Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

MoSQL: An Elastic Storage Engine for MySQL

Alexander Tomic[1], Daniele Sciascia[1], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

We present MoSQL, a MySQL storage engine using a transactional distributed key-value store system for atomicity, isolation and durability and a B+Tree for indexing purposes. Despite its popularity, MySQL is still without a general-purpose storage engine providing high availability, serializability, and elasticity. In addition to detailing MoSQL's design and implementation, we assess its performance with a number of benchmarks which show that MoSQL scales to a fairly large number of nodes on-the-fly, that is, additional nodes can be added to a running instance of the system.

Report Info

Published
29 November 2012

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Relational database management systems (RDBMS) have had remarkable staying power in real-world applications despite many alternative approaches showing promise (e.g., XML-based storage, object databases). In recent years, however, the realities of scaling database systems to Internet proportions have made customized solutions more practical than general-purpose one-size-fits-all RDBMSs [32]. Despite the disadvantages of using a general-purpose RDBMS in comparison to more specific solutions, 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 an open-source RDBMS at the core of many multi-tier applications based on the "LAMP software stack" (i.e., Linux, Apache, MySQL and PHP). Although the LAMP stack initially thrived in environments where the cost, complexity, and capabilities of enterprise-grade frameworks and RDBMSs were prohibitive or unnecessary, MySQL has also been deployed in large and complex environments (e.g., Wikipedia, Google, Facebook, Twitter). Yet, despite its popularity, MySQL is essentially a standalone database server. Multi-server deployments are possible but provide weaker system guarantees than single-server configurations (e.g., weak isolation levels, absence of distributed transactions). This is clearly detrimental to the many applications based on MySQL that have evolved into large and mature systems with originally unexpected scalability, fault tolerance, and performance requirements.

The continuing migration of services and applications to the web has exposed RDBMSs to workloads that are larger, growing faster, and behaving more unpredictably. This trend typically results in over-provisioning in the difficult-to-scale database tier to ensure responsiveness for all expected ranges of client load, while costly and disruptive system upgrades to meet growing scalability requirements remain a reality. Elastic storage holds the promise of saving over-provisioning costs while maintaining high performance and low latency, especially for highly variable or cyclical workloads [2].

MoSQL, the distributed storage engine we have designed and implemented, provides near-linear scalability with additional nodes and strongly consistent, serializable transaction isolation. MoSQL stores data across several *storage nodes*, with each storage node containing a subset of the dataset in memory only. Although each node is responsible for a portion of the database, it provides upper layers the abstraction of a single-partition system: database entries not stored locally on a storage node are fetched from the remote

nodes responsible for storing them; for performance, remotely fetched entries are locally cached. This mechanism is far more efficient than the one used by standalone databases, which fetch missing items in the cache from an on-disk copy of the database.

MoSQL's storage nodes offer simplified concurrency control and single-threaded execution; parallelism can be exploited by deploying multiple node instances on a single physical server. Update transactions proceed optimistically: there is no global synchronization of update transactions across nodes during execution (i.e., no distributed locks and deadlocks [15]). At commit time, update transactions are certified; the certifier decides which transactions must be aborted in order to keep the database in a consistent state. Read-only transactions always see a consistent snapshot of the database and need not be certified. This mechanism enables high performance in large databases where contention for the same data is infrequent.

We have implemented all the features described in the paper and conducted a performance evaluation of MoSQL using the TPC-C benchmark. We show that MoSQL is capable of scaling TPC-C throughput sublinearly to 16 physical servers. With two physical servers, MoSQL is able to surpass the throughput of a single-server instance of MySQL using the standard transactional storage engine InnoDB, while still maintaining elastic capability and fault tolerance. We also demonstrate the elastic capabilities of MoSQL: we add clients to a running system until a given latency threshold is passed and then add nodes to the storage tier and launch a new MySQL node and redistribute the clients.

The rest of this paper is structured as follows. Section 2 details MoSQL's design. Sections 3 and 4 discuss isolation and performance considerations. Section 5 presents an experimental evaluation of MoSQL's prototype. Section 6 reviews related work and Section 7 concludes the paper.

2 System design

2.1 Model and definitions

We consider an environment composed of a set $C = \{c_1, c_2, ...\}$ of client nodes and a set $S = \{s_1, ..., s_n\}$ of database server nodes. Nodes communicate through message passing and do not have access to a shared memory. We assume the crash-stop failure model (e.g., no Byzantine failures). A node, either client or server, that does not crash is *correct*, otherwise it is *faulty*.

The environment is asynchronous: there is no bound on message delays and on relative processing speeds. However, we assume that some system components can be made fault tolerant using state-machine replication, which requires commands to be executed by every replica (agreement) in the same order (total order) [18, 26]. Since ordered delivery of commands cannot be implemented in a purely asynchronous environment [7, 13], we assume it is ensured by an "ordering oracle" [18].

The isolation property is *serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [4]. Serializability prevents anomalous behaviors, namely, dirty reads, unrepeatable reads, lost updates and phantoms [16].

2.2 Overview

The architecture of MoSQL decouples some of the components typically bundled together in monolithic databases. In particular, concurrency control and logging management are separate from data storage and access methods. In this sense our approach is similar to the model proposed in [20] and expands upon the work in [34] and [27]. Figure 1 shows the architecture of MoSQL.

In brief, MoSQL is composed of three main components:

- **MySQL servers** handle client requests by parsing SQL statements and executing them against the storage engine. MySQL allows third-party storage engines through its pluggable storage engine API. This allows us to plug in our calls to our distributed storage nodes.

- **Storage nodes** handle MySQL requests and keep track of the transactional state. Each storage node keeps a subset of the dataset and indexes in-memory only. Storage nodes act as a distributed store in that read requests are either served from local memory or from the memory of a remote storage node. This is effective since retrieving rows from a remote storage node is usually faster than retrieving rows from local disk.

- The **Certifier** is a replicated state machine that logs transactions on disk, ensures serializable transaction executions, and synchronizes system events (i.e., recovery, storage node additions and removals).

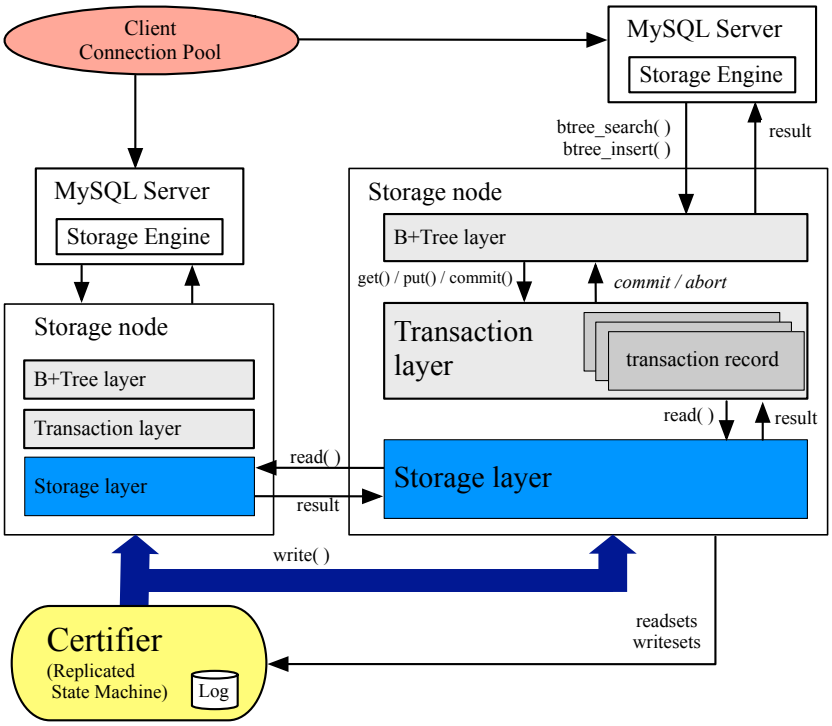

Figure 1: MoSQL global architecture.

At storage nodes, transactions proceed without synchronization. At commit time, storage nodes submit transactions to the certifier, which ensures serializable execution.

Any number of MySQL instances can be connected to any number of storage nodes. Depending on the workload, it may be advantageous to assign multiple storage nodes per MySQL instance, or vice-versa. We typically deploy one MySQL process together with a small number of storage nodes per physical machine, depending on how many cores are available in the machines.

In the following sections, we detail each one of MoSQL components.

2.3 Storage nodes

Storage nodes are divided into three distinct layers, where each layer builds upon the abstraction offered by the layer below. The bottom layer implements a distributed storage abstraction. Each storage node is assigned a subset of entries, and the storage layer provides operations to read and write such entries. This layer abstracts away the fact that entries are distributed among storage nodes: if a read operation accesses an entry that is not stored locally, the operation is turned into a remote read request sent to the node responsible for storing that entry. The transactional layer uses read and write operations provided by the storage layer to keep track of the state of active transactions. Some metadata is kept for each transaction, such as its snapshot timestamp (described later), and the set of entries it has read and written, i.e., the transaction's *readset* and *writeset*. This metadata is later used for certifying transactions. Finally, the B+Tree layer adds indexing capabilities and exposes the typical operations of a B+Tree index for searching, scanning and adding entries.

Storage nodes are "elastic" and can be added or removed while MoSQL is running. We distinguish two reasons for adding storage nodes: (1) to improve performance and accommodate sudden workload spikes; or (2) to increase aggregate system main memory. For the latter case, increasing the capacity of storage nodes requires adding one or more machines to the system and redistributing the dataset so that new storage nodes take ownership of a portion of entries to even out the load. Reconfigurations should be relatively rare and only necessary when the system reaches a certain threshold of memory utilization.

Reconfiguration can be disruptive, so we use *volatile storage nodes* to handle spikes in the workload. When spawned, a volatile storage node starts with an empty storage and does not take ownership of entries. As transactions are executed, volatile storage nodes will turn read requests into remote requests (using the same

mechanism we mentioned above) and cache the results of the remote requests locally. We demonstrate and evaluate the effects of adding volatile storage nodes in Section 5.

The **storage layer** of a node implements a multiversion store with operations to read specific versions and write new versions of entries. Entries are partitioned among storage nodes based on their key. A node is the owner of all entries assigned to it as a result of the partitioning scheme. We use consistent hashing in our prototype as the partitioning scheme, but other schemes could be used. Consistent hashing has the advantage of reducing the number of entries that must be moved between nodes as a result of a membership change (i.e., adding or removing a node) [12]. In addition to its assigned entries, storage nodes can cache any other entries, as long as space is available. An entry is typically cached after it is accessed remotely. Cached entries allow a storage node to exploit access locality, and thereby improve performance. Volatile storage nodes are a particular instance of this design in that they cache entries without taking ownership of them.

The **transaction layer** is responsible for transaction execution. For each new transaction, it creates a *transaction record*, containing the transaction's unique identifier, readset, writeset, and snapshot timestamp. The transactional layer exposes a BerkeleyDB-like key-value interface [22] supporting the following core operations: *transaction_get*(k), *transaction_put*(k, v), *transaction_commit*() and *transaction_rollback*().

When a transaction t executes its first *transaction_get* operation, t's snapshot timestamp is assigned the current value of the node's *transaction counter*. All further *transaction_get* operations of t will be consistent with t's snapshot timestamp. This guarantees that read-only transactions always see a consistent view of the database and do not need to be certified. The transaction counter of a node is incremented each time a transaction is locally committed.

The **B+Tree layer** has support for variable key and value sizes, multi-part table keys (i.e., keys composed of multiple fields), 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, as with a traditional implementation, we have calls to the *transaction_get()*, *transaction_put()* and *transaction_commit()* APIs provided by our transaction layer. Our B+Tree supports the typical B+Tree operations, including *search(k)*, *insert(k,v)*, *update(k,v)* and *delete(k)*.

In order to make it possible to index arbitrary types of data with different collation strategies, our B+Tree layer 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.

New B+Tree node structures are assigned a unique key and persisted in our storage layer using the MessagePack serialization library.[1] Optionally, B+Tree nodes can be compressed after serialization. Multiple B+Trees can be stored within the same storage node, and multiple active client sessions are supported. These features enable the use of a single storage node to store multiple indexes on multiple tables with the API enabling seamless switching among them.

2.4 Certifier

The certifier has three main tasks: (a) checking whether the items read by a committing update transaction are up-to-date when the transaction requests to commit; (b) logging committed transactions and system events (e.g., membership events) to disk; and (c) propagating new entries created by committed transactions to all nodes. The certifier is replicated for fault-tolerance using state-machine replication [17, 26] and Paxos [18, 21].

To certify a transaction t, the certifier checks whether the readset of t intersects the writeset of transactions that committed after t executed its first *transaction_get()* operation. We use Bloom filters to efficiently implement the intersection between two sets. The intersection of two Bloom filters is empty if the result of and-ing their bitmaps is zero. Bloom filters offer two advantages: (1) finding their intersection is linear in the size of their bitmap; and (2) the information stored in Bloom filters is compressed, thus reducing memory requirements. However, Bloom filters have the disadvantage of false positives, resulting in a tunable number of transactions unnecessarily aborted.

Besides storing the state needed for transaction certification, each certifier replica also stores the log of committed transactions, which allows a storage node to recover from a certifier replica. The replicas store the log on stable storage. To avoid scanning a large portion of the log to find the last version of a key, certifier replicas maintain hints to the location of the last version of a key in the log.

The certifier also helps nodes to be added online. To join a running MoSQL instance, a node announces itself to the certifier. When the certifier receives this request it forwards it to all nodes. This mechanism

[1] http://msgpack.org/

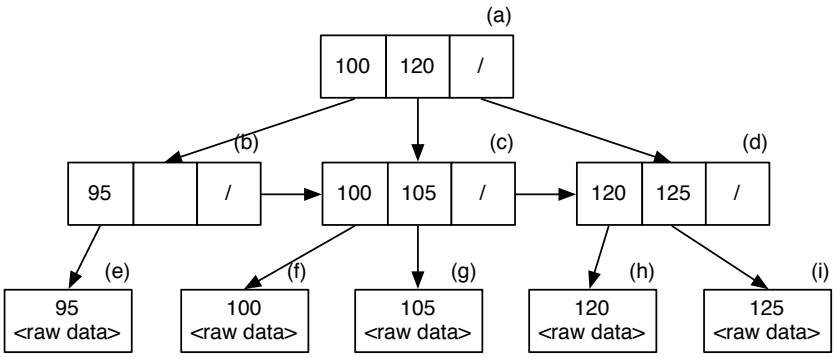

Figure 2: B+Tree example for `employee` relation.

globally serializes the membership change request with other system events.

2.5 Storage engine

MoSQL effectively turns MySQL into a middleware for interpretation and execution of SQL statements. Existing applications need not make any significant changes in order to use MoSQL, they only need to make use of a client connection pool software to distribute connections to available MySQL servers, or "MySQL instances" in our parlance.

MoSQL implements five core interface methods corresponding to the typical database operations for persisting, scanning, deleting and retrieving data in a table: *index_read()*, *index_next()*, *write_row()*, *update_row()* and *delete_row()*.[2] Parsing and optimization of SQL statements are handled by higher layers within MySQL and are not part of the scope of our implementation (see Section 4). Depending on the workload and characteristics of the row data, the storage engine is capable of storing the packed representation of the row data inside the B+Tree node itself, or storing the primary key value in the primary key index and using this value to retrieve the row data from the storage layer through a transaction_get() operation. Our storage layer currently imposes approximately a 100-byte overhead per key stored. For tables with "skinny" rows, this can result in a substantial overhead, and it is far more sensible to store both primary key and row data inside the B+Tree node itself. Conversely, for tables with wide rows, storage of the row data inside the B+Tree node will greatly increase the size of the nodes and potentially waste much bandwidth when transferring B+Tree nodes.

Secondary indexes are also supported with the secondary key stored in the key part of the B+Tree, and a primary key pointer stored in the value part that can then be used to retrieve the pointed-to row data from the storage layer.

3 Isolation considerations

In MoSQL, serializable transaction isolation is provided by the transaction layer, the B+Tree layer, and the certifier combined. Dirty reads and unrepeatable reads are avoided by the transaction layer: read requests return committed items which correspond to a consistent database snapshot. Lost updates are checked during certification, using transaction readsets and writesets, collected during the execution of transactions by the transaction layer. Phantoms [16] are checked during certification with information about the B+Tree, as we now explain.

For our discussion to follow, we consider operations done against a relation `employee` with attributes `id` and `name`. Figure 2 shows an example of the primary key index for this table. Notice that table entries are not stored within the B+Tree structure in this example. Each square corresponds to a key-value pair stored in our storage layer.

Suppose transactions t_1 and t_2, as defined in Figure 3, execute concurrently. Both see a consistent view of the database as of their respective start times, each being assigned a snapshot timestamp when they perform their first transaction_get() operation. Transaction t_1 and t_2 both read table entries (e) and (f) (see Figure 2)

[2] http://forge.mysql.com/wiki/MySQL_Internals_Custom_Engine

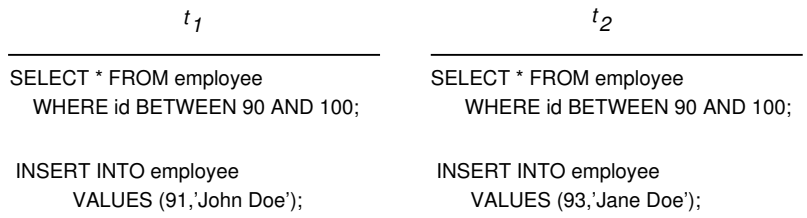

Figure 3: MoSQL phantom anomaly involving transactions t_1 and t_2.

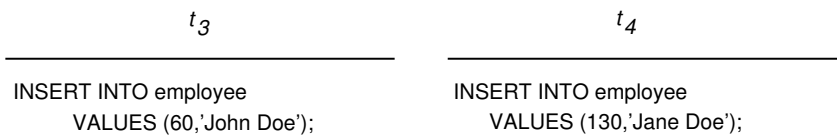

Figure 4: Concurrent execution of t_3 and t_4 may result in unnecessary aborts.

and each transaction adds a new table entry, with key 91 and 93, respectively. As a result, if during certification the certifier considers only table entries to detect conflicts, both transactions will commit, despite the fact that the execution is not serializable (i.e., in a serializable execution, the second transaction must see the value insert by the first transaction). MoSQL avoids this problem because B+Tree entries are also used in certification. Entries (a), (b), and (c) will be in the readset of t_1 and t_2, and the insert operations of t_1 and t_2 both change entry (b), causing it to be in the writeset of both transactions. As a result, t_1 and t_2 will conflict on (b) and only one transaction will pass certification.

Our approach to preventing phantoms resembles the idea of granular locks [16], applied to certification of B+Tree entries, and as such, it is subject to similar tradeoffs. In particular, it results in some concurrent non-conflicting transactions being aborted, depending on the B+Tree node size, the specific arrangement of keys, and the execution order. For example, consider transactions t_3 and t_4 in Figure 4. To insert table entry with key 60, t_3 reads entries (a) and (b) and writes (b); t_4 reads and writes entries (a) and (d) since these B+Tree nodes will be split. Thus, if t_3 is certified after t_4, then it will be aborted, even though the two transactions do not access any common table entries (although they access common B+Tree entries).

4 Performance considerations

MoSQL scales throughput by clustering multiple storage nodes under a single MySQL node or multiple MySQL nodes. As we discuss in this section, having multiple instances of MySQL sharing a common underlying storage has important implications on performance optimization. In brief, this stems from the fact that while distributed indexing and storage are provided by the lower layers of our architecture, an unmodified MySQL optimizer is still used to do query planning and optimization, and was never intended to be used in settings with multiple interdependent instances.

One of the key advantages of a general-purpose RDBMS is the flexibility and expressive power provided by SQL and the heuristics used by the optimizer to ensure that queries are efficiently executed. In order to work well, however, the optimizer requires statistics that reflect the state of the system.

We illustrate the implications of MoSQL's design with two statistics used by the MySQL optimizer that must be provided by the storage layer: the estimated *cardinality* of the table (i.e., number of rows or records) and the *records_in_range(p_1, p_2)* statistic, a storage engine interface method that returns the estimated number of records in the storage layer between predicates p_1 and p_2.

In our current implementation we do not collect any statistics about the distribution of keys. We use a naive, local estimate for the cardinality, based on the number of locally attempted insert statements (deletes and inserts which eventually abort do not cause this value to decrease); optionally we can scale this value to the number of nodes in the system. When a node has no information about a table, a constant value is assumed (in our experiments, this constant is 5000).

In order to provide an estimate for the *records_in_range(p_1, p_2)* statistic, we have assumed a uniform distribution of rows among the keys and exponentially fewer rows estimated for predicates involving a greater number of parts of the index, i.e., for a table with primary key $(k_1, k_2, ..., k_m)$ we assume that the number of

rows matching a predicate range using the first i parts of the index is proportional to C^{m-i}, where C is an appropriate constant (10 for our experiments). We have found this naive approach to be sufficient for the relatively simple TPC-C queries and also to result in good query plans for most of our complex validation queries operating against the TPC-C schema.

Moreover, with typical OLTP workloads involving a set of predefined queries that are prepared, potential optimizer mistakes due to inaccurate or incomplete statistics can be mitigated through overrides: in our modified TPC-C client we make extensive use of the FORCE INDEX extension to ensure an optimal access plan is used. We note that a similar issue affects MySQL Cluster.[3]

Finally, notice that MoSQL is less subject than traditional databases to the potential problems of poor access plans since all data is stored in main memory. In order to create an appropriate query plan, a traditional database optimizer must take into account, among other things, the speed of random index lookups, sequential disk read speed, and index scan speed, since traditional databases must take great care to avoid expensive disk accesses. With all data stored in-memory in MoSQL and accessed by key, all accesses effectively become "random" accesses, with the only important difference between keys that are cached local to a node and those which must be retrieved remotely from another storage node.

5 Performance evaluation

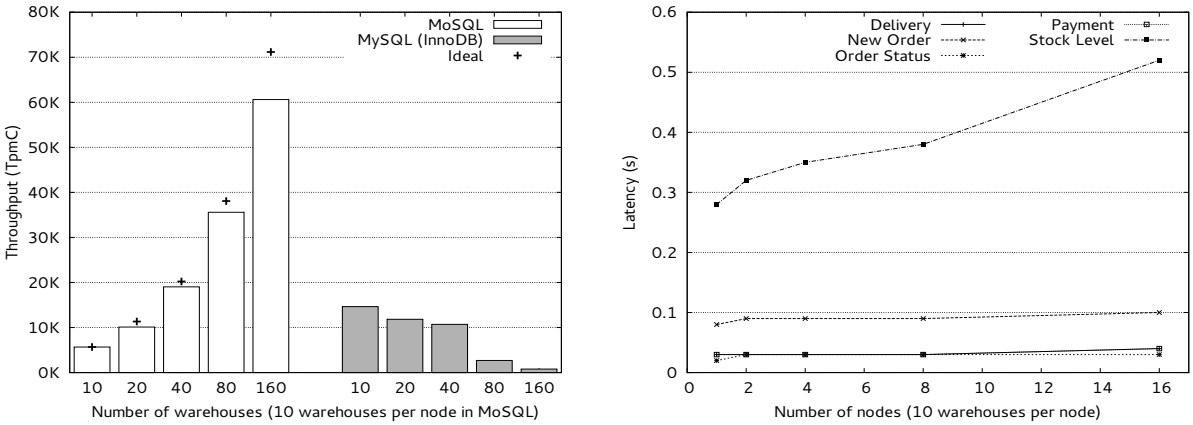

Figure 5: TPC-C throughput (L) for MySQL+InnoDB (1 server) and MoSQL (1-16 nodes) and latency (R) for different transaction types in MoSQL.

We have evaluated MoSQL on the TPC-C benchmark and a microbenchmark, used to stress particular parts of the system. All tests were done on a cluster of servers with 8GB of RAM, two quad-core, 2.50GHz Intel Xeon L5420 CPUs and HP 500GB 7200 RPM SATA HDDs. Unless otherwise noted, we have colocated on each physical server a single mysqld instance and two storage nodes.

Client connections to MySQL are assigned to underlying storage nodes in a round robin fashion. For experiments involving InnoDB, we have configured it with a 2GB buffer pool, serializable transaction isolation and to flush transactions to disk on commit.[4] For tests involving MySQL Cluster, we used a single SQL node and data node per server, with mostly default configuration options.[5] We have used MySQL 5.1.56 as our base version for the MoSQL storage engine and for tests involving InnoDB. For MySQL Cluster we have used version 7.2.5 (5.5.20-ndb-7.2.5).

5.1 TPC-C benchmark

For our TPC-C experiments, we have loaded 10 standard warehouses per physical node. Our TPC-C client is a modified version of the MySQL TPC-C benchmark tool made available by Percona,[6] which itself is based

[3] MySQL recommends the use of optimizer overrides in cases where the lack of an accurate `records_in_range` statistic is an issue: http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-limitations-performance.html

[4] innodb_flush_log_at_trx_commit = 1, innodb_buffer_pool_size = 2G and transaction_isolation = SERIALIZABLE

[5] 2 replicas, 800M/380M data/index memory

[6] https://code.launchpad.net/~percona-dev/

on the reference C-based implementation provided in the TPC-C specification. We have modified most of the SQL statements to use the `FORCE INDEX` statement to override optimizer decisions about what index to use, and configured each client thread to be "sticky" to a particular warehouse or range of warehouses for the duration of its execution, so as to improve cache performance on the underlying data node.

From the nine tables in the TPC-C schema, we configured the `item, order_line, orders, new_order` and `history` tables to store rows inside the B+Tree leaf node; the `warehouse, district, customer, stock` tables were configured to store rows in a separate key-value in our storage layer. None of the tables were configured with compression.

In Figure 5 we show peak throughput for MoSQL from one to sixteen physical servers; we also show the points representing ideal scalability and the performance of InnoDB on a single server with increasingly large warehouses. Due to the nature of the transactions in the TPC-C benchmark, and the increasing size of the database to sixteen nodes, MoSQL does not achieve perfect scalability. The reason for this is primarily due to the reduced effectiveness of the cache as nodes are added: as the database grows in size, and the share of keys that a single storage node owns decreases, there is a greater likelihood that keys must be retrieved remotely.

We also include for reference the peak performance of InnoDB with the same sized databases on a single server. As expected, performance degrades rapidly as the size of the database surpasses the configured amount of memory for the buffer pool, increasing the chance that row retrievals happen from disk. We note that the threshold for where MoSQL overtakes the throughput of single-server InnoDB is at approximately 2 nodes (i.e., 20 warehouses). While memory could be added to the single system to maintain performance, at some point, as the size of the database grows beyond available memory, the benefit of an elastic system such as MoSQL is clear.

In Figure 6 we show two example runs of the TPC-C benchmark, one against an 80-warehouse, 8-node system, and another against a 40-warehouse, 4-node system, to illustrate the problem of cache performance on increasingly large systems. We show New-Order transaction throughput and the total number of remote requests performed, system wide, over time, on a cold-started system (i.e., freshly loaded from a database backup, no keys cached). As the systems run, there is a rapid improvement in throughput as keys are cached locally, but note that the eight-node system remains consistently under double the throughput of the four-node system, and that similarly the ratio of remote requests between the eight and four node configurations stays slightly above two.

In Figure 5, the 90th-percentile latency of the Stock-Level transaction from one to sixteen nodes is most reflective of this problem; the Stock-Level transaction is a heavy, read-only transaction that can potentially read several hundred rows from the large `order_line` table. As the number of remote calls increases relative to the size of the system, this transaction shows the greatest degradation in latency.

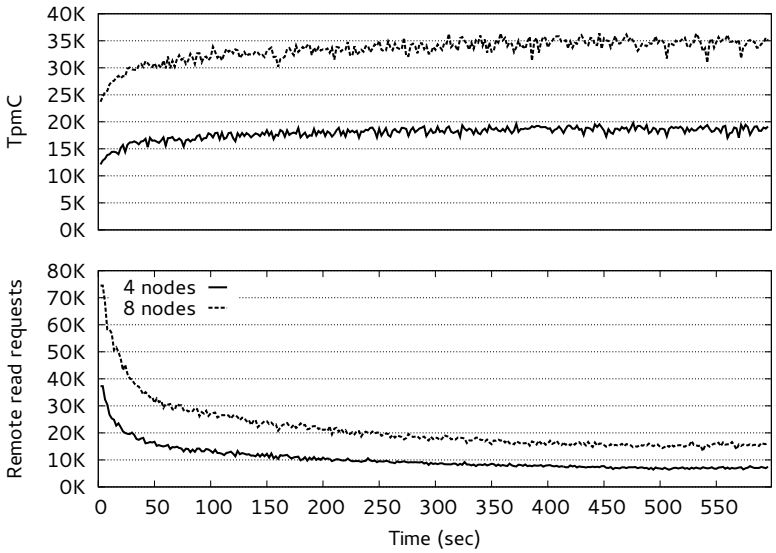

Figure 6: Remote reads and New-Order throughput over time for 4- and 8-node configurations.

5.2 Microbenchmark

We have also evaluated MoSQL with microbenchmarks against a simple table with an integer primary key and single text field. The table has been configured uncompressed and with row storage inside the B+Tree leaf node. All operations were done by assigning each benchmark thread a range of keys and randomly choosing from that range and applying either a select, insert or update statement, writing a 10-byte string along with the primary key value. All tests were done with a table of 3.2M rows and each statement done as a separate transaction. The size of the database was held constant in these tests to help eliminate the effect of the cache on performance (i.e., the database size can easily fit into the cache of a single node).

As can be seen in Figure 7, updates and selects both show near perfect scalability to 16 nodes. Inserts show slight performance degradation on a per-node basis due to the slight increase in the number of transaction aborts per node as nodes are added. Figure 8 shows the corresponding latency for the throughput values. As a reference, we also show the performance of MySQL Cluster; we note, however, that MySQL Cluster offers weaker isolation (See Section 6).

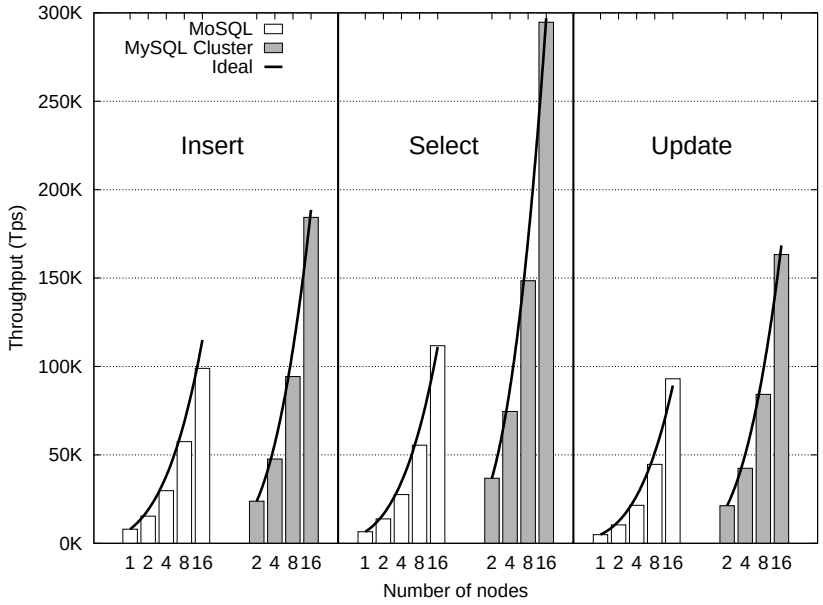

Figure 7: Microbenchmark throughput for MoSQL and MySQL Cluster.

For the update workloads at 16 nodes, due to the high commit frequency, the single certifier process begins to approach saturation and is a natural bottleneck with our current architecture. Scaling the system to configurations with more nodes would require partitioning the work of the certifier, as discussed in [28].

5.3 Online node addition

Depending on the workload, MoSQL is capable of providing improved performance by the addition of volatile storage nodes to a running system. This effectively allows the system to use more CPU for transaction processing and additional memory for caching, but without changing key ownership at the storage node level or increasing the overall capacity of the system, making them ideal for responding to temporary spikes in client load.

To illustrate, we have run an example scenario: we load a 60-warehouse TPC-C system into a four-node MoSQL configuration and steadily increase client load by eight clients every twelve seconds. We then add a fifth and then sixth MySQL server at times 72 and 108 and redistribute the client load. Figure 9 shows the run in its entirety, along with the same workload against a static four-node baseline. Throughput reaches a peak of about 18000 TpmC at time 50 and then latency begins increasing more rapidly. Throughput and latency begin to diverge from the static system in the shaded area corresponding to where the fifth node is added, with the augmented system reaching a peak throughput of 24500 TpmC after the addition of a sixth node, while the static system never surpasses 19500, a 25% improvement in throughput in less than 60 seconds.

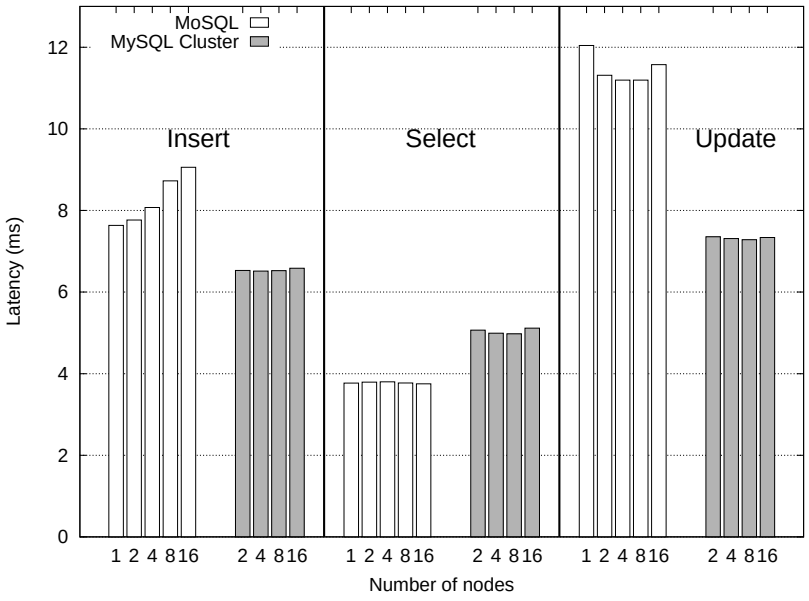

Figure 8: Microbenchmark latency for MoSQL and MySQL Cluster.

6 Related Work

The scaling problems of data management architectures have been studied for quite some time, while the desire for elasticity is a relatively recent phenomena with the increasing unpredictability and variability of workloads in internet-connected applications. In this section, we first focus on the scalability of data management systems in general and then on solutions specific to MySQL. We conclude with a short discussion on elasticity.

6.1 Database scalability

Many scalable storage and transactional systems have been proposed recently, several of which expose applications to a simple interface based on read and write operations (i.e., “key-value" stores). Some storage systems (e.g., [6, 12, 36]) guarantee *eventual consistency*, where operations are never aborted but isolation is not guaranteed. Some storage systems such as Dynamo [12] and Cassandra[7] guarantee *eventual consistency*, where operations are never aborted but isolation is not guaranteed. Eventual consistency allows replicas to diverge in the case of network partitions, with the advantage that the system is always available. However, clients are exposed to conflicts and reconciliation must be handled at the application level.

COPS [19] is a wide-area storage system that ensures a stronger version of causal consistency, which in addition to ordering causally related write operations also orders writes on the same data items. Walter [31] offers an isolation property called Parallel Snapshot Isolation (PSI) for databases replicated across multiple data centers. PSI guarantees snapshot isolation and total order of updates within a site, but only causal ordering across data centers. Differently from previous works, Sinfonia [1] offers stronger guarantees by means of minitransactions on unstructured data. MoSQL differs from these systems in that it implements both a rich interface (i.e., SQL) and provides stronger guarantees (i.e., serializability).

Google’s Bigtable [8] and Yahoo’s Pnuts [9] are distributed databases that offer a simple relational model (e.g., no joins). Bigtable supports very large tables and copes with workloads that range from throughput-oriented batch processing to latency-sensitive applications. Pnuts provides a richer relational model than Bigtable: it supports high-level constructs such as range queries with predicates, secondary indexes, materialized views, and the ability to create multiple tables. However, neither of these databases offer full transactional support. Spanner [10], Google’s successor to BigTable [8], offers a semi-relational model with wide-area transaction support, but relies on an assumption of globally-meaningful timestamps provided through spe-

[7] http://cassandra.apache.org/

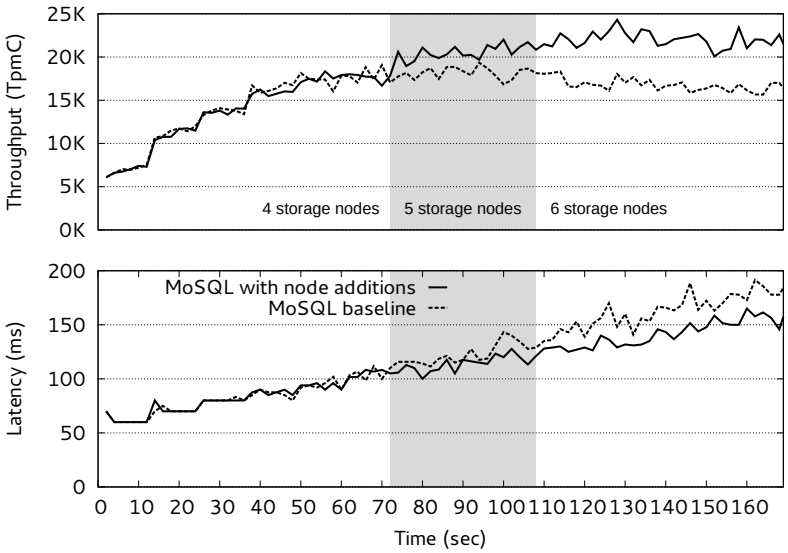

Figure 9: Adding two storage nodes online.

cialized hardware. Google's Megastore [3] similarly provides a relational model and wide-area transaction support, but with low latency only within small partitions; cross-partition transactions use an expensive two-phase commit protocol.

With multicore architectures now the norm, considerable research has gone into improving single-server performance. Multimed [33] treats a multicore system as a distributed system and runs parallel database instances mediated through replication middleware. DORA [23] proposes a coupling of threads to disjoint sets of data rather than transactions in order to reduce locking overhead.

Several fault-tolerant database protocols have exploited partial replication to improve performance. One class of such protocols requires transactions to be atomically broadcast to all participants. When delivering a transaction, a server may discard those transactions that do not read or write items that are replicated locally (e.g., [24, 29, 30]). Alternatively, some protocols implement partial database replication using either atomic multicast primitives (e.g., [14, 25]) or a two-phase commit-like protocol to terminate transactions [28]. Differently from MoSQL, all these systems expose a simple read-write interface to applications.

6.2 MySQL scalability

Approaches to scaling MySQL can be roughly divided in (a) asynchronous and "semi-synchronous" replication, (b) sharding, (c) middleware resting between MySQL and the client, and (d) middleware resting between a low-level storage mechanism and MySQL.

Asynchronous, statement-based replication has been supported by early versions of MySQL and was among the most important features leading to its extensive use in web-based applications. In a basic configuration, a master server writes committed SQL statements to a replication log that are sent asynchronously to one or more slave servers which then apply each SQL statements as is. The approach scales well for heavy read workloads where synchrony requirements are not strict, characteristic of many early web applications. More recently, MySQL has enabled row-based asynchronous replication that sends row change states to slaves rather than statements. An extension to MySQL called "semi-synchronous" replication shipped with core MySQL from version 5.5. It enabled a master server to wait until the receipt of a notification from at least one slave machine that the event was received (not necessarily committed locally) before proceeding, providing improved data integrity.

Horizontal partitioning and sharding are the common approaches to scaling write throughput in MySQL applications. Local horizontal partitions have been supported by core MySQL since version 5.1, while "sharding" (or the horizontal partitioning of tables into separate databases) has become a well-known approach for large-scale applications. Many of the problems with sharding stem from its demotion of a MySQL server to

a component of a larger system of which it has no awareness, resulting in pushing up significant complexity from the RDBMS into the application layer. Hibernate Shards[8] is one example of middleware that insulates application developers from this burden.

Sprint [5] and ScaleBase[9] are examples of middleware that reside between the application and "demoted" RDBMS nodes and manage transactions and the distribution of data across nodes. In both systems, the middleware intercepts SQL commands issued by the applications, and parses and decomposes them into sub-SQL requests, which are submitted to the appropriate databases. Results are collected and merged, before being returned to the application. MoSQL does not translate SQL requests into sub-SQL requests; it relies on MySQL's parser to handle this task.

MySQL supports a storage-layer interface with which upper layers of the database engine interact. This enables, in principle, any storage strategy to be used transparently. InnoDB is the default transactional storage engine for MySQL and is used in most cases, but storage layers targeting efficient archival, compressed storage, in-memory storage, and network-based storage exist depending on application needs. MySQL Cluster provides a scalable, low-latency and shared-nothing storage engine for MySQL. It offers high performance but with weak transaction isolation (READ COMMITTED). Additional data nodes can be added online, but not in a seamless fashion and manual intervention is required to reconfigure and rebalance data. Our approach also falls into this category.

6.3 Elasticity

With the advent of large, distributed storage layers running on typically commodity hardware, the requirement of up-front investment in infrastructure to support peaks in throughput has received more attention. This has resulted in a focus on the idea of database "elasticity", the ability of a system to contract and expand as needed in order to respond to peaks and troughs in activity.

ecStore [35] is a peer-to-peer elastic storage system providing range-query and transaction support. It uses an efficient load-adaptive replication scheme to load-balance requests for frequently accessed data, but in an eventually consistent manner. ElasTraS [11] provides transactional multi-key access to data. The ElasTraS system is composed of multiple transaction managers on top of a fault-tolerant distributed storage (such as Amazon S3). Owning transaction managers (OTM) are assigned to a partition of the data stored in the distributed storage, to which it is granted exclusive access. To ensure ACID transactional guarantees, ElasTraS uses minitransactions [1] to handle cross partition transactions. ElasTraS provides also Higher Level Transaction Managers (HTM) responsible for caching data and absorbing load of read-only transactions. Like MoSQL, ElaTraS provides the ability to add and remove servers depending on the load. Unlike the above systems, MoSQL is a storage engine for MySQL executing full–fledged SQL transactions.

7 Final remarks

We propose MoSQL, a distributed and fault-tolerant storage engine for MySQL. MoSQL preserves ACID properties while offering good performance tradeoffs: despite being distributed, MoSQL reaches similar performance compared to InnoDB, when deployed on two servers, while offering linear or sublinear scalability with additional storage nodes. We also demonstrate the elasticity of MoSQL, its ability to add storage nodes online, immediately contributing throughput while lowering overall system latency.

Acknowledgements

This work was supported in part by the Swiss National Science Foundation under grant number 127352 and the Hasler Foundation under grant number 2316.

References

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):174, 2007.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[8] http://shards.hibernate.org/
[9] http://www.scalebase.com

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. *ACM SIGOPS Operating Systems Review*, 41(3):385–398, 2007.

[6] http://cassandra.apache.org/.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2):1–26, 2008.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[10] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. *OSDI*, 2012.

[11] S. Das, D. Agrawal, and A. Abbadi. Elastras: An elastic transactional data store in the cloud. *HotCloud, USENIX*, 2009.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[14] J. Fritzke, U. and P. Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *ICDCS*, 2001.

[15] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. SIGMOD, 1996.

[16] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[18] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. SOSP, 2011.

[20] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[21] P. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. DSN, 2010.

[22] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. USENIX, 1999.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. of the VLDB Endow.*, 3(1-2):928–939, 2010.

[24] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *OPODIS*, 2006.

[25] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS*, 2010.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[27] D. Sciascia and F. Pedone. Ram-dur: In-memory deferred update replication. In *SRDS*, 2012.

[28] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *DSN*, 2012.

[29] D. Serrano, M. Patino-Martinez, R. Jiménez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*, 2007.

[30] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *NCA*, 2001.

[31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[32] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). VLDB, 2007.

[33] T. Subasu and J. Alonso. Database engines on multicores, why parallelize when you can distribute? *Eurosys*, 2011.

[34] A. Tomic. MoSQL: A Relational Database Using NoSQL Technology. Master's thesis, USI, 2011.

[35] H. Vo, C. Chen, and B. Ooi. Towards elastic transactional cloud storage with range query support. *Proc. of the VLDB Endow.*, 3(1-2):506–514, 2010.

[36] http://project-voldemort.com/.