MoSQL: A Relational Database Using NoSQL Technology

Master's Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Alexander Tomic

under the supervision of
Fernando Pedone

September 2011

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Alexander Tomic
Lugano, 14 September 2011

Abstract

We discuss the architecture and construction of "MoSQL", a MySQL storage engine using the Tapioca key-value database as its row persistence layer and a distributed B+Tree built on top of Tapioca for indexing purposes.

Relational database systems (RDBMS) have long had a monolithic architecture that is difficult to scale while preserving serializable transaction semantics; MySQL as a RDBMS, despite its popularity, is still without a non-commercial, general-purpose storage engine providing out-of-the-box high availability and the ability to transparently add database nodes to scale database performance.

The MoSQL storage engine thus represents a fusion of MySQL, with its rich SQL interface, and a more modern NoSQL-type system such as Tapioca with its scalability and fault tolerance characteristics. We discuss the construction of the storage engine and assess its performance relative to two other popular MySQL storage engines: InnoDB and MyISAM. We show that MySQL, when used with the MoSQL storage engine, shows the ability to scale performance linearly in the number of database nodes to at least sixteen, greatly surpassing the performance of single-server installations of MySQL with InnoDB or MyISAM in certain workloads. In addition we evaluate the performance of MoSQL on the well-known TPC-B and TPC-C benchmarks.

Acknowledgements

I would like thank my advisor, Professor Fernando Pedone, for humouring my sometimes silly ideas and his always incisive analysis of what at times seemed like insurmountable problems. I would also like to thank Daniele Sciascia, with whom I worked very closely these past seven months, for his patience with my never-ending questions and problems with Tapioca, a tiny minority of which were actually legitimate issues. Without their patience, expertise and wisdom, this thesis would not have become a reality.

Contents

Figures

Tables

Chapter 1

Introduction

Relational database management systems (RDBMS) have held mindshare among application developers as the most appropriate way to store critical data for several decades. This has remained true despite many alternative approaches, such as XML-based storage and object databases, showing promise. In recent years the realities of scaling database systems in the Internet age have made customized solutions more practical than attempting to fit general-purpose RDBMSs to diverse application needs [SMA+07].

Despite the disadvantages of using a general-purpose RDBMS in comparison to more specific solutions as discussed in [SMA+07], we expect that a significant number of legacy applications will remain in the years ahead and thus the need to improve the scalability, performance and fault-tolerance of RDBMSs is acute.

MySQL is a popular open-source RDBMS used extensively in Internet-connected applications, with Facebook as an example of one of its largest users.[1] Despite its continued popularity, MySQL is still without a non-commercial, general-purpose storage engine providing out-of-the-box high availability and the ability to transparently add database nodes to scale database performance. The goal of the MoSQL project is thus to create a working MySQL storage engine that takes advantage of the properties of the Tapioca key-value database.[2] Tapioca provides near-linear scalability with additional nodes and strongly consistent transaction semantics. Tapioca incorporates many of the recommendations made by [SMA+07], such as in-memory data storage, simplified concurrency control and single-threaded execution. Thus, we hope with our project to bridge the current, now-ending "era" with the next era in database design. Given the time constraints, we have attempted to prototype a system that can perform well on simple, primary-key oriented workloads such as a TPC-B, while also enabling

[1] https://www.facebook.com/MySQLatFacebook

[2] We discuss Tapioca in further detail in Chapter 3.

support for the more complex types of queries required by a workload such as TPC-C.

1.1 Related Work

1.1.1 BerkeleyDB Storage Engine

BerkeleyDB (BDB) is a popular database originally developed by Sleepycat Software. It is an embedded key-value database with support for transactions and the ability to store arbitrary data for both keys and values - salient features also provided by Tapioca. Until version 5.0 of MySQL Server it was used as a basis for a storage engine, although it has been deprecated in later versions of the server.

The BDB storage engine maps relational MySQL data to a key-value format by storing a packed representation of the row as a BDB value; the primary key for the table is used as the BDB key. The storage engine has the following important characteristics, that, we will see later, are similar to those of our implementation:

- All MySQL row data in a BDB database is accessed via a B+Tree; row data is not stored separately from index data, so all sequential accesses are done through the B+Tree.
- Tables require a primary key: if one is not provided, an internal primary key is created in order to uniquely access records.
- Secondary indexes are supported, but are stored with the primary key and thus incur an additional storage cost proportional to the size of the primary key.

1.1.2 GenieDB Storage Engine

GenieDB is a commercial storage engine for MySQL developed by GenieDB Inc.[3] At the time of this writing, no peer-reviewed publication is available describing the storage engine or its back-end storage strategy, but some features and capabilities can be inferred from the white papers available from their commercial website.[4]

GenieDB appears to provide two levels of functionality. The lower-level GenieDB datastore is described as a distributed database providing immediate consistency across a number of nodes. Replication is made more efficient, where

[3] http://www.geniedb.com/

[4] http://www.geniedb.com/wp-content/uploads/2011/04/sql-overview-revised.pdf

possible, through use of a reliable broadcast protocol. The datastore is accessed through a type of "NoSQL API" that, we assume, is similar to APIs for most key-value systems.

The GenieDB MySQL storage engine is then built on top of the GenieDB datastore to provide relational access, implementing the MySQL table handler.

1.1.3 Building a Database on S3

Amazon Simple Storage Service[5] (S3) is among the first utility computing services offered by Amazon, providing users "buckets" of highly available and scalable storage. In [BFG+08] we see the novel idea of building a transaction-processing system using S3 as its storage system. The cost-per-performance results were shown to be too high to justify its use for high-performance transaction processing, although it was shown to be a viable approach for many classes of Web 2.0 and interactive applications.

1.1.4 NoSQL

The term "NoSQL", while dating as far back as 1999 [Pat99], has recently come to be associated with a loose collection of databases after a 2009 conference[6] meant to discuss the growing number of database systems that discarded the constraints and semantics of relational databases and SQL interfaces. Non-relational systems such as key-value stores, document, object and graph databases typically fall within the scope of this term.

1.2 Thesis Outline

Chapter 2 provides background on the MySQL relational database system and its salient features. Chapter 3 discusses the Tapioca key-value database and how its properties lend itself well to use as a back-end for MySQL. Chapter 4 discusses in detail the design and implementation of the MoSQL storage engine. Chapter 5 presents performance data for the MoSQL storage engine and comparisons to the performance of the standard single-server MyISAM and InnoDB storage engines. Chapter 6 discusses our conclusions and areas for future work.

[5] http://aws.amazon.com/s3/
[6] https://nosqleast.com/2009/

Chapter 2

MySQL

2.1 Relational Database Architectures

RDBMSs trace their theoretical foundations to the work of Codd [Cod70], while much of the work of implementing real-world, useable systems has fallen upon the traditional relational database vendors such as IBM and Oracle. The physical architecture of a typical RDBMS installation has remained single-server focused into the 21st century, with higher transaction throughput requirements typically requiring more expensive hardware. Shared-disk and shared-nothing architectures[Sto86] have become more popular in recent years. For applications that are able to function with relaxed data synchrony requirements, the case for many high-volume web-based applications, asynchronous replication schemes across groups of inexpensive servers have become popular for internet-connected applications. Real-world installations often make use of MySQL with its built-in asynchronous replication and PostgreSQL with add-ons such as Slony, with large web-oriented companies such as Google and Skype open-sourcing versions of internal tools for improving replication and functionality such as MySQL semi-synchronous replication [1] and an extensive suite of PostgreSQL tools.[2]

2.2 Important Database Concepts

We review here some relevant database concepts that we will touch on in later parts of this text.

[1] http://code.google.com/p/google-mysql-tools/wiki/SemiSyncReplicationDesign
[2] http://wiki.postgresql.org/wiki/SkyTools

2.2.1 Consistency

Consistency is an important area of study in distributed systems research, but our focus will be on two consistency models that are particularly important within the RDBMS and "NoSQL" worlds.

Serializiability is the property that a concurrent execution of transactions leaves the database in an identical state as if each transaction were run serially, i.e. non-concurrently.

Eventual consistency is a consistency model used in many large distributed databases that requires that all changes to a replicated piece of data eventually reach all affected replicas; conflict resolution is not handled and responsibility is pushed up to the application author in the event of conflicting updates.

2.2.2 Transactions

ACID is an acronym for the four required criteria of a transaction as defined by Jim Gray et al. [GR93]. A transaction, or a unit of work, in a set of modifications or views to a subset of the database that are subsequently committed or aborted. Informally, it requires that:

- (*Atomicity*) All modifications made by a transaction are applied in full, or none at all.
- (*Consistency*) The transaction leaves the database in a consistent state.
- (*Isolation*) Transactions cannot view modifications made by other concurrently running, but uncommitted, transactions.
- (*Durability*) When a transaction has been committed, it must survive system failure, such as a RDBMS or OS crash.

2.2.3 Locking

RDBMSs typically implement a core locking strategy to provide ACID-compliant transaction support. Generally these strategies can be classified somewhere on a *pessimistic - optimistic* spectrum [MN82]. A highly pessimistic locking strategy, such as row or block-based locking, involves an RDBMS holding locks on individual pieces of data and blocking any access to that data to proceed until the lock-holder is finished. Such strategies are considered pessimistic because they assume that data contention is a common thing and hold only one copy of a piece of data in the system at a given time.

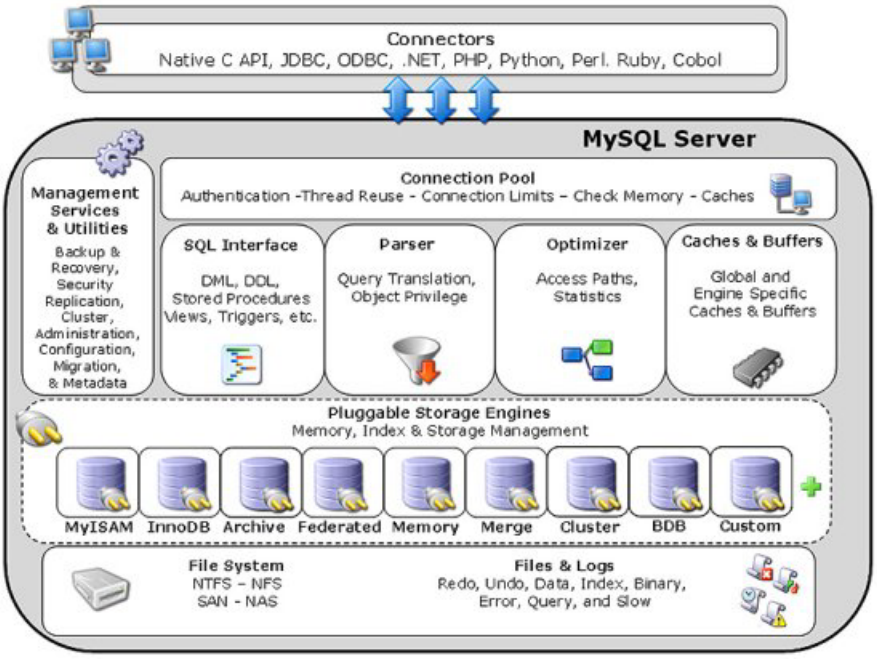

Figure 2.1: High-Level MySQL Architecture

Multi-Version Concurrency Control (MVCC) systems [BHG87], instead of depending on locks on pieces of data, allow new versions of rows to be written by writers; when such changes have been committed, the previous versions of the data are marked invalid although they likely still physically exist on disk. Write-transactions are able to proceed without interfering with other transactions and any possible inconsistencies resulting are handled at commit time. This strategy is more optimistic and based on the number of major database vendors implementing MVCC, has proven to be a better performing strategy for the majority of RDBMS workloads.

A highly optimistic strategy would be based on the assumption that databases are typically large and contention for the same data not a frequent occurrence.

2.3 MySQL Architecture

MySQL is architected in a modular way, as shown in Figure 2.1.[3] Connectors from a variety of languages are provided and implement the MySQL client protocol through TCP/IP or a shared memory interface. As SQL is the interface

[3] http://forge.mysql.com/wiki/MySQL_Internals_Custom_Engine#Overview

language for accessing data stored in the database, the parser, SQL interface and optimizer are the primary layers through which client connections interface when accessing data. These layers then make use of a re-implementable storage engine interface.

2.3.1 Pluggable Storage Engine Interface

The storage engine interface of MySQL is a cursor-like interface that enables any back-end service to be used as row storage for MySQL, provided that a required minimum of methods can be implemented using the service, with some engines providing far more capability than others[4]. Storage engine developers are provided with a series of storage-engine global and table connection-specific methods that need to be implemented as a C++ class. More details on our implementation of this interface are given in Chapter 4.

2.4 Standards Compliance

Due to competitive differentiation pressures between RDBMS vendors, extensions to SQL are often added and later become adopted by other vendors and eventually incorporated into the latest revisions of the SQL standard. MySQL has a number of proprietary extensions but does comply to the SQL:92, SQL:1999, SQL:2003 and SQL:2008 standards.[5]

2.5 Storage Engines

MySQL provides a number of storage engines with the base MySQL Community Edition. There are also many commercial third-party storage engines available targeting specific market segments and niches, one of which is GenieDB which we discussed in Section 1.1.

2.5.1 MyISAM

MyISAM is the default MySQL storage engine, featuring high-performance reads, but lacking transaction support and limited concurrent write capability. ACID-compliant transactions are not supported, although transaction-like semantics are possible through the use of MySQL SQL extensions such as LOCK TABLE and

[4] http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html
[5] http://dev.mysql.com/doc/refman/5.1/en/extensions-to-ansi.html

UNLOCK TABLE. Despite its many drawbacks, its speed and simplicity make it popular as a general-purpose storage engine.

2.5.2 InnoDB

InnoDB is an ACID-compliant, MVCC-based transactional storage engine for MySQL, originally developed by Innobase Oy. It is the most appropriate storage engine for high-concurrency, high-transaction workloads.

2.5.3 BerkeleyDB

The BerkeleyDB (BDB) storage engine was deprecated after version 5.0 in favour of the InnoDB storage engine, which provided many of the same features but with superior performance and functionality.

2.5.4 Other Storage Engines

Archive

Only supports SELECT and INSERT statements and does not support indexes. Data is compressed before storage on disk. Used as an efficient way of storing large amounts of data that has infrequent access or relaxed response time requirements.

Memory/Heap

An in-memory storage engine that provides high-performance for applications requiring a SQL interface to temporary data but with no requirement for persistence.

Federated

Provides the capability to link to other MySQL tables and databases over a network connection in a transparent fashion.

Chapter 3

Tapioca

Tapioca is a system developed at the University of Lugano, Switzerland, that provides ACID-compliant, strongly consistent transactions with a simple interface. Transactions proceed optimistically, with a certifier responsible for deciding which transactions must be aborted at commit time in order to keep the database in a consistent state. This enables high performance in databases where contention for the same data is infrequent. This is in contrast to traditional database systems that proceed more pessimistically by locking rows or even entire tables.

3.1 Interface

Tapioca presents a simple core interface through C or Java. The core functions are:

- *get(k)* : returns value stored for key k, or NULL if the k does not exist.
- *put(k,v)* : replaces value stored for k with v if k exists, or adds it otherwise.
- *mget(k[])* : returns value stored for array of keys $k[]$.
- *mput(k[],v[])* : replaces values stored for array $k[]$ with $v[]$ if elements of $k[]$ exist, or adds them otherwise.
- *commit()* : commits the currently active transaction.
- *rollback()* : aborts the currently active transaction.

There is no explicit *begin()* call to start a transaction; any call to *get(k)* or *put(k,v)* implicitly begins a new transaction. Only simple transactions are supported: no concept of a savepoint or transaction nesting exists.

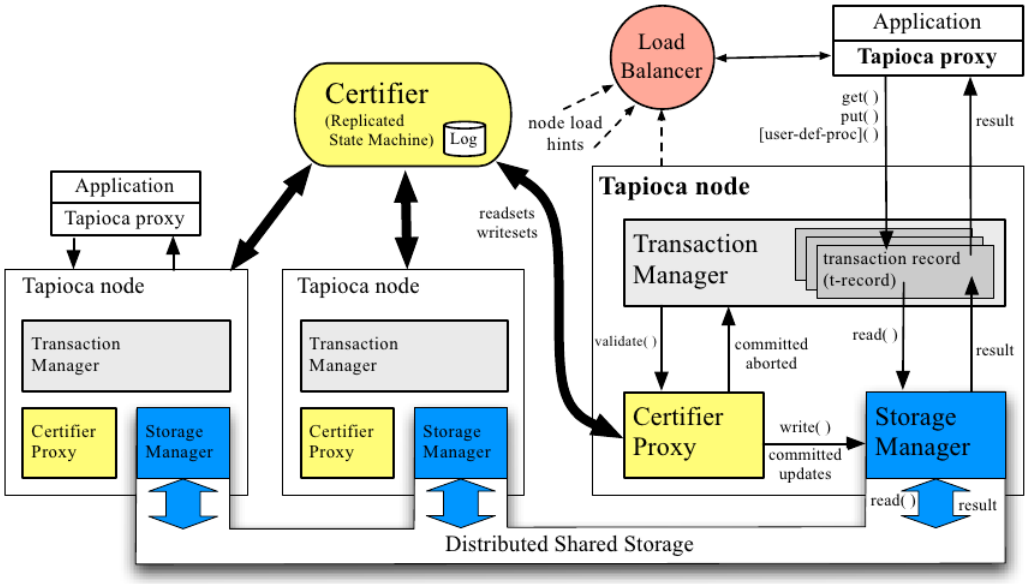

Figure 3.1: Tapioca Architecture

3.1.1 Properties

A typical Tapioca database will contain multiple nodes. A client can make a TCP-based connection to any given Tapioca node and an identical view of the database is provided to a client connected to a different node. Applications written to use BDB should have relatively few problems using Tapioca, with the obvious advantage of Tapioca being a scalable, distributed system.

3.2 Related Systems

In terms of functionality and properties provided, Scalaris [SSR08] is the closest system we are aware of. It provides strongly consistent transaction semantics and can scale to arbitrarily many nodes.

3.3 Architecture

Figure 3.3 shows a high level architecture diagram of Tapioca.

3.3.1 Transaction Manager

The transaction manager is responsible for tracking the keys read and written by a transaction, also known as the *readset* and *writeset*. These sets are used later by the certifier in order to decide what transactions must be aborted, if any, in order to maintain the consistency of the database.

3.3.2 Storage Manager

The storage manager provides consistent hashing-based partitioning of key values among the nodes in the system. Tapioca makes extensive use of multicast: when a key is broadcast, each node decides which keys it must keep based on the hashing scheme. Keys can also be cached locally in order to improve performance.

3.3.3 Certifier and Ring Paxos

The certifier is what enables Tapioca to provide serializable transactions. The certifier receives all transactions that are to be committed, and based on the read and write sets of each transaction, decides which transactions should be committed and aborted. The transaction log is maintained as a replicated state machine using the Ring-Paxos library [MPSP10].

3.3.4 Embedded Transactions

In addition to supporting transactions through the standard API, Tapioca has the capability to execute more complex transactions directly within the Transaction Manager, improving performance through reduced communication overhead.

3.4 Performance

Tapioca offers comparable performance to BDB in a single-node case but has the advantage that transaction throughput scales near-linearly with additional nodes added to the system. Latency also remains low as nodes are added. [Sci10]

3.5 Limitations

Tapioca has some built-in limitations that ultimately restrict the types of RDBMS workloads that can be used effectively. Due to its reliance on multicast, UDP is used extensively, and the current limit of a complete transaction is 64KB.

There is also currently no explicit virtual memory manager: Tapioca is intended to be an in-memory database, although theoretically swap space could be used as overflow. This is currently untested, however. It should be stressed that although Tapioca is intended to be run in-memory, it *does* persist data to disk, and it is capable of being shutdown and restarted, unlike in-memory systems such as Scalaris[1].

[1] http://code.google.com/p/scalaris/wiki/FAQ#Is_the_store_persisted_on_disk?

Chapter 4

MoSQL: A Tapioca-based MySQL Storage Engine

The core contribution of this thesis is the development of a working system that provides RDBMS capability resting on a distributed, key-value system as a persistence layer, rather than a traditional disk or disk-abstraction such as a storage controller. Our storage engine is built in a mixture of C and C++, with our current prototype containing approximately 6000 lines of code.

4.1 Key-Value Mapping

Tapioca, and key-value systems in general, clearly provide a far simpler interface than that of RDBMSs. As discussed in Section 1.1 however, it is established that a transactional key-value system such as BerkeleyDB is capable of serving as the basis of a transactional MySQL storage engine.

4.2 Architecture

The MoSQL storage engine effectively turns a MySQL Server into a piece of middleware for interpretation of SQL statements. Existing applications need not make any significant changes in order to leverage the scale-out capability provided by MoSQL, they only need to make use of some type of connection pooling software to distribute connections to available MySQL servers, or "MySQL nodes" in our parlance.

4.2.1 Flexible MySQL Node Allocation

As can be seen in Figure 4.1, Tapioca provides an abstraction of a single database spread across many nodes. In turn, any number of MySQL nodes can be connected to the individual Tapioca nodes. Each MySQL node sees the same data. One reason to add MySQL nodes, possibly even more than exist Tapioca nodes, is that there is still considerable processing required on the individual MySQL nodes in order to respond to SQL queries. A current limitation is that a MySQL node is not capable of load balancing its connections across more than one Tapioca node.

Both Tapioca nodes and MySQL nodes can be added and removed *online*.

4.3 Metadata

A convenient aspect of Tapioca is its support for hashing arbitrary key data, including binary buffers. This allows great flexibility in how the abstraction of a series of schemas, tables, rows and columns is created.

The Tapioca system itself serves as a convenient metadata store. We use Tapioca to transactionally manage sequence identifiers for important metadata such as table IDs, B+Tree IDs, etc, as well as storing metadata for the root elements of the various B+Trees stored.

4.4 Storage Engine Internal Architecture

The developer of a MySQL storage engine is required to interact with two key data structures: the *Handler* object, which must be overridden, and the `handlerton` struct. Figure 4.2 shows the interaction of the key data structures within MySQL and Tapioca.

Handler object instances are created when a client references a table by way of a SQL statement. MySQL manages a pool of such objects internally for performance reasons and these references can be passed from one thread to another.

4.4.1 Client Threads

The MySQL server process (mysqld) operates as a single process and so all client connections are spawned as threads. MySQL provides thread-local storage where thread-specific data can be managed. The TCP connection to Tapioca is managed within thread-local space, thereby enabling reads and writes to be done to multiple tables within a single transaction scope.

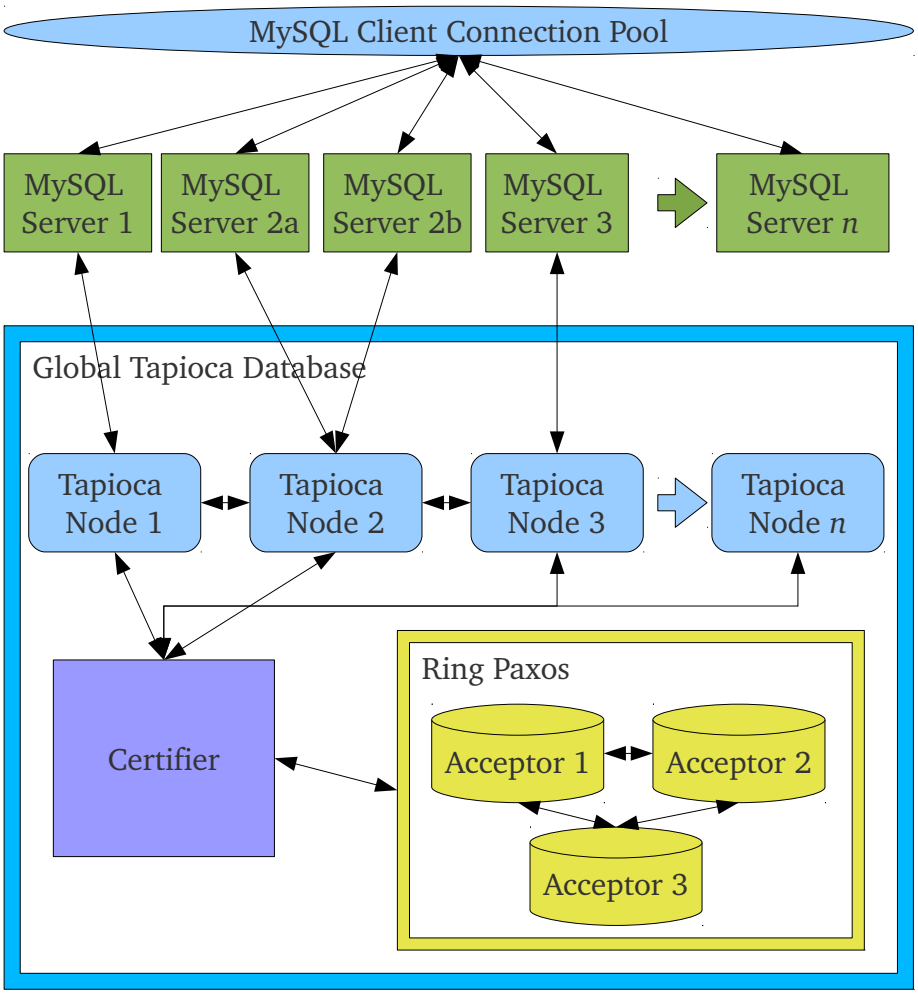

Figure 4.1: MoSQL Global Architecture

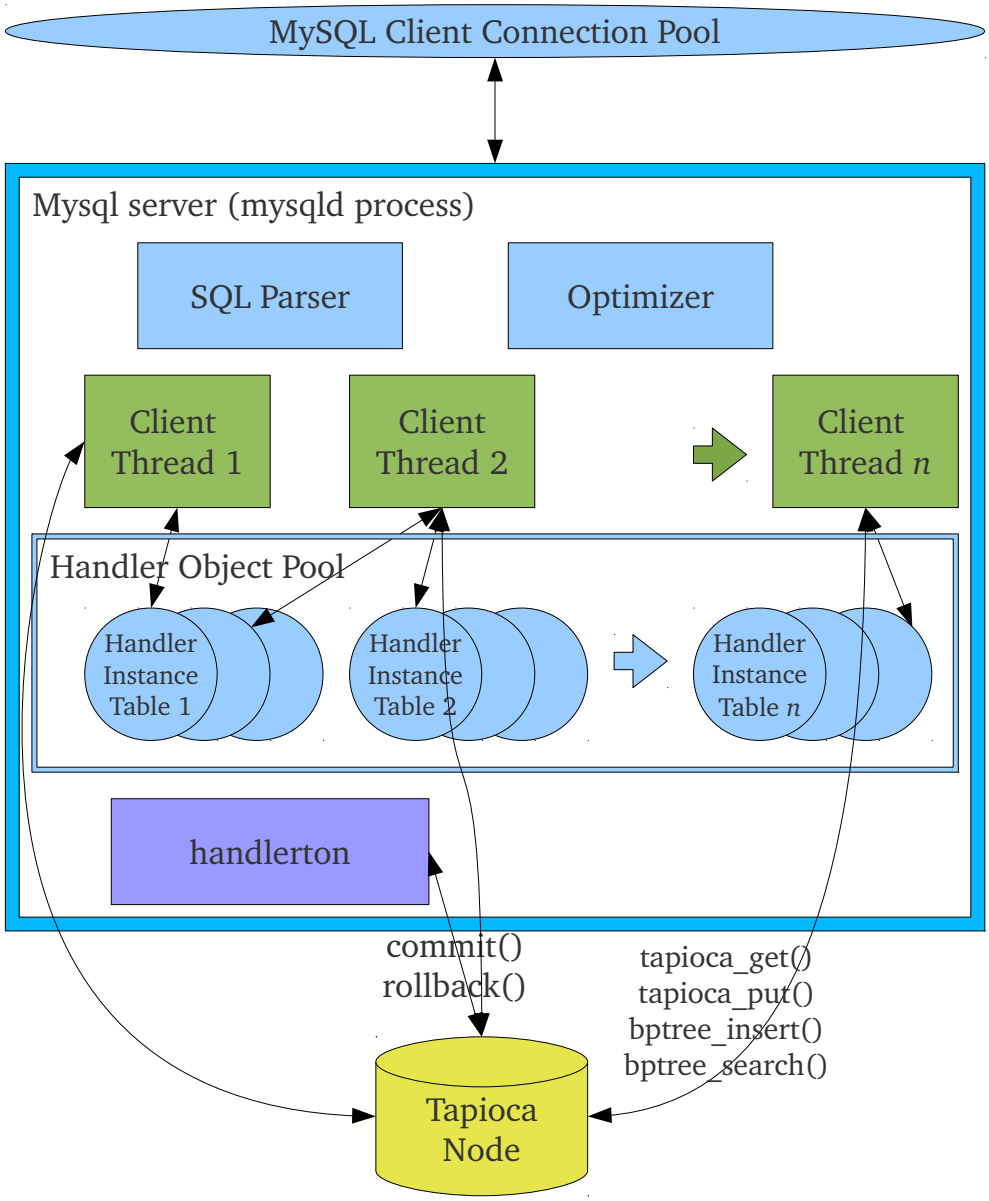

Figure 4.2: MoSQL Storage Engine Internal Architecture

4.4.2 Handler object

The Handler object is a type of cursor that is instantiated for a table. It does the bulk of the work, managing primary operations such as reads, writes, etc. It is responsible for table creation and removal, row storage and retrieval, as well as index-based retrieval and navigation, discussed in detail in the sections to follow.

4.4.3 handlerton struct

The handlerton is a storage-engine global structure, responsible for global operations such as transaction commit and rollback. The handlerton is not a C++ object, and methods to handle functionality such as commit are passed as function pointers. Global metadata for all tables and B+Trees existing in the Tapioca database are also stored here.

4.5 Table Management

The MoSQL Handler implements the following table management methods:

- *create()* : On table creation, MoSQL first checks Tapioca for the existence of an active table with the same name. If the table does not exist, it retrieves the next table id from the sequence maintained in Tapioca. This table id forms the header of any key that is retrieved from Tapioca. If it exists, the method is effectively a pass-through, allowing MySQL to create its internal structures to identify the table to the upper layers like the SQL parser. The primary key index, along with secondary indexes, are created within Tapioca at this time as well.
- *drop()*: Does not actually remove anything, but merely marks the table id within Tapioca as inactive.
- *repair()*: This method is used for multi-MySQL node configurations; it updates the internal metadata of the tables that a MySQL node is aware of.

4.6 Row Storage and Retrieval

MoSQL implements three core methods for persisting and retrieving data: *index_read()*, *write_row()* and *update_row()*.

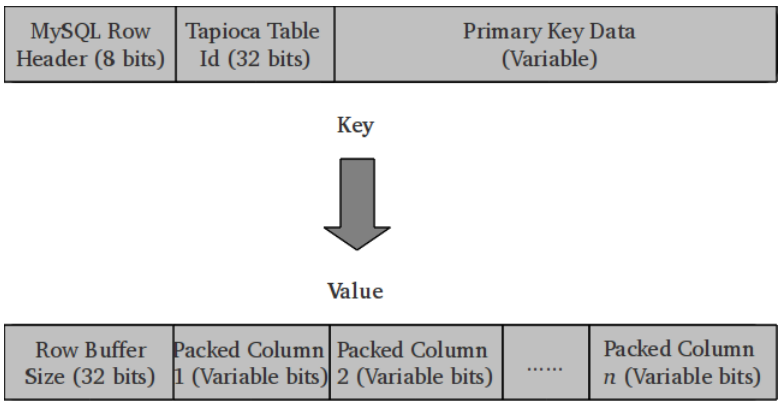

Figure 4.3: Components of Key/Value Pair Used to Identify MySQL Table Rows

When the MoSQL storage engine receives a request to persist data from MySQL (e.g. from an INSERT statement), the following steps are taken:

- Create key and value buffers that will ultimately be the key and value stored; prepend the header to the key buffer (i.e. the table id).
- Loop through the columns of the table and pack, using the MySQL *pack()* API, each column into the value buffer.
- Loop through, in primary-key order, the primary key fields of the table and pack these into the key buffer.
- Store in Tapioca, with *tapioca_put()*, the resulting key and value buffers.

The resulting key and value buffers stored in Tapioca are shown in Figure 4.3.

Requests to read values, when requested uniquely by primary key, go through a similar procedure:

- The values of the primary key values to be looked up are packed into a key buffer in the same manner as above.
- The generated key buffer is used to retrieve a value buffer with *tapioca_get()* .
- The value buffer goes through an inverse unpacking procedure to retrieve discrete field data that is returned to MySQL.

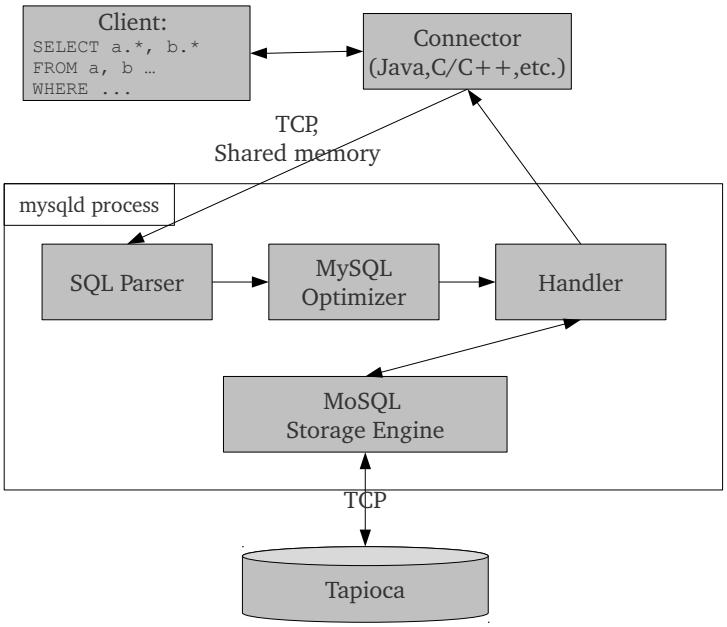

Figure 4.4: Select Statement Round Trip From Client to Tapioca

In principle a B+Tree is not required to retrieve individual rows from Tapioca to MySQL when based on primary key. In fact the B+Tree is skipped in many cases in our implementation if the primary key value is already known in advance. There are legitimate uses for such a light-weight storage mechanism and we are exploring enabling such a mode for applications that require very high speed primary key-retrieval performance, but have no requirement for the ability to scan, or know what keys have been stored. For more typical RDBMS usage scenarios, and in order to implement the remaining storage engine methods for scanning and non-primary key based retrieval, the functionality enabled with a B+Tree is required, as discussed in the next section.

Figure 4.4 shows the high level flow of a simple SELECT statement, illustrating the interaction with the two interface points of the MoSQL storage engine for the process described so far: the MySQL Handler interface and the Tapioca key-value data store.

4.7 Distributed B+Tree

One of the key features of the MoSQL storage engine is its support for a distributed B+Tree, maintained inside the Tapioca key-value store. The B+Tree is a traditional implementation, supporting variable cell sizes, recursive search and "sequential" reads along the leaf nodes. However, in place of system calls such as *fread()*, *fwrite()* and *fsync()* to read, write and flush data to local disk storage, we have calls to *tapioca_get()*, *tapioca_put()* and *tapioca_commit()*.

The B+Tree implementation is functional entirely on its own, and can be used as a library for distributed indexing of any data.

4.7.1 Interface

Our B+Tree supports the typical B+Tree operations:

- *search(k)*: returns the value stored at k, or NULL if not found.
- *insert(k,v)*: inserts the value v for key k into the tree; returns error if k exists.
- *update(k,v)*: updates the value with v stored for k.
- *commit()*: commit all currently applied changes to B+Tree.

The API described above is a summary of the core methods; more methods are required to be called for typical usage in order to handle complications such as support for multiple field types, ability to switch from one tree to another and for connecting to the database. Notably, the current B+Tree implementation does not support *delete(k)*; the current version of Tapioca does not provide a *delete()* primitive. While this does not prevent an implementation from making use of flags in the B+Tree to indicate active and inactive cells, it was decided early on that this complication could be pushed up to the client layer to reduce the scope of the implementation effort. We plan to implement a *delete(k)* call in a future version of the B+Tree.

4.7.2 Variable Field Support

In order to make it possible to index arbitrary types of data with different collation strategies, our B+Tree implementation permits the user to define a specific number of fields that will be indexed, along with a pointer to a comparator function that determines whether one field is smaller than, equal to or greater than the other (similar to how the standard C *qsort()* operates). Currently the length

of the fields to be indexed must be fixed, so only fixed-length fields such as numeric values or fixed-length character streams can be indexed. The length of the value indexed by the B+Tree can still be variably sized.

4.7.3 Variable Value Support

The B+Tree permits the storage of an arbitrarily sized buffer as a value, making the B+Tree itself a type of key-value store. While it was initially explored that the actual row data be stored within this buffer, it was eventually decided that the row data would be pointed to by the B+Tree to reduce the size of transactions against it. This buffer is still used for non-unique indexes, as a pointer to the actual primary key value.

4.7.4 Session Support

The B+Tree API also supports multiple B+Trees stored within the same Tapioca database, and multiple active client sessions. These features enable the use of a single Tapioca database to store multiple indexes on multiple tables with the API enabling seamless switching among them.

4.8 B+Tree Implementation

Our implementation is a standard B+Tree implementation where calls to write to disk are replaced with calls to the Tapioca API to read, write and commit data to the key-value database. As there is no support for directories, named files or anything beyond the simple key-value interface within Tapioca, our B+Tree is effectively an in-memory data structure where data is persisted and read as needed from Tapioca over the network. We use the TPL library [1] for serialization and de-serialization of C-based structs.

Figure 4.5 shows a global view of the components of our B+Tree, with the interaction of the important data structures such as the B+Tree meta node, the individual tree nodes, a B+Tree handle and session.

4.8.1 B+Tree Meta Node

The B+Tree meta node is a single node that maintains the current root key for a particular B+Tree id. All accesses to the main B+Tree library functions must

[1] http://tpl.sourceforge.net/

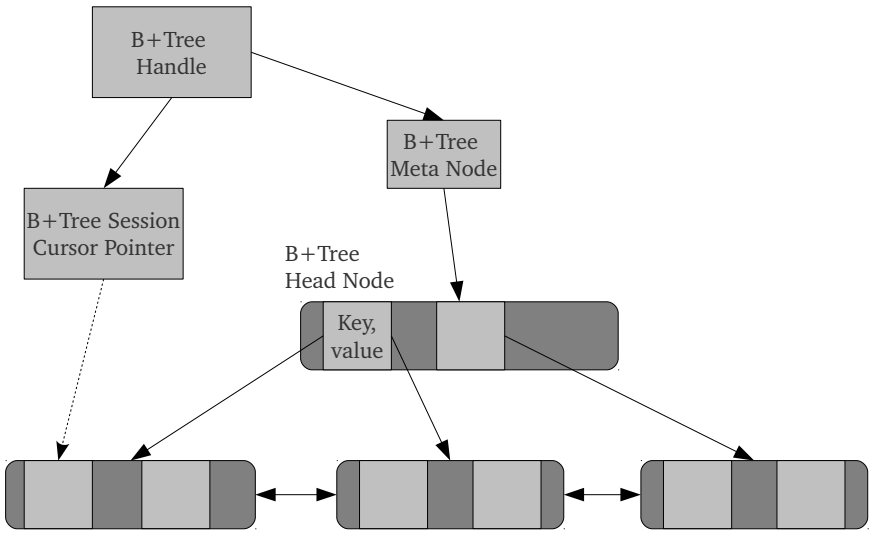

Figure 4.5: Overview of B+Tree Data Components

first retrieve this node in order to determine which key is currently at the root of the tree.

4.8.2 B+Tree Node

The `bptree_node` struct is our lowest level structure within the B+Tree implementation. Each node is identified with a 64-bit integer, in addition to storing the details of the keys and values stored inside as simple pointers.

Key Generation

Each B+Tree node must be uniquely identified in order to be retrieved from Tapioca, but for performance reasons we must have a scalable way of generating new keys that is not dependent on a sequence in a single critical section. To accomplish this, we assume that each connection to the B+Tree has generated, or can provide, a unique id that we call an *execution id*. This execution id is then prepended to the eventual B+Tree node id along with a local counter, ensuring that B+Tree node ids are unique as long as there is no overflow of either counter within the 64 bits provided for the id.

Figure 4.6 shows the bit structure of the key used to identify a B+Tree node

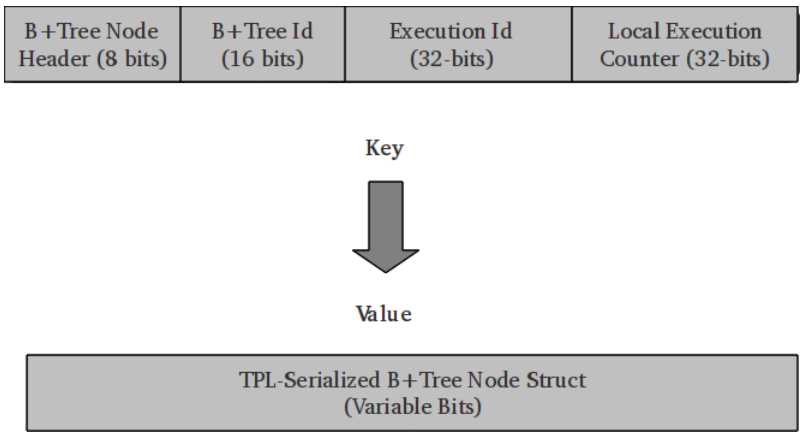

Figure 4.6: Components of Key/Value Pair Used to Identify B+Tree Nodes Inside Tapioca.

within Tapioca. The corresponding value stored in Tapioca is a byte array representation of the B+Tree node C struct serialized using the TPL library.

4.8.3 B+Tree Handle

Our B+Tree library provides its own wrapper for the Tapioca *connect()* method that opens a connection to Tapioca and initializes an execution id. This is a unique sequence persisted within Tapioca that is incremented transactionally each time a new connection to Tapioca is made.

4.8.4 B+Tree Session

The B+Tree session is a client data structure, not persisted to Tapioca, storing the currently used B+Tree (identified by a `bpt_id`), managing the local counter of creation of B+Tree nodes and maintains the current position of the cursor within the tree when searching or navigation is done. The session also maintains the field metadata for the index, which enables multiple fields with different collations to be used as the basis for the ordering of the tree.

4.8.5 B+Tree Field

The B+Tree field is a client-side structure that serves as metadata describing the type of data stored in a part of a key. The B+Tree field metadata stores the type

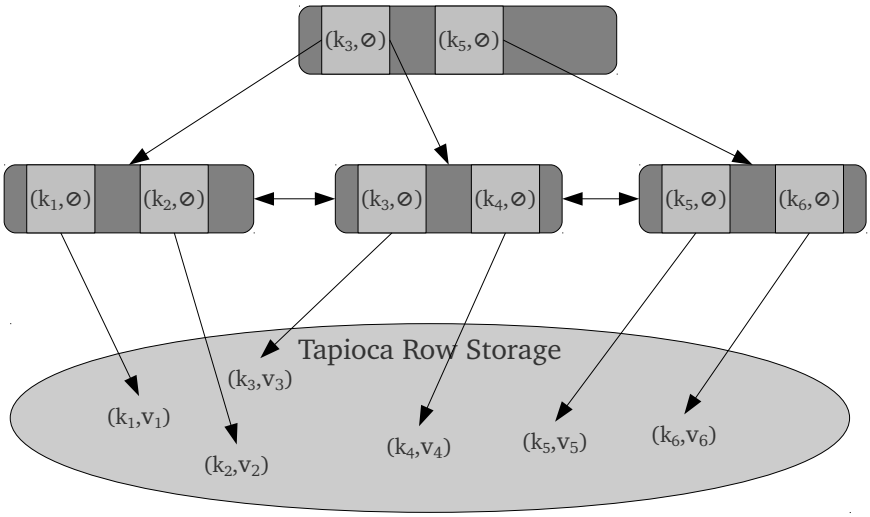

Figure 4.7: Example B+Tree for Primary Key Index

of data stored, such as integer or string, and a pointer to a collation function for the field implementing a comparator-like function signature.

4.9 MySQL Index Integration

4.9.1 Primary/Unique Keys

Tapioca includes no mechanism enabling the retrieval of keys that currently exist in the system. As such, once a key-value pair is inserted, if the key is lost, the value is also lost. Since a core capability of any RDBMS is to be able to retrieve all the rows in a table, each row that is persisted to Tapioca must be pointed to by a primary-key B+Tree. In this way, a table scan simply becomes a full index scan, starting from the first to last elements of the primary key index. This is why MoSQL requires that each table have a primary key.

As shown in Figure 4.7, the B+Tree maintained within Tapioca storage points to all rows stored in the database. A null value is stored in the value field for the B+Tree, due to the fact that the primary key value can be used to retrieve the data from Tapioca as part of a separate key.

For implementation-specific reasons, we do not include row values in the B+Tree itself, due to the relatively small amount of data that can be included within the scope of a transaction (64KB at the current time). This adds an

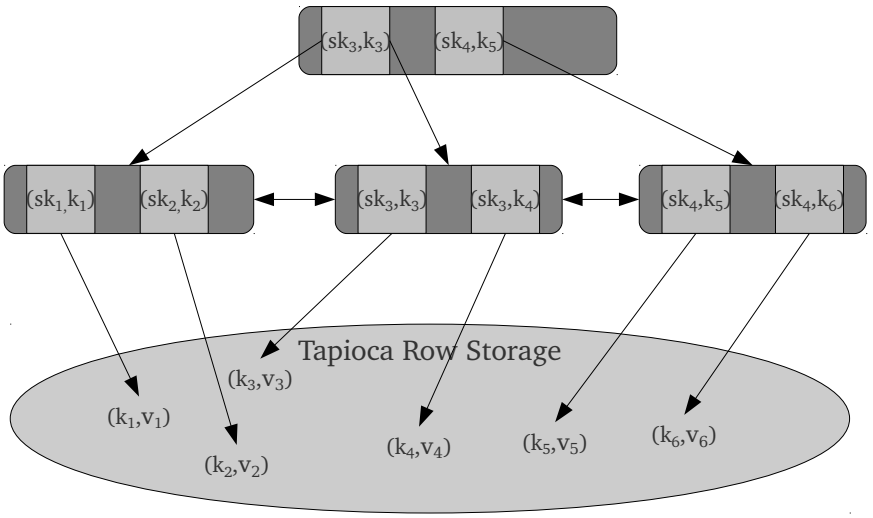

Figure 4.8: Example B+Tree for Secondary Key Index

additional network roundtrip to retrieve row data once the appropriate key has been found, but with the benefit of enabling an entire 64KB Tapioca value buffer to be used for the row.

4.9.2 Secondary Indexes

Secondary indexes are implemented also using B+Trees, with the difference that the keys store the indexed data, while the values store a primary key buffer that can then be used to retrieve the actual row data from Tapioca. Figure 4.8 shows how secondary indexes are handled.

4.10 MySQL Index Implementation

The MySQL storage engine interface provides a series of methods enabling index-based retrieval and scanning of rows in the database. In this section we discuss the important index-oriented methods of the MySQL *Handler* interface and how they are integrated with the B+Tree and Tapioca:

- *index_read()*: This method is used for all types of index-based reads, including single-row primary-key reads and non-unique accesses. It is used

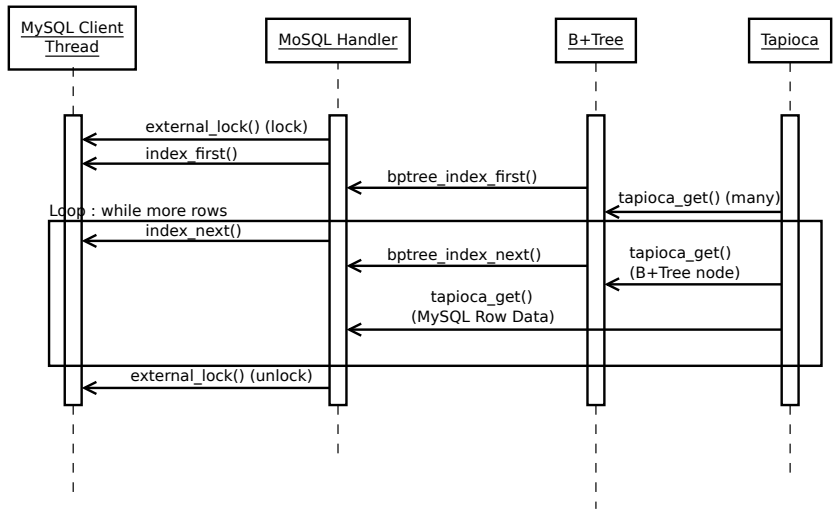

Figure 4.9: Simplified UML Sequence of Calls for Index Scan

to set a cursor that can be then used by calls to index_next() to traverse an index.

- *index_first()*: Sets the index cursor to the first position in the index and returns the row value.

- *index_next()*: More the position of the index cursor one forward and returns the row value. Requires that a previous call to a method has been made that will set the position in an index such as *index_read()*.

MySQL also specifies optional methods *index_last()* and *index_prev()* as complements to *index_first()* and *index_next()*, but these are currently unimplemented.

Figure 4.9 shows a simplified UML sequence diagram for a SELECT statement showing the important method calls from the client thread through to the B+Tree and Tapioca. The example given retrieves a variable number of rows with an index scan access pattern.

4.11 Optimizer Integration

The optimizer of any RDBMS is typically one of the most complex and performance-critical components, both from the RDBMS developer and SQL developer's perspectives. The difference between a poor execution plan and a good one on a complex query can easily mean a difference in runtime and system load in orders of magnitude.

Our integration with the MySQL optimizer, by way of statistics provided by the Handler, is simplistic. There are two principal reasons for this. The collection of statistics meaningful to a SQL-based optimizer, from a distributed key-value system where the number of keys is unknown, is non-trivial and a considerable research effort on its own.

Second, a typical RDBMS optimizer must take into account, among other things, the speed of random index lookups, sequential disk read speed and index scan speed. With all data stored in Tapioca, however, all accesses effectively become "random" accesses. Consider the query `SELECT COUNT(*) FROM tpcc.stock`, where `stock` is a large table. MoSQL will retrieve the first value in the primary key index and make successive calls to *index_next()*, resulting in a network-based fetch for every row in the table. The fact that these rows could be read sequentially from a disk-based database file is of little significance; reading these rows in random order across the network would likely perform the same. This is in contrast to random and sequential access on disk-based tables. In the MyISAM storage engine, the given query will result in a sequential table scan of the entire datafile; the index is not used at all. The optimizer, in some cases, may even decide upon a sequential table scan for a query where an index is available, if the expected cost of traversing the index is higher than simply reading the entire table from disk and filtering the rows as such. Such considerations are not an issue when a system such as Tapioca is used as the back-end storage.

As such, we provide statistics to the optimizer such that it will always choose an "index-scan". In the event of a "sequential" type scan, we have remapped the sequential access methods to do index-based scanning, making index-based scanning the only possible way of traversing the data.

4.12 Restrictions

4.12.1 Transaction Size

Due to current limitations of Tapioca, a transaction size greater than 64KB is not supported. This becomes more problematic as the table size grows, as the changes to the B+Tree(s) for a table are also done transactionally. For the OLTP

transaction sizes we have tested as part of TPC-C, this 64KB transaction size has proven to be enough.

4.12.2 Frequency of Transaction Aborts

Due to the optimistic nature of transaction management in Tapioca, a relatively high and unpredictable number of transaction failures can occur, in comparison to MVCC or other more pessimistic locking strategies used in most RDBMS implementations today. This requires the writer of the application to be prepared to handle transaction failure and attempt to retry even in cases where it may not seem necessary.

4.12.3 First Normal Form

E.F. Codd defined the various normal forms [Cod72] that have become a popular standard for representing data in an efficient manner in RDBMSs. First normal form informally requires that a table have no duplicate rows, requiring that there be a unique key.

As discussed previously, the MoSQL storage engine has the restriction that it must have a primary key index in order to retrieve the rows that exist in a table. This is equivalent to requiring that all tables be in first-normal form or higher; tables with no primary key are not supported. We may include in a future version an internally generated primary key sequence to enable support for tables not in first-normal form.

4.12.4 Indexing

Only B+Tree-type indexing is supported with fixed-length index fields; also multi-part text-based indexes are currently not supported.

Chapter 5

Performance

We have evaluated the performance of the MoSQL storage engine with a suite of tests chosen to highlight its current strengths, weaknesses and areas of future improvement. Where relevant, we compare results to existing (single-server) MySQL storage engines. We have developed our own micro-benchmark tool using the MySQL C API for primary-key and range-based selects, inserts and updates. The tool does not attempt to simulate interactive sessions, but rather focuses on applying as many operations against the server as possible. For TPC-B and TPC-C testing, we have used slightly modified open-source versions in order to work around some remaining limitations in the capability of the storage engine.

5.1 Hardware Considerations

The performance of any system is ultimately tied to the usage and capabilities of the underlying hardware. From the perspective of a single or multi-node RDBMS, the distinction between disk speed, network speed and memory speed is the most important, with disks, network and memory generally being an order of magnitude slower, respectively, in terms of latency and bandwidth. While the speed difference between CPU registers, L1/L2/L3 cache and main memory is also important, we will ignore this distinction and consider anything accessible at a faster level than the network or disk to be "memory".

5.1.1 Latency and Bandwidth

One of the principal goals of a virtual memory manager for a database is to avoid disk reads. For the time that a RDBMS waits for one access to disk to return, several thousand reads from memory can be done, a performance difference of

several orders of magnitude, with typical network access times somewhere in between.

The difference in theoretical bandwidth for newer disk standards such as SAS and Fibre Channel, however, is now within the same order of magnitude as that of the latest network standards such as Gigabit and 10-Gigabit. Given enough underlying physical drives, recent standards such as SAS and Fibre Channel can come close to saturating Gigabit and 10-Gigabit network connections respectively. Indeed, the iSCSI standard has now made it possible to use ethernet as a transport mechanism for storage controllers[SMS+], reducing the need to invest in typically proprietary, expensive storage transport standards such as Fibre Channel. Memory bandwidth in modern systems, however, is still considerably higher.

5.1.2 Importance to Tapioca and MySQL

Clearly, the best-case scenario for any relational system is to have everything locally resident in memory, providing low latency and high bandwidth. This is, of course, expensive, not scalable, not fault tolerant nor practical for many applications. A typical single-server RDBMS has a sophisticated memory manager (or it may rely on the underlying operating system filesystem caching) to help keep frequently accessed data in memory to minimize the number of accesses to very slow disk storage.

For systems that have the capability, network access is an intermediate step between slow disk access and fast memory access. The primary reason to depend on network access is to leverage the memory available in a cluster of relatively inexpensive servers as opposed to maintaining a large amount of memory in a single, monolithic and costly server. MoSQL in many respects is limited by the latency, and to a lesser extent, the bandwidth provided by the network, and so there are workloads where a single-server InnoDB instance, for example, will out-perform MoSQL by an order-of-magnitude. On the other hand, MoSQL suffers from none of the performance degradation that typical monolithic RDBMS servers do when managing datasets larger than available memory, due to the dependency on disk-based storage beyond available memory.

5.1.3 Hardware Configuration

Unless otherwise noted, the following experiments were run on our cluster of servers containing 4GB of RAM, 2 dual-core Opteron 2212 processors and Fujitsu MAX3073RC 73GB SAS drives.

5.2 Single-Server Reference Tests

We ran our micro-benchmark tool for the same set of tests against the MyISAM and InnoDB storage engines as a means of comparing the scalability and performance of the MoSQL storage engine. In addition to being useful due to the popularity of these storage engines, the comparison is interesting because all three storage engines employ different locking strategies: MyISAM being an extreme case of a pessimistic locking strategy, where an entire table is locked on write, Tapioca being at the other extreme, where all writes may proceed and a certifier decides which transactions may be committed, and MVCC-based InnoDB residing somewhere in between.

It should be stressed that, although we have run our tool against the MyISAM storage engine as a comparison point, transactions against MyISAM are *not* ACID-compliant; disk flushes are not guaranteed on commit and so latency figures especially for the insert and update tests are not a fair comparison to the results for InnoDB and MoSQL. Select performance is a reasonably fair comparison across all three storage engines, however.

Unless otherwise noted, we have used the following parameters for InnoDB:

- `innodb_file_per_table`: Writes each table to its own file in mysql data directory
- `innodb_log_size`: 100MB
- `innodb_bufferpool_size`: 1.5GB
- `innodb_flush_tx_on_commit`: 1

and the following setting for MyISAM:

- `key_buffer`: 500MB

5.2.1 InnoDB

We have run our tests against InnoDB using the `innodb_flush_tx_on_commit` parameter set to 1, guaranteeing that an `fsync()` or equivalent has been run on successful transaction commit, so transactions are ACID compliant. The effect of this parameter is reflected in the insert and update tests shown in Figure 5.3 and Figure 5.2: up to 32 concurrent clients transaction throughput is limited by disk I/O, which, on our test system is approximately 200 IOPS.

Select performance is an order-of-magnitude faster than MoSQL, as shown in Figure 5.1, due to the small size of the test with all data locally resident in

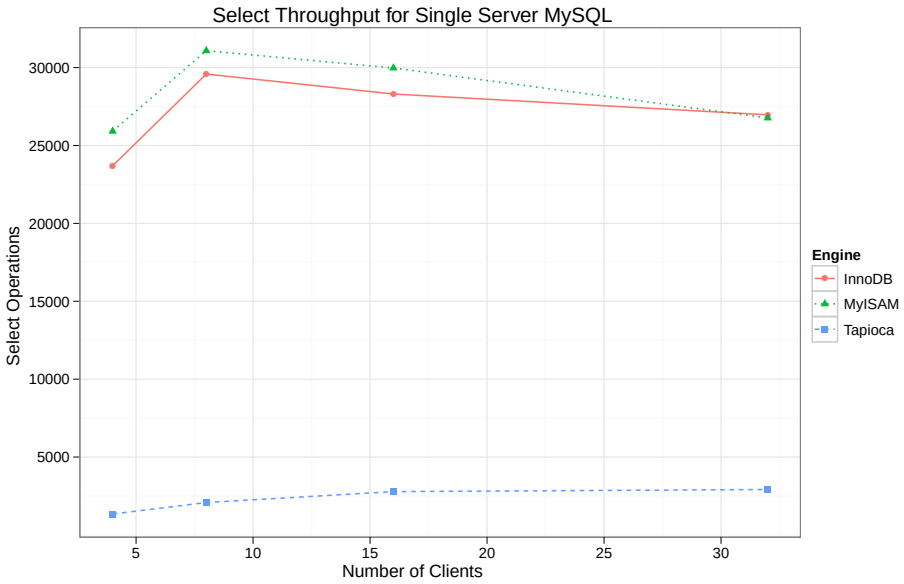

Figure 5.1: Select Throughput for MyISAM, InnoDB and MoSQL

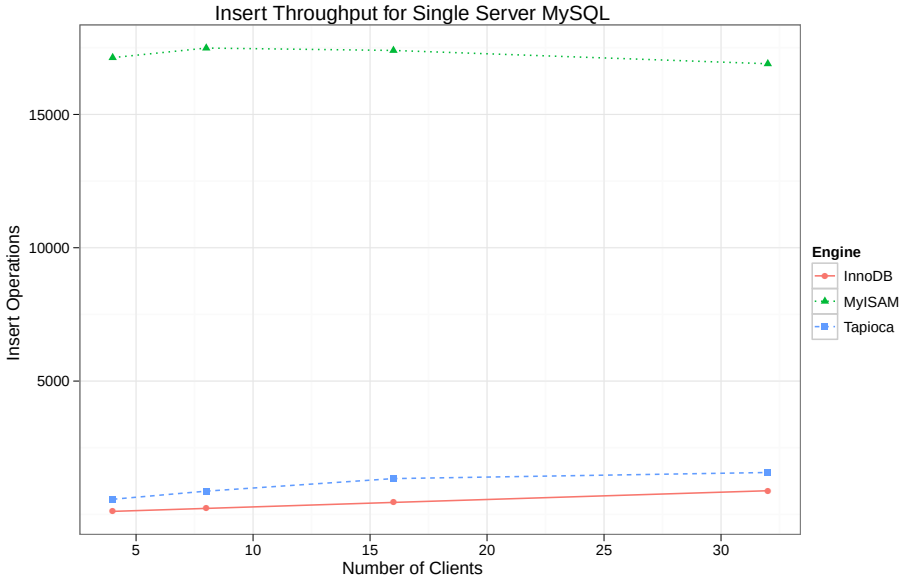

Figure 5.2: Insert Throughput for MyISAM, InnoDB and MoSQL

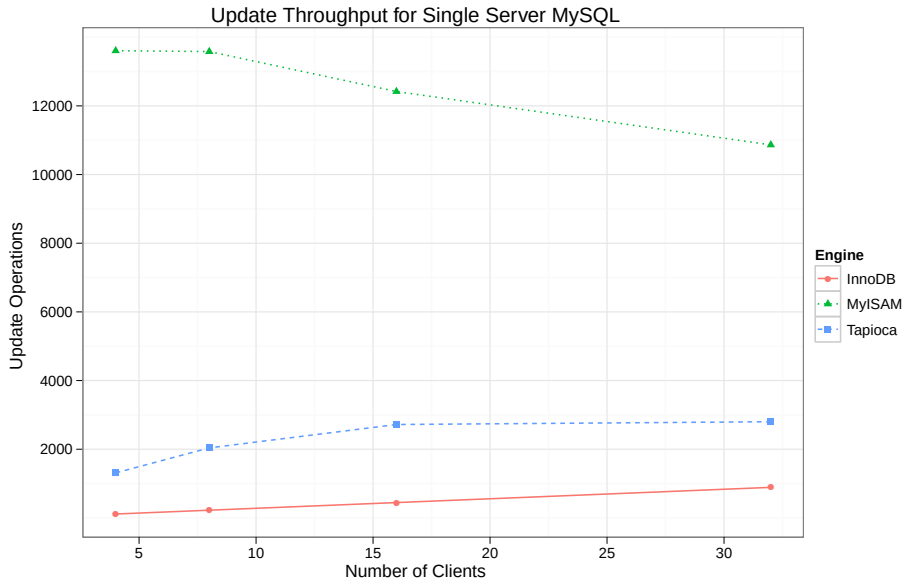

Figure 5.3: Update Throughput for MyISAM, InnoDB and MoSQL

memory. We observe that both MyISAM and InnoDB perform similarly, with a peak at 8 threads and a decline in throughput beyond that. MoSQL does provide the capability to cache data locally, so a currently unexplored option is to use co-located MySQL and Tapioca nodes where the CPU bottleneck is not so much of a concern to improve latency. We discuss some of these issues in more detail in Chapter 6.

5.2.2 MyISAM

MyISAM shows high performance inserts and updates even with few clients. This is due to the nature of locking with MyISAM and lack of durability: all writes lock the entire table, so the addition of threads merely adds overhead. Moreover, as transactions are not guaranteed to be flushed to disk, thereby violating ACID semantics, the operating system is free to flush data as it prefers. This results in very high performance as shown in Figures 5.1, 5.2, 5.3.

5.3 Primary Key-Based

Primary Key-based operations are a strength of the MoSQL storage engine due to the underlying characteristics provided by Tapioca. Primary key reads and

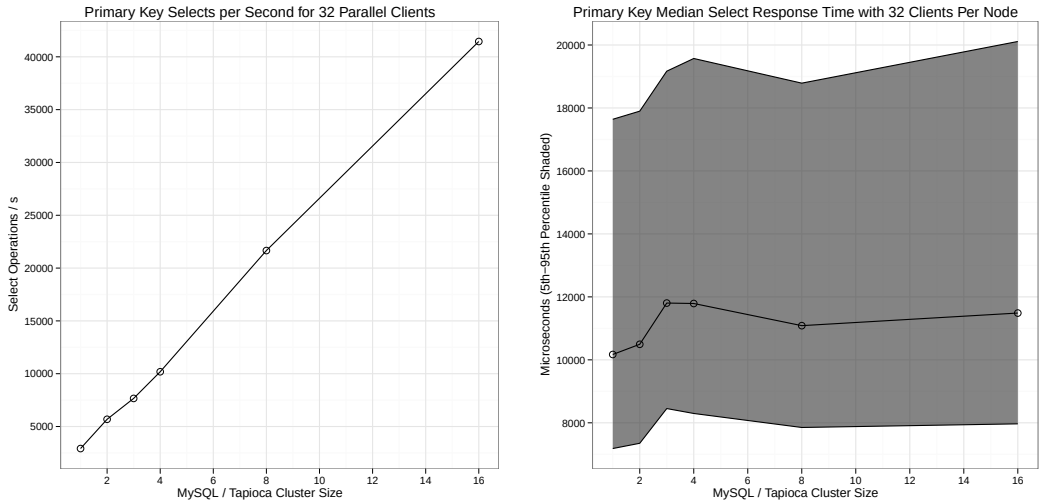

Figure 5.4: Select Throughput and Response Latency for 1 - 16 Nodes

updates are able to bypass the B+Tree completely as they are used to form the key that is used to store the row data in Tapioca.

5.3.1 Node Configuration

As the MySQL-based code tends to be CPU limited at high concurrency levels, our Tapioca data nodes are run on separate physical servers from the MySQL nodes. In addition, for all cluster configurations, three acceptors are used as part of the Ring-Paxos implementation used by Tapioca, and a single certifier node.

5.3.2 Reads

Figure 5.4 illustrates the scalability properties of the MoSQL storage engine. As nodes are added to Tapioca and MySQL nodes provided to connect to them, read performance scales linearly up to 16 servers, while average response time grows slowly but not significantly for a 32 client-per-node test (i.e. 512 total clients in the 16 node case).

5.3.3 Writes

Insert statements are considerably more complex for MoSQL to handle as they require transactional modifications to the B+Tree. The total number of rows in

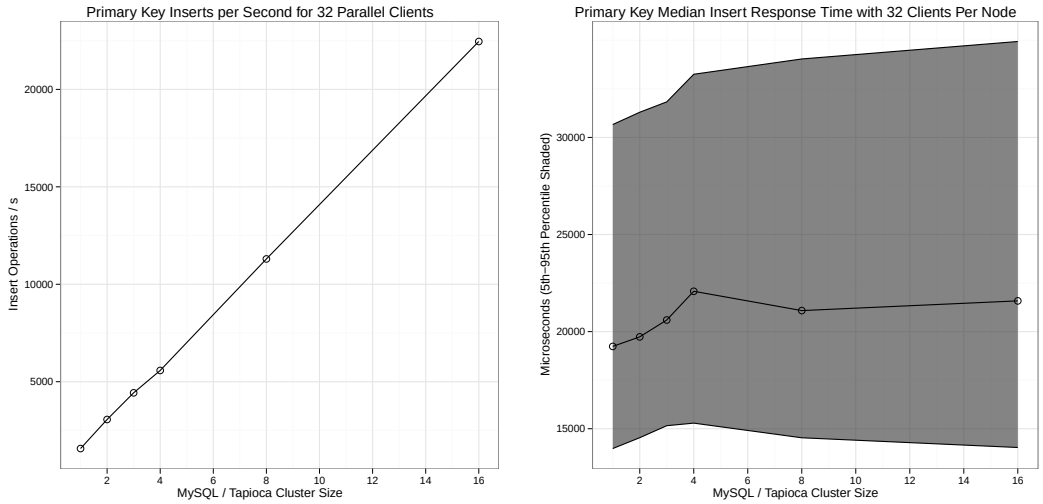

Figure 5.5: Insert Throughput and Response Latency for 1 - 16 Nodes

the table after the end of these tests was 256,000; the overhead of modifying the B+Tree adds approximately a 50% overhead compared to straight lookups to the underlying key stored in Tapioca. This is shown is Figure 5.5.

5.3.4 Updates

Updates show almost parallel performance as reads: this perhaps counter-intuitive result is due to the way in which the UPDATE statement is implemented. In the case of a primary-key based update with no secondary indexes, no changes are required to the B+Tree and so updates "pass-through" directly to row storage. The throughput and response time results are shown in Figure 5.6.

5.4 Databases Larger than Main Memory

The micro-benchmarks seen so far show fairly good results for single-server installations using MyISAM and InnoDB. The datasets tested so far, however, have been well below main memory in size and therefore have only stressed the internals of the storage engine when not presented with disk I/O bottlenecks. In this section we present an admittedly unfair comparison: we compare single-server MyISAM and InnoDB with a 4 Tapioca-node MoSQL cluster using 32 parallel threads, 2M total keys and 2KB row size, bringing the database size to approximately 4GB. As our test machine has 4GB of main memory, we used a 1.5GB

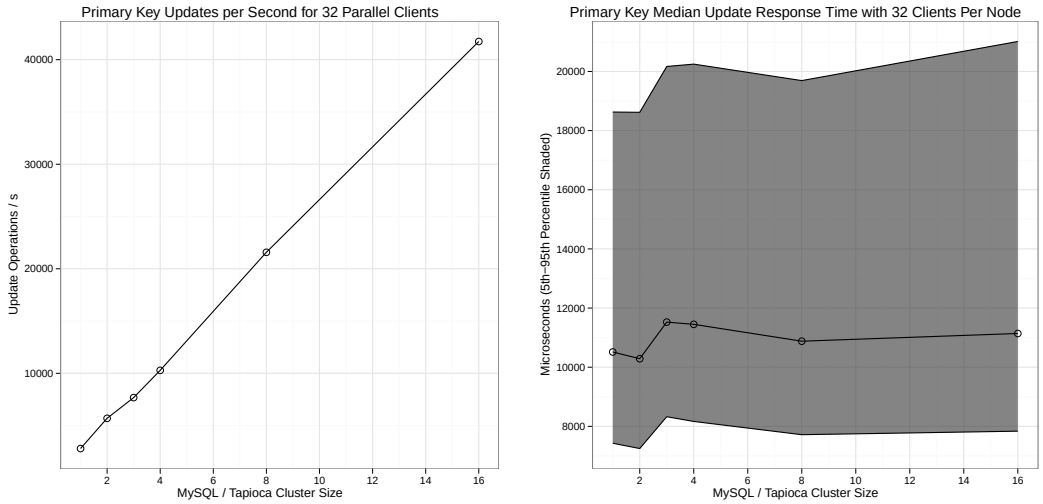

Figure 5.6: Update Throughput and Response Latency for 1 - 16 Nodes

InnoDB buffer pool, and left remaining memory to be used by the filesystem cache for MyISAM.

The results, shown in Figure 5.7, show that the MoSQL storage engine has a minor drop-off in performance relative to the results of the 4-node configuration shown in Figures 5.4, 5.5 and 5.6. InnoDB, on the other hand, shows a significant deterioration in performance due to a disk I/O bottleneck. MyISAM performance for writes remains high, due to the lack of an *fsync()* call, however updates and selects, which both must the access disk, both suffer similar performance degradation as InnoDB.

5.5 TPC-B

The TPC-B benchmark, while officially obsoleted by the Transaction Processing Council (TPC), still remains a useful test of core RDBMS performance. It is not intended as a general-purpose OLTP test, but rather as a stress test for evaluating the performance of the RDBMS on simple primary-key based selects, updates and writes.

We have slightly modified a Java-based implementation of TPC-B available at Launchpad[1] for our TPC-B testing. It provides many convenient features including pre- and post-benchmark integrity, isolation and atomicity checks.

[1]https://launchpad.net/tpc-b

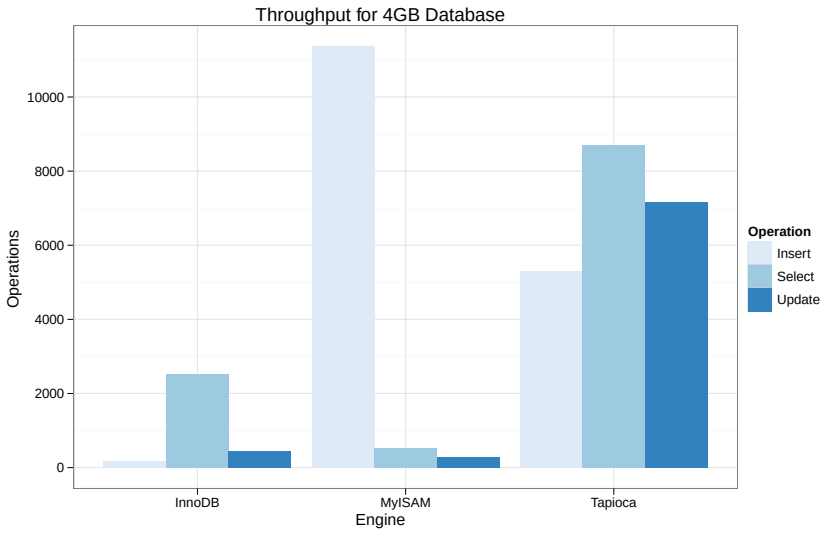

Figure 5.7: Transaction Throughput for MyISAM, InnoDB and 4-Node Tapioca with 4GB Database

5.5.1 TPC-B Overview

The TPC-B standard defines four simple tables related to the operation of a bank. Figure 5.8 shows the schema diagram used for our tests. A particular test run is configured with a "scale" factor, used to adjust the size of the test set. Given a scale factor S, the cardinality of the loaded tables is:

- ACCOUNTS: 100000S
- TELLERS: 10S
- BRANCHES - S
- HISTORY - 0 - the history of committed transactions is stored here

TPC-B transactions are simple and primary-key based, but modify several tables, making it a more complex test than our simple micro-benchmarks above.

5.5.2 Deviations from TPC-B

We have made few modifications to the base implementation, apart from some minor usability changes. Unless otherwise noted, we have tested the scale factor equal to the number of threads, resulting in 100,000 times the scale factor rows in the *ACCOUNTS* table and a benchmark time of 20 minutes.

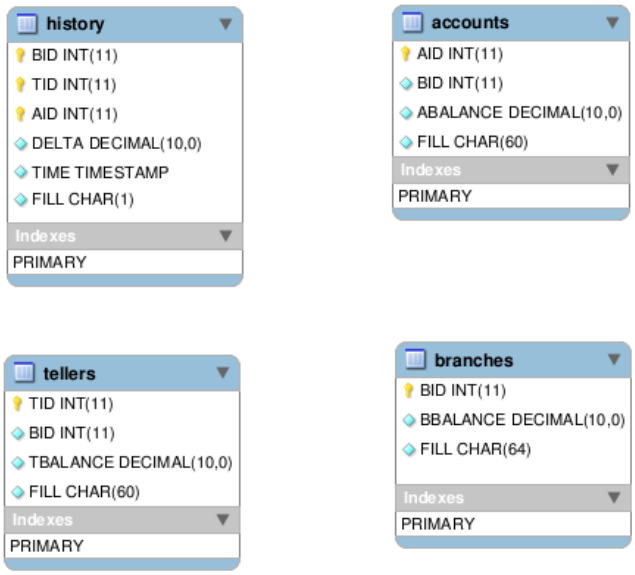

Figure 5.8: TPC-B Schema Diagram Used in Experiments

5.5.3 Performance: InnoDB vs. MoSQL

The TPC-B results highlight one problem with our current implementation and a limitation in Tapioca itself. Due to the very low cardinality of the branches table, it is often the case that two concurrent clients update the same row, or the same part of the B+Tree. Transactions in Tapioca all proceed "optimistically" until commit time, where the certifier is responsible for determining which transactions must be aborted in order to maintain consistency. The result is that roughly half of the transactions in this case are aborted. Had the standard mandated the branches table be of size $5S$ or $10S$, we would expect the abort rate to be much lower. Figure 5.9 shows the results for scale 4, 8 and 16 TPC-B runs with 4, 8 and 16 threads respectively.

A current limitation in the speed of bulk-loading has also restricted the size of the TPC-B tests we were able to do: the scale 16 test required 5 hours to load and a further 2 hours to perform consistency tests! We discuss this problem in the future work section of Chapter 6.

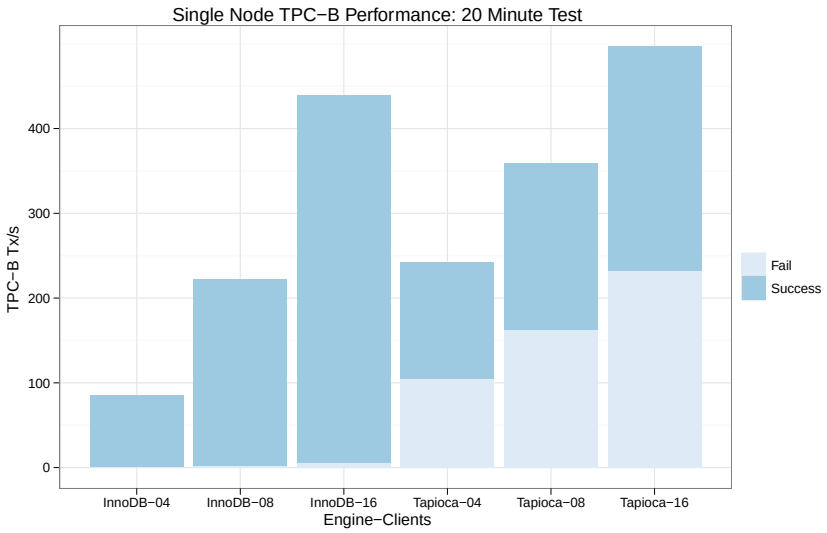

Figure 5.9: TPC-B Results for Tapioca vs. InnoDB With Given Scale Factor and Number of Client Threads

5.6 TPC-C

The TPC-C benchmark was created to provide an industry standard benchmark for typical OLTP workloads.[2] It tests the performance of a complex set of transactions simulating the usage of a terminal for maintaining orders and stock in an example warehouse. There are five core transactions that are tested randomly, with TPC-C defining the minimum percentage of each transaction type in order for a TPC-C test to be considered valid.

5.6.1 TPC-C Transactions

New Order

The New Order transaction simulates the entry of a complete order into the system. It is classified as a mid-weight, read-write transaction by the TPC-C standard, involving a mix of INSERT, SELECT and UPDATE statements.

[2] http://www.tpc.org/tpcc/detail.asp

Payment

The Payment transaction processes a payment made by a customer, updating their balance and updating the relevant summary statistics on the warehouse and district tables. It is intended to simulate a typical lightweight, frequent transaction where fast response time is important.

Order Status

The Order Status transaction simulates the querying of a customer's last order, retrieving data from several tables. It is a read-only transaction intended to be run infrequently. It is considered mid-weight by the TPC-C standard.

Delivery

The Delivery transaction is intended to simulate a typical deferred or "batch" type of transaction that can be run in the background on a given system. It is considered to have the least stringent response requirements and is executed infrequently.

Stock Level

The Stock Level transaction is read-only, querying the number of recently sold items that are below a certain threshold. As a heavyweight transaction, it intended to be run infrequently, with relaxed response requirements.

5.6.2 Implementation

For a TPC-C implementation, we have used a customised version of the Percona TPC-C implementation for MySQL available at LaunchPad [3], which in turn closely mirrors the sample implementation provided in the appendix of the current TPC-C specification (Revision 5.11 at the time of this writing). It was written specifically for MySQL, thus making it harder to compare the results against other RDBMSs. Written in C, it has the advantage of being fast, lightweight and targeted at the MySQL platform. The schema used in our tests can be found in Figure 5.10.

[3] https://code.launchpad.net/ percona-dev/perconatools/tpcc-mysql

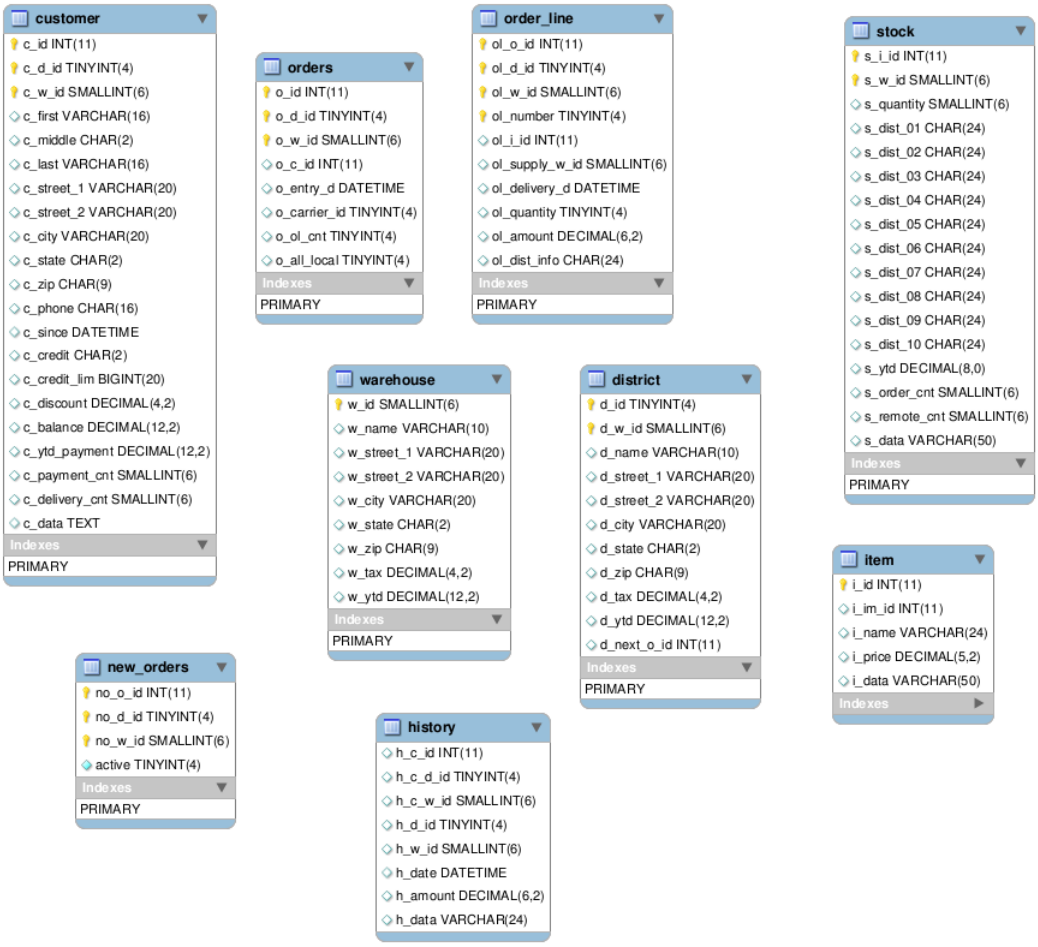

Figure 5.10: TPC-C Schema Diagram Used in Experiments

5.6.3 Known Deviations from TPC-C

While we have used what we consider to be a reasonably faithful implementation of the TPC-C standard, we must emphasize that we have not done a thorough review of the TPC-C spec and the closeness of our implementation to it. The following lists the deviations from the TPC-C standard implementation that we are aware of.

Delivery Transaction

One important feature notably absent from the MoSQL implementation at this time is the DELETE statement. At step 2 the delivery transaction expects to delete the orders from the new_order table it will process. This has been changed to an UPDATE statement which sets a flag on the new_order table, setting it inactive. This also required an additional column on the new_order table.

History

The MoSQL storage engine has the requirement that tables be at minimum in first-normal form. The history table of the TPC-C schema has no primary or unique key, and serves as an archive. In our tests, we use an InnoDB history table.

Index Usage

For tests which use the MoSQL storage engine, we have altered most of the SELECT statements to include a MySQL-extension FORCE INDEX statement with the primary key index; this is done to ensure the primary key is used in a range scan, which we have found to be the best performing access pattern. We discuss some of the optimizer integration difficulties in Chapter 6.

5.6.4 Performance

With our current implementation, aggregate TPC-C performance is an order of magnitude slower than InnoDB on a single-server configuration, as shown in Figure 5.11. This remains the case even when a TPC-C test is done with a database size greater than main memory. The performance of each transaction type is not uniformly worse, however, but a result of a combination of factors. A breakdown of the throughput of each individual transaction type for 1-8 concurrent clients on InnoDB and MoSQL can be found in Table 5.1.

The Payment transaction is required to be run at least 43% of the time and thus a fast response is critical for a good TpmC score. One of the first statements

Storage Engine	Tx Type Clients	1	2	4	8
InnoDB	Delivery	1.72	2.33	4.42	8.45
	New-Order	17.15	23.4	44.28	84.52
	Order-Status	1.73	2.35	4.43	8.48
	Payment	17.17	23.48	44.22	84.58
	Stock-Level	1.72	2.33	4.43	8.47
Tapioca	Delivery	0.08	0.08	0.05	0.03
	New-Order	0.78	0.92	0.68	0.52
	Order-Status	0.08	0.1	0.07	0.03
	Payment	0.73	0.9	0.63	0.22
	Stock-Level	0.08	0.1	0.07	0.05

Table 5.1: Transaction Type TPS for 2 Warehouse TPC-C Test for 1-8 Clients

in the Payment transaction is an update to the warehouse table. As this table is typically small (cardinality equal to the number of warehouses in the test), the entire B+Tree for the table is contained within one B+Tree node. This results in only one transaction successfully being able to commit once the full transaction has completed. Making the situation worse, due to the optimistic nature of concurrency control in Tapioca, the rest of the transaction will be allowed to run, using system resources although the work of all but one transaction will be rolled back. The symptoms of this problem can be seen clearly in Figure 5.12 – in the event of 8 concurrent clients with the 2 warehouse test, the retry frequency for the Payment transaction is nearly 400%, meaning that for each successful transaction 4 retries were required! This is a similar problem as we observed in the TPC-B tests.

In Figure 5.13, we show the average response times in a single-client case for each transaction type. An encouraging result is that for the New-Order transaction MoSQL responds equally fast as does InnoDB.

5.7 Performance Conclusions

Our performance evaluation highlights a few areas of weakness of our approach. Complex workloads which may access a large number of rows perform poorly due to the minimal buffering done by MoSQL. Also, workloads such as TPC-B and TPC-C that may create contention on very small tables will perform very poorly as concurrent clients are added, due to the concurrency control mechanism used by Tapioca. A particular strength, however, is in large, highly concurrent workloads where database size is greater than the main memory of a

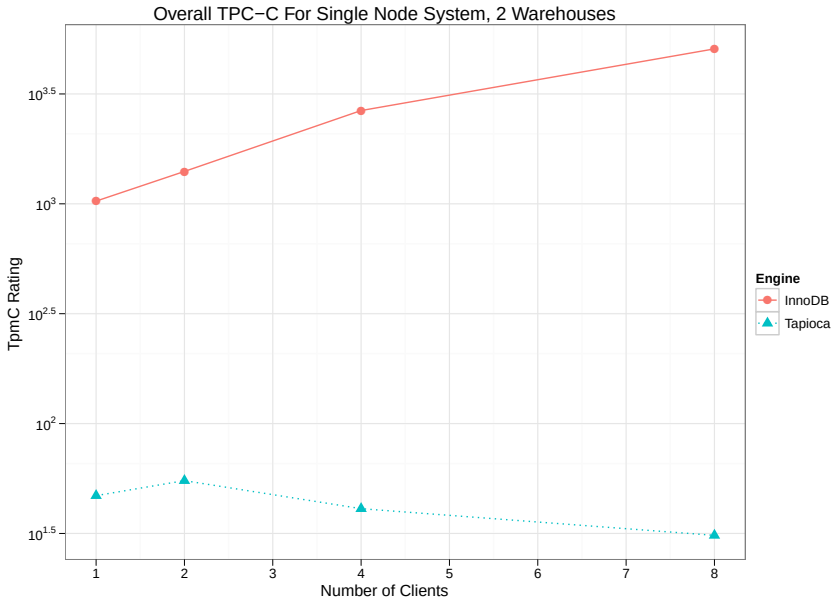

Figure 5.11: Overall TpmC for Single-Node, 2 Warehouse Test

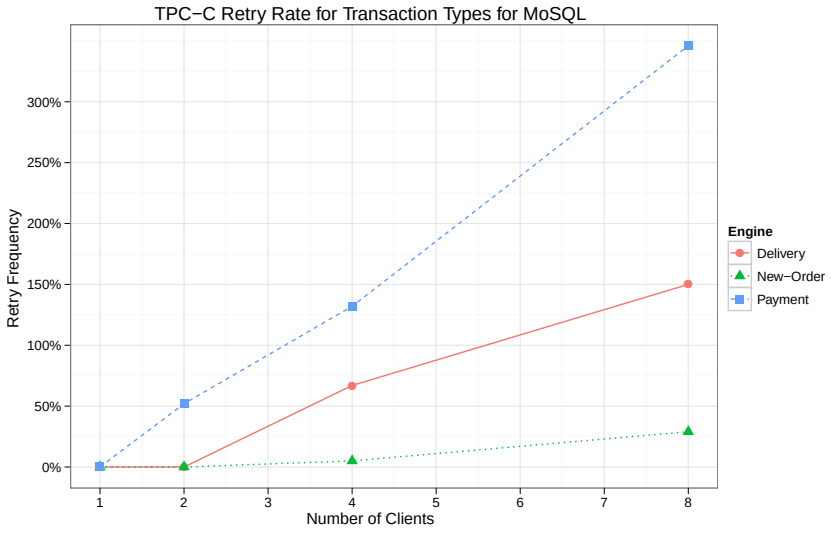

Figure 5.12: Retry Rate for Transaction Types for MoSQL Storage Engine

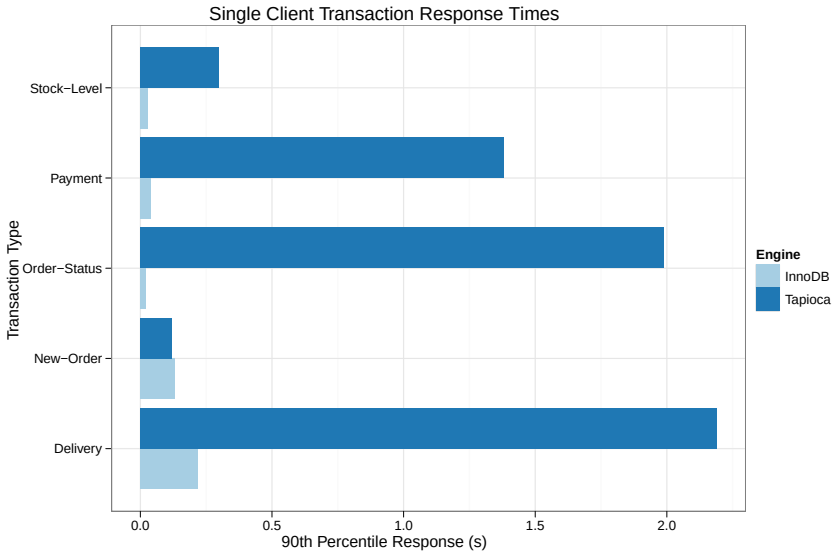

Figure 5.13: Transaction Average Response Times for TPC-C Single-Client, 2 Warehouses

single server, and where transaction size and the number of rows accessed per transaction remain relatively small.

Chapter 6

Conclusions

Relational database management systems are among the best-tested and relied-upon complex software systems in existence today; to have created a reasonably capable system within the scope of a Masters project we consider to be a success. The many deficiencies we outline here show that still much work is to be done if a complete prototype is to be created.

6.1 Future Work

6.1.1 Optimizer Enhancements

For reasonably simple queries, MySQL is able to select an access plan that exhibits good performance with Tapioca as the back-end store. Due to the minimalistic nature of our implementation of the engine, some optional methods that MySQL could make use of to perform a better scan, e.g. *index_last()*, are not implemented. MySQL does well to find alternate ways to execute the query, but these tend to be suboptimal.

We also have much work to do in finding better ways to provide meaningful statistics to the MySQL optimizer. However, given the distributed nature of the data in Tapioca, we suspect that it may present a considerable challenge to present good statistics to the individual, local MySQL optimizers split across many MySQL nodes in a multi-node configuration.

6.1.2 Bulk-Loading and Sequential Performance

A significant problem with the MoSQL engine at the moment is its very poor performance in reading and writing sequentially, particularly bulk-loading. This is primarily due to the inability of the engine to do batching of reads or writes,

which are provided by the *tapioca_mget()* and *tapioca_mput()* APIs. We anticipate that a relatively simple buffering/caching scheme could provide substantial improvement to some of the poor-performing queries of the TPC-C benchmark.

6.1.3 General Optimization

In our experiments, especially with highly concurrent, simple primary-key lookups as in the micro-benchmarks shown in Section 5.3, we have observed that CPU quickly becomes the bottleneck limiting performance, slightly under-loading the Tapioca node. Two MySQL nodes assigned to one Tapioca node, however, tend to overload a single Tapioca node, suggesting that the current ideal ratio of MySQL to Tapioca nodes lies somewhere between 1 and 2 for simple, highly concurrent workloads such as those we would expect to find in typical web applications. More performance tuning and research will be required to assess whether a performance bottleneck resides in our code.

6.1.4 Embedded Tapioca Transactions

Tapioca provides an embedded transaction capability that can be used to reduce communication overhead; our B+Tree currently resides external to the transaction manager, although we expect to see significant improvement in performance by making the core functions it provides as internal transaction manager stored procedures.

6.1.5 Usability Enhancements

Considerable work is also needed to make the engine more usable. Such features include:

- General support for ALTER TABLE statement, including the ability to change table layout after creation.
- Support for CREATE INDEX and DROP INDEX: indexes at the moment must be created at create time and cannot be modified thereafter.
- Support for AUTO_INCREMENT and tables with no specified primary key.
- Support for multi-part indexes with character fields.

6.2 Usage Niche

Our performance numbers highlighted a few areas of strength and weakness of our approach. We consider the MoSQL engine to be most appropriate for applications with the following characteristics:

- High transaction throughput: With additional nodes, we have shown that our system can near-linearly scale the number of TPS for simple transactions
- Relatively small database size: Since databases must fit in memory, our target database size must fit in the available main memory of a cluster. Assuming a cluster of 32 nodes with 8GB of memory each, this would put our maximum expected database size at 250GB.
- Simple transactions: Due to the underlying limitations of Tapioca, transactions that modify more than 64KB worth of data are not supported.
- Limited or non-time-critical use of large queries: since one weakness of our approach is in retrieving large blocks of data due to network overhead for each individual read, an application that must perform large queries in a time-critical fashion would perform very poorly, as shown by some of our TPC-C results.

Despite the limitations, we believe that our approach shows great promise for applications that fit within the criteria outlined above. As our implementation matures, we hope to demonstrate its performance using real-world applications.

Bibliography

[BFG+08] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264. ACM, 2008.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 5. Addison-wesley New York, 1987.

[Cod70] E.F. Codd. A relational model of data for large shared banks. *Communications ACM*, 13(6):377–387, 1970.

[Cod72] E.F. Codd. Further normalization of the data base relational model. *Data base systems*, pages 33–64, 1972.

[GR93] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[MN82] D.A. Menascé and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[MPSP10] P.J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 527–536. IEEE, 2010.

[Pat99] Giuseppe Paterno. NoSQL tutorial: A comprehensive look at the noSQL database. 1999.

[Sci10] D. Sciascia. Performance evaluation of dsmdb. 2010.

[SMA+07] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. 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*, pages 1150–1160. VLDB Endowment, 2007.

[SMS+] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Rfc3720: Internet small computer systems interface (iscsi). *IETF (Aprile 2004). URL: http://www. ietf. org/rfc/rfc3720. txt*.

[SSR08] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 41–48. ACM, 2008.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.