms$)
Fault-Tolerance	Backup nodes	RAID
Down-Time in case of node failure	short (few seconds)	long (requires manual intervention)

Table 1: Comparison of RamCloud vs Disks

3.2.3 Availability

RamCloud uses sharding to distribute the data over several nodes. The data is also stored in other backup nodes (in an asynchronous written journal file onto disk). So if a node goes down, this part of the data is not available anymore.

Therefore RamCloud is not does not guarantee total availability. But RamCloud is designed in a way, that recovery of a node should only take a few seconds (see [12] for additional information on how this is achieved). So one can say that RamCloud is *nearly* highly-available by guaranteeing a very short downtime.

3.3 RamCloud vs Disks

Traditional transactional databases use hard drives as storage medium. Using RamCloud instead of disks has several advantages and disadvantages, some of them are listed in table 1.

The main motivation for using main memory data stores instead of disks is the low latency of main memory access. In case of RamCloud, the latency is mostly the same as the network latency. RamCloud is optimized to run in an Infiniband network, which provides much lower latency than Ethernet, but it can also run over Ethernet. Another big advantage is the behavior of such a system in case of failure: if a disk breaks, the system won't go down (but the performance of the system will be significantly lower), but a human must replace the hard drive. If the whole node goes down (for example because of a power supply failure), a new node must be brought up, and the hard drives from the broken system need to be assembled into the new one. This results in a high downtime in case of failure. If the data is stored in RamCloud, failure detection and recovery could be done automatically which will result in very low downtimes.

But of course there is a price to pay - literally. While the cost of disk space is very low, DRAM is much more expensive (the cost per GB is usually about a factor 100 higher). Also the power consumption is much higher. In addition, several computers will be needed to provide the same capacity as a disk. While

a RAID with several terabytes of capacity is easy to build and inexpensive, one node will usually have not more than a few gigabytes of capacity. This leads to additional cost for computers and networking infrastructure and it will increase the power consumption of the whole system. The authors of RamCloud argue, that the disadvantages of DRAM are compensated by using less caching in the system.

4 Alternative Database Architectures for RamCloud

4.1 Traditional - Replace Disk with RamCloud

One possibility to run a transactional database on top of RamCloud is, to just replace the disk with the key-value store. One big advantage of this architecture is that it is well understood. We can use the same indexes and caching algorithms that a traditional database uses. Since a network has a lower latency than a disk, the performance should be at least as good as with a disk.

Using a key-value store instead of a disk for a database has several advantages. One of them is that the key-value store already brings data safety and high availability of the storage. So if we compare this to a traditional setup with a disk RAID, where a crash of the processing node (for example a broken power supply unit) needs manual removal of the disks and placing them into a new computer, on a key-value store a broken database server can just replaced by another one by starting a new instance of the database program. This could even be done automatically, so better availability could be provided with hot stand-by.

On the other hand, this architecture shares a lot of the downsides of traditional database systems. Probably the biggest drawback is, that the architecture scales poorly. It can be scaled by doing sharding at the application level, but this puts the burden on the application level developers.

Nonetheless a database with a key-value store has a lot of use-cases. In a cloud computing environment, where a lot of small applications run in one huge network, a database system like the one described here looks like the perfect fit. Each database instance would serve one application and all would use one large key-value store. So moving the database node from one machine to another one would be trivial.

4.2 Distributed processing with shared disk

As discussed before, the approach of traditional databases on a key-value store is great for a lot of use-cases. Although it has the flaw that scale out gets difficult. Since we are now using a distributed key-value store, it is just logical to think that several nodes in a network could access the same key-value store to execute queries.

The problem is that this won't work out of the box. Several assumptions which hold for a traditional database do not hold anymore on such a distributed environment. One example is caching: while a write through cache will always be up to date on a traditional database, it won't be in a distributed one, since at every point in time another processing node could have written new data to the storage. Another problem is that the implementation of transactions does not work the same way as on transactional databases.

4.3 Other execution models

Besides traditional databases, one could also consider to look into different execution models. One possible alternative execution model could be the one used by Crescando [6]. But no alternative is covered by this master thesis, so evaluating different execution models on RamCloud could be a topic of future work.

5 Implementation

5.1 Traditional - MySQL on RamCloud

5.1.1 Motivation

MySQL is one of the most widely used databases. It is an open source relational database with support for ACID transactions.

One unique feature of MySQL is the concept of storage engine. A storage engine (described in [13]) is a MySQL plugin that manages the storing and retrieving of data, handles indexes etc. MySQL already comes with several storage engines. Some of them are:

- InnoDB - A high performance storage engine with support for transactions.
- MyISAM - The oldest storage engine, which has support for full text search but does not support transactions but only table locking.
- Tina - stores the data in regular CVS files, does not support indexes.
- HEAP - an in-memory store, it loses all data after a reboot.

There are also companies and organizations who developed their proprietary storage engines which suit some special use-cases.

So a big part of this master's thesis was to implement a storage engine for MySQL, which stores all data in RamCloud instead of disk. Having the data in RamCloud brings several benefits to the user. A reasonable setup could store all meta data (like table information) on a network file system and use the RamCloud storage engine. The RamCloud will then take care of durability and availability. If the MySQL node goes down, the only thing the user (or some kind of watchdog process) has to do is to start a new MySQL instance with the same configuration and the same meta data on another node. One could also imagine to build support for several MySQL instances using the same storage for scale out (although there are still some problems, because MySQL assumes that it has explicit access to its storage - things like result caching and auto increment won't work correctly anymore).

But the main motivation to build a storage engine on top of RamCloud was to answer the question whether RamCloud will be able to suit as a storage for a traditional transactional database.

5.1.2 Architecture

An overview of the architecture can be seen in Figure 1. The clients still communicate with MySQL in the same way they usually do (for example with JDBC, MySQL C driver, ODBC etc). The only difference is that the user will either tell MySQL to use the RamCloud store (called *kvstore*, since the store also allows the usage of other key value stores than RamCloud) when she creates a table

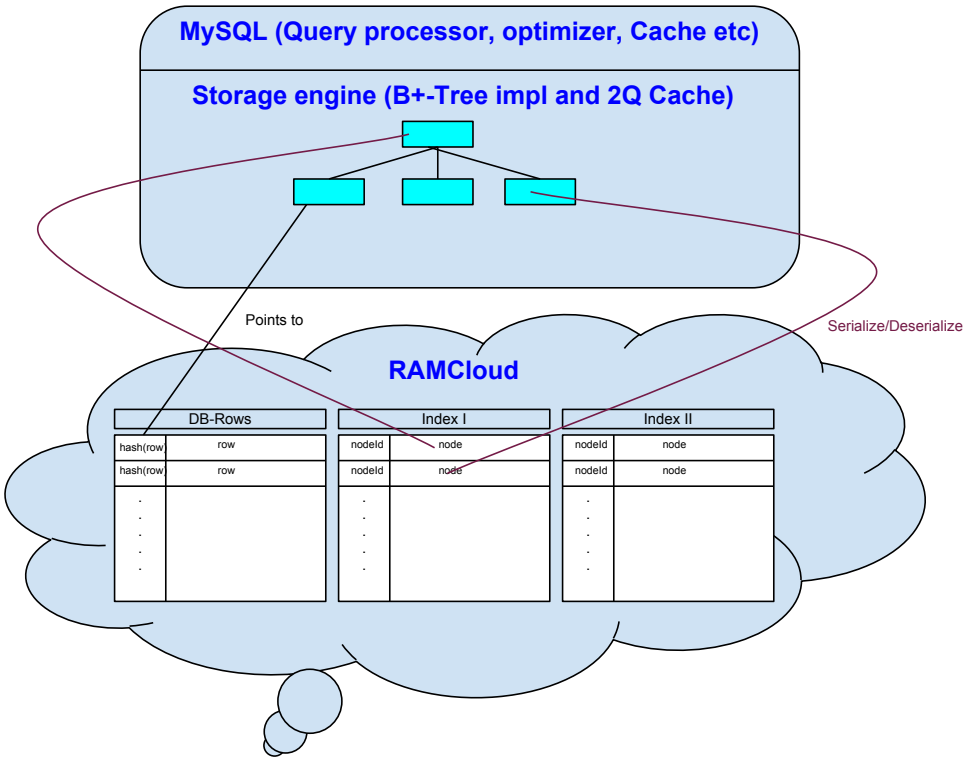

Figure 1: Overview of the architecture

(alternatively, the administrator could define kvstore as the default storage engine in MySQL's configuration). A create statement that creates a table "foo" with a primary key "i" of type *int* and a *varchar(255)* named "bar" would look like presented in Listing 2.

Note that at the first start-up (i.e. once after installation), the administrator has to tell MySQL to install the kvstore plugin. This can be done with the command in Listing 3 on the MySQL shell. This has to be done only once - at the next start up, MySQL will load all plugins installed before.

Listing 2: A simple create statement which tells MySQL to use the kvstore

```
CREATE TABLE foo (i int primary key, bar varchar(255)) engine = kvstore;
```

Listing 3: Command to load the plugin

```
INSTALL PLUGIN kvstore soname 'libkvstore.so';
```

The kvstore engine implements indexes with a B+-Tree, locking and caching. MySQL also implements locking, but the storage engine downgrades all locks to allow several concurrent write and read requests.

The storage engine implements the following functionality:

- Iteration over a table (full table scan).
- Index look up.
- Iteration over all rows of a given indexed value.
- Handling of BLOBs and TEXT (strings with arbritrary length) entries.
- Row-level locking.
- Updating rows.
- Auto increment.
- Unique checks on indexes.
- Approximation of the table size (needed to support large tables - otherwise MySQL runs out of space for certain queries).

The features which are not implemented yet are:

- ACID transactions.
- Index over several columns.
- Full text (the only storage engine in MySQL which provides this feature is MyISAM).
- Tables without a primary key (this is because RamCloud does not allow full table scans - but implementing this feature would be straight forward by using a generated primary key).

The rest of this section explains how the storage engine implements these features

5.1.3 Saving database rows in RamCloud

MySQL passes a character array to the storage engine to store it. The storage engine will create a unique 64 bit unsigned integer which is associated with this row (using the primary key) and save it in a RamCloud table. A special case are BLOB and TEXT fields, those are not part of the row. So for each BLOB and TEXT field, a new RamCloud table is created (during table creation) and the data is saved there.

Primary key	
id	*value*
1	5
2	6
3	node{keys[N] values[N+1] ...}
4	node{keys[N] values[N+1] ...}
5	node{keys[N] values[N+1] ...}
6	node{keys[N] values[N+1] ...}

Rows	
id	*value*
975	byte[]
124876	byte[]
234983	byte[]
249776	byte[]
398762	byte[]
399127	byte[]
987641	byte[]
...	

BLOBs	
id	*value*
975	byte[]
124876	byte[]
234983	byte[]
249776	byte[]
398762	byte[]
399127	byte[]
987641	byte[]
...	

Table 2: An example of how a MySQL table with one index and one block entry is stored in RamCloud

5.1.4 B+-Tree

B+-Trees are essential in the MySQL storage engine. They are not only used for index look-up, but also for full table scans (a full table scan is basically an iteration over all leaves in the B+-Tree for the primary key). The engine implements a B-Link-Tree as described in [11].

Each B+-Tree is stored in its own RamCloud table and each node get its own id (an unsigned 64 bit integer). The node stores a boolean variable indicating whether the node is a leaf node or not. It further maintains two arrays: one array with all keys for that node and one with all values. A value is either a node id (a link to another node) or the id of a database row. The first entry in the RamCloud table just stores the id of the root node, the next stores the last node id which was assigned to a node (this is used to make sure that all node ids are unique). The node id 0 is always invalid.

Summing everything up, for each MySQL table, the following tables in RamCloud are created:

1. A table for the data (the rows).
2. For each BLOB and each TEXT column one table.
3. For each index one table to store the corresponding B+-Tree.

An example of the low level structure for a simple table is in Table 2.

5.1.5 Caching

Caching in this architecture is essential. RamCloud look ups are significantly more expensive than just a cheap memory look-up. Since the storage engine runs under the assumption that there is only one MySQL instance per database, everything is cached in a write through manner. The kvstore storage engine implements a 2Q cache as described in [9].

The cache is placed in front of the key value store - so before each request to the store, the client first looks up the key in the cache. If the key is found in the cache, no request gets sent to the store. Otherwise a request will be sent and the result is stored in the first queue of the 2Q cache. For write operations the semantics of a write through cache is implemented: first, the value gets written into the cache (if the value was not stored into the first queue before, into the second otherwise) and then the write request is sent to the RamCloud.

Since we are assuming only one processing node, this behavior is correct. One problem with this approach is locking: since so many threads try to access the cache at the same time, the threads wait most of the time for a lock to access the cache. To solve this problem a very simple solution is chosen: instead of one big cache, the storage engine conducts several (by default 100) caches. For each key it just computes $key \bmod n$ (where n is the number of caches) and than looks up the appropriate cache. The drawback of this solution is of course that the available space for caching will not be used as efficiently as with just one cache.

5.2 A distributed architecture

The logical next step is to port this architecture so that we can execute it in a distributed way. That means we want to run several MySQL instance on one database stored in the same RamCloud store.

To achieve this, finding the best way to run transactional workloads on several nodes which all access the same storage is crucial. Running MySQL in a distributed way is out of scope for this Master thesis, since it includes a lot of challenges, which are out of scope of this thesis because they are very specific to MySQL (like how to make sure that auto increment will still work, how to disable result caching etc).

The Systems Group at ETH is already doing experiments with the shared architecture. So the architecture presented here and the work presented in this thesis are based on this work.

5.2.1 Architecture

The general idea of what we want to achieve is illustrated in Figure 2. So instead of a shared nothing architecture, where the data is sharded over several nodes, all nodes share one storage with all data. To coordinate the requests to the node and provide some load balancing, we have a master node, that knows about all active processing nodes. So a client will first connect to the master node, asking for a list of active processing nodes and will then start a transaction on one of them.

One problem on a distributed architecture like this is caching. Since every node could execute an update at every point in time, the cache on a node is not guaranteed to be up to date anymore. A simple solution to this problem would be to just disable caching. But this would obviously decrease the performance of a node dramatically and increase the load on the key-value store in a way

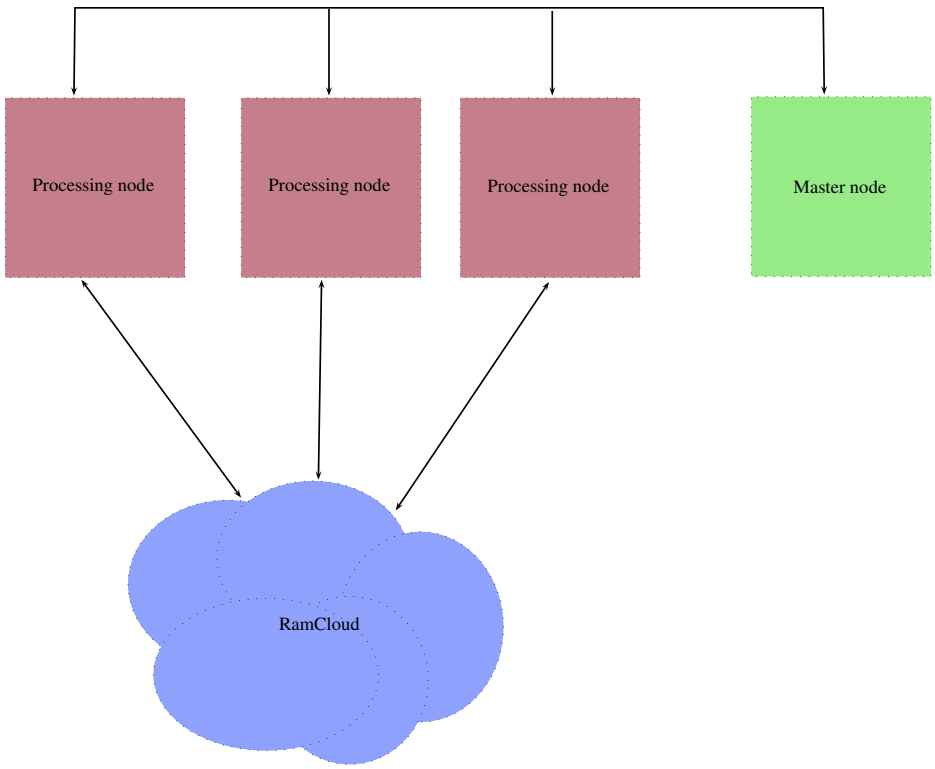

Figure 2: Overview of the architecture

that is not acceptable. So to decrease accesses to the key-value store, we need to provide some minimal caching. Our B-Link-Tree [11] has the property that it does not make any assumptions about changes made concurrently if we are only accessing it for reading. So even if we do not have the newest version of the nodes, we can still find all keys saved in the B-Tree, if we do not cache the leaf nodes.

For database rows the story is slightly different. If a transaction on a node has a lock on a row, we know that the row won't change until the transaction releases the lock (i.e. commits or aborts the transaction). Therefore we can cache a row as long as at least one transaction running on the same node has a lock on the corresponding row and we have to evict the row from the cache as soon as the last transaction holding a lock on the row releases its lock. This can easily be implemented using the same mechanisms as in reference counting - acquiring a lock increases the reference count and gets the row from the key-value store if the reference count was zero before, releasing a lock decreases the reference count and evicts the row from the cache if the reference count went to zero.

5.2.2 Distributed Locking

In a distributed environment locking gets more difficult. On a single node configuration, all locks are local and locking can be implemented simply by using the locking mechanisms provided by the operating system. In a distributed environment this does not work anymore, since acquiring a lock on one node does not have an effect on any other node. This thesis proposes three variants how one could implement distributed locking by using a key-value store.

The first variant uses a slightly adapted version of the algorithm described in [15]. For every column we save three integers used for locking in the key-value store: an event count called *evtcnt*, a sequence number called *seq* and the number of current read locks called *rlocks*. We assume that we can increment and decrement these numbers in an atomic way - some key-value stores provide features for that, on RamCloud this can be easily implemented on the client side (using RamCloud's versioning feature). The *seq* and *rlocks* are initialized to 0, the *evtcnt* is initialized to 1. When a transaction wants to acquire a read lock, it will first increment the *seq* number and memorize the new *seq* as *my_seq* and wait until the *evtcnt* is equal *my_seq*. When they are equal it will simply increment the *rlocks* number. A write lock will also first increment the *seq* and wait until *evtcnt* is equal its *my_seq*. At this moment the transaction does not yet have the write lock, since it still could be that other transactions have a read lock. So it also need to wait until *rlocks* goes to 0. Unlocking is very simple: unlocking a read lock means to only decrement the corresponding *rlock*, to unlock a write lock one needs just to increment the corresponding *evtcnt*. This locking variant is illustrated in pseudo code in Listing 4.

Listing 4: Pseudo code how to implement variant 1

```
void rlock(int id) {
  int my_seq = ++seq[id];
  wait_for(my_seq == evtcnt[id]);
  ++rlocks[id];
  --evtcnt[id];
}

void wlock(int id) {
  int my_seq = ++seq[id];
  wait_for(my_seq == evtcnt[id]);
  wait_for(rlocks[id] == 0);
}

void unlock_rlock(int id) {
  --rlocks[id];
}

void unlock_wlock(int id) {
  --evtcnt[id];
}
```

One challenge is to implement the waiting. In variant 1 we just assume that all transactions periodically pull the values from the key-value store to

check whether they can acquire the lock. In another solution we call variant 2, we introduce the possibility that nodes can communicate with each other. So every node would open a socket where it would listen for packages informing a transaction that an eventcount reached the value it is waiting for. Therefore a value would need to be stored in the key-value store for each waiting transaction, so that another node would know how to inform a waiting transaction. So this would increase the bookkeeping. But beside the added complexity to the system, it would also slightly change the architecture to what is illustrated in Figure 3.

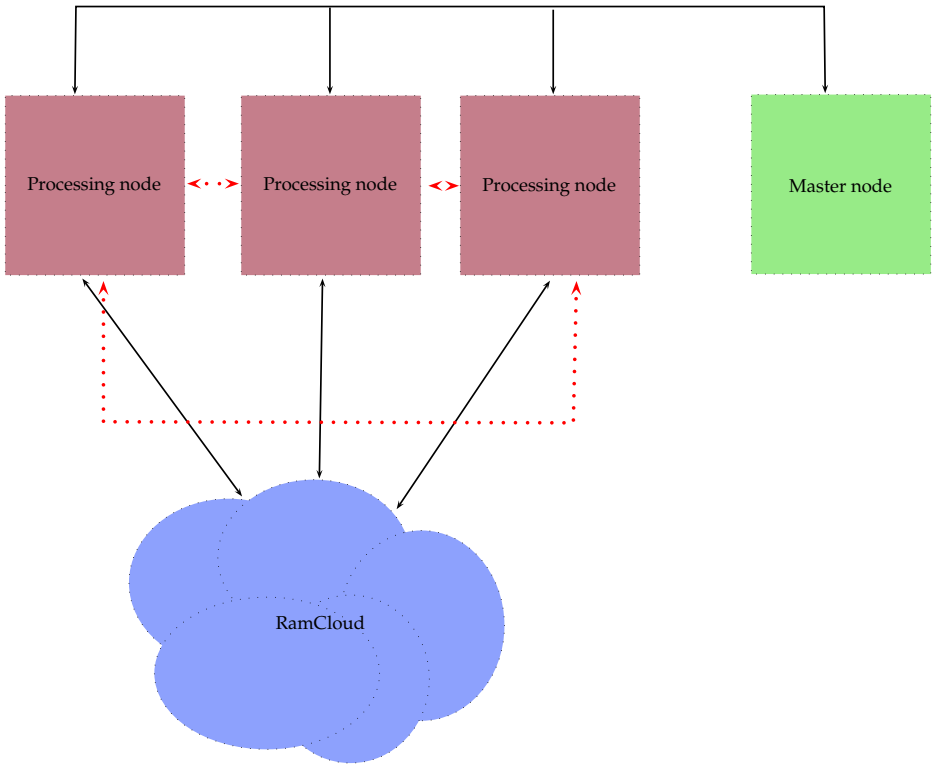

Figure 3: Overview of the architecture with inter-node communication

A third possibility (our variant 3) would be to just use a central locking service that would write back its data synchronously to the key-value store. This service could easily inform transactions when they got the lock and such an architecture would significantly decrease accesses to the key-value store. Also the locking service could use sharding to scale out.

An open problem with our locking proposals are deadlocks. Detecting a deadlock in a distributed environment is possible but expensive. Therefore we do not detect deadlocks but use time outs instead - so if a deadlock occurs, a transaction will eventually time out. The problem with timeouts is that if one transactions holds a lot of locks and takes a long time to run, some transaction

would probably time out, even if no deadlock occured. Therefore finding a good value for a time out is difficult and even depends on the work load: on an OLTP workload, a small time out would be the better choice, since queries usually don't run very long and having deadlocked transactions will prevent other transactions to continue. On an OLAP workload on the other hand, short time outs will make a lot of non-deadlocked transactions abort which will decrease the throughput of the system. Therefore the time out should be either a configurable value (giving the liability to the user) or the nodes could measure the query times and try to agree on a time out based on their data.

Variant 1 is implemented, the benchmarking of the three variants are future work.

6 Experiments and Results

As part of this thesis the MySQL storage was completely implemented. The distributed architecture described in Chapter 5 was implemented, but no experiments were run against it. In this chapter the benchmark methodology is explained and the results are presented.

6.1 Benchmark Methodology

A detailed description of the benchmark methodology can be found in [10]. To test the performance of the MySQL database on RamCloud we run the TPC-W benchmark with the update intensive ordering mix against it. The TPC-W benchmark emulates an online bookstore with a mix of fourteen different requests, such as placing an order, browsing, and shopping. A request is not a SQL instruction but a request from a browser to an application server. The applicatoin server will then start a new transaction on the database and execute a series of queries for each request. In our experiments we are interested in the throughput and we measure the throughput in Web Interactions per Seconds (called WIPS from here on). An overview of the setup is illustrated in Figure 4.

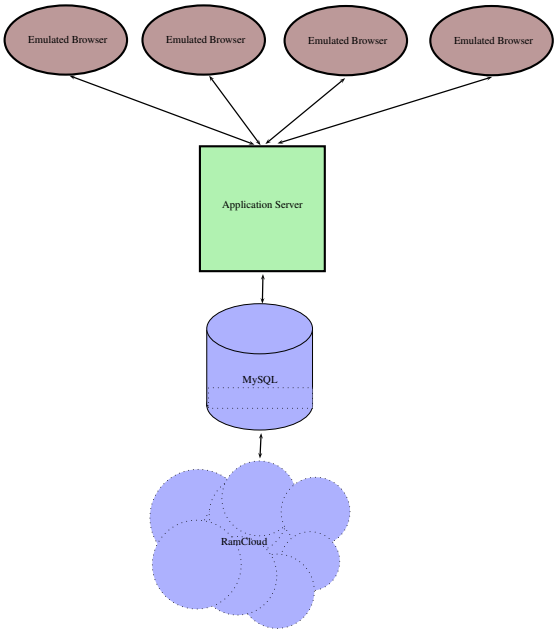

Figure 4: The benchmark setup

Each emulated browser (EB) sends a request to the application server and waits a few seconds for the response. If the response arrives after the set response

time, the response is ignored (modelling impatient users). When a response arrives, the EB waits for a few seconds (simulating a users "thinking time") and then sends another request. The emulated browsers are simulated using threads on the benchmarking system. Of course this removes the network latency which would be there in a real setup, but since our system under test is the database, this setup is good enough. As a baseline we first executed the benchmark using MySQL with InnoDB. For the tests we have the following machines running:

- One machine which runs all emulated browsers and the application server. We could easily run this on several machines, but since one machine is sufficient to generate enough load, this setup is simpler.
- One machine which runs the MySQL server.
- Four machines which run RamCloud: one machine runs the coordinator and thre machines run a RamCloud server. Of course these three machines are not used for testing MySQL with InnoDB.

The testing server will start a new EB every 200 milliseconds, increasing the load linearly. We are interested in the following numbers:

1. The peak of the throughput (measured in WIPS in response time). After the peak is reached, the throughput will break down, since not all requests will be answered in the required response time.
2. The load to the RamCloud. From micro-benchmarks not presented here, we know that RamCloud can reach a throughput of 9000 requests per second, afterwards the network becomes a bottleneck. We want to know whether MySQL can reach this number. As explained in the implementation part of this thesis, we are maintaining a queue of all requests waiting to be handled by the RamCloud thread. When this queue starts to grow, we know that RamCloud starts to become the bottleneck.

So beside the throughput we are also interested in the answer to the question: is one MySQL instance able to saturate the RamCloud (or to saturate the network communication to the RamCloud). Since most of the data is cached, after some while mostly write requests will go to the RamCloud - one big advantage of the single node architecture. Therefore we do not expect a high load on RamCloud.

6.2 Experimental Results

As shown in Figure 5, MySQL with InnoDB reaches its peak when there are 2000 EBs running and it gets to a throughput of 270 WIPS in response time. MySQL on RamCloud 6 on the other hand reaches its peak when there are about 3000 EBs running getting to a throughput of 413 WIPS in response time.

When looking at these results one should also take into consideration that the RamCloud storage engine does not yet implement transaction but only

guarantees row level consistency. A transaction implementation will also slow down the queries, so that a fully transactional version of the RamCloud storage engine will have worse results than the ones presented here. But most probably the engine could be made at least as fast as InnoDB, since RamCloud is a long way from being saturated.

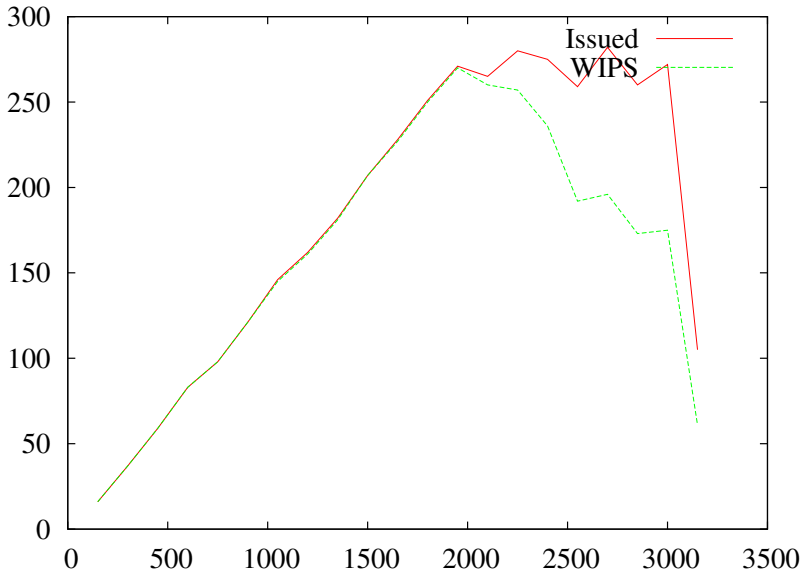

Figure 5: The results with MySQL and InnoDB

Not shown in the graph is the load MySQL produces on the RamCloud. At the peak the load reaches about 300 requests per second. Therefore we can also answer the second question: one MySQL instance can not produce enough load to saturate the RamCloud.

The important result here is that we can run MySQL on top of a key-value store without any loss of performance or scalability but still gain the advantages this architecture provides. We have the desired elasticity and several applications could run in the same network using each its own database system, but all on the same key-value store.

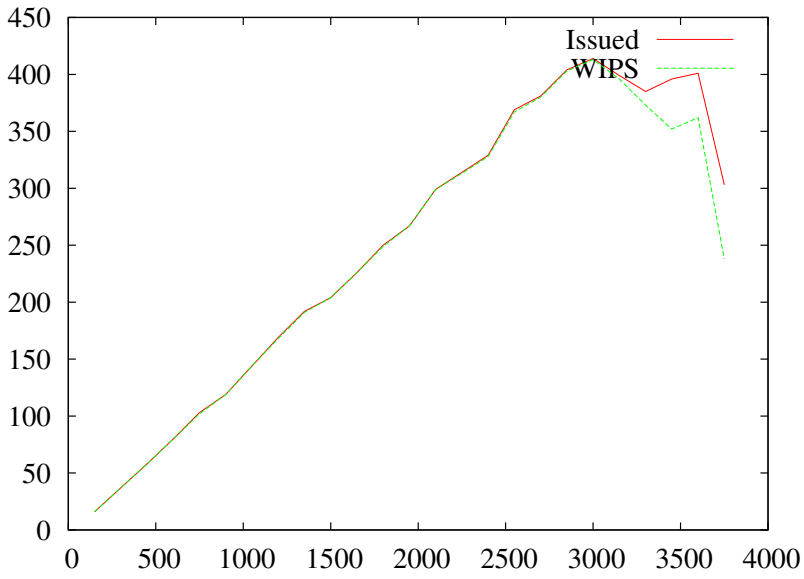

Figure 6: The results with MySQL on RamCloud

7 Future Work

Several questions arose during the work on this master thesis which are not yet answered. First of all, it is still unclear which locking strategy would be the most performant for the distributed architecture. For that purpose the proposed variants for distributed locking could be implemented and compared with each other. It could also be that a more optimistic approach, like snapshot isolation, would perform much better in our environment than pessimistic locking. An implementation of snapshot isolation and a comparison between optimistic and pessimistic approaches would be interesting.

RamCloud was designed to run on an InfiniBand[5]. Infiniband is not only faster than Ethernet, requests over an InfiniBand network provide a much lower latency. So the distributed architecture should also be tested in an InfiniBand network. This should scale further and eliminate the network as a bottleneck. Maybe we also will have to rethink caching, since it could be that caching will just introduce more complexity without providing any further benefits, since the network will be fast enough to handle all requests without doing any caching. Furthermore it is imaginable that different concurrency control mechanisms are preferable on different network architectures.

The last but most important step would be to bring these results together and implement the whole architecture. To do so, transactions need to be implemented in MySQL. Probably one will run into several problems when trying to run MySQL in such a distributed way. We expect following problems are likely to occur:

- By default MySQL caches query results and reads results directly from the cache if the same query was executed before. This will result in an incorrect behavior in a distributed environment, since another MySQL instance could have executed an update since the last execution of the cached query. This problem can be easily solved by deactivating this cache.

- MySQL stores all schema information in files and the storage engine does not have any control about this behavior. So having these files synchronized will be problematic. A simple solution would be to just shut down all but one MySQL instances when a query arrived that alters schema information, execute the query only on one node, synchronize the information over all nodes and bring all MySQL instances back up again. However this will make altering tables extremely expensive. So in the long run, it would be better to change the MySQL code, so that schema information can also be stored in the key-value store.

- MySQL does not provide sequences. For automatic key generation, MySQL provides an auto increment feature. The implementation of auto increment will always get the last entry in the table and increment its key

[5] http://www.infinibandta.org/

value by one. This could lead to problems since the query engine can not make sure that this incremented new key value will be globaly unique. Probably this won't result in problems with correctness, because the storage engine will just refuse to insert a new row if the key is not unique, but it will potentially result in less performance if this happens often. So the cleanest fix would be to implement seuqences in MySQL.

References

[1] Adelson-Velskii, G. E. M. L. *An algorithm for the organization of information.* 1962, pp. 263–266.

[2] Bayer, R., and McCreight, E. *Organization and maintenance of large ordered indexes.* Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 245–262.

[3] Daniel A. Menasc, T. N. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems 7* (June 1981), 13–27.

[4] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19* (November 1976), 624–633.

[5] Garcia-Molina, H., Ullman, J., and Widom, J. *Database systems: the complete book.* Pearson Prentice Hall, 2009.

[6] Giannikis, G., Unterbrunner, P., Meyer, J., Alonso, G., Fauser, D., and Kossmann, D. Crescando. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 1227–1230.

[7] Guttman, A. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec. 14* (June 1984), 47–57.

[8] Guttman, A. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), SIGMOD '84, ACM, pp. 47–57.

[9] Johnson, T., and Shasha, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), J. B. Bocca, M. Jarke, and C. Zaniolo, Eds., Morgan Kaufmann, pp. 439–450.

[10] Kossmann, D., Kraska, T., and Loesing, S. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 579–590.

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

[12] 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 Oper. Syst. Rev. 43* (January 2010), 92–105.

[13] Pachev, S. *Understanding MySQL Internals*. O'Reilly Media, Inc., 2007.

[14] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. ADDISON-WESLEY, 1987.

[15] Reed, D. P., and Kanodia, R. K. Synchronization with eventcounts and sequencers. *Commun. ACM 22* (February 1979), 115–123.