Rubato DB: a Highly Scalable Staged Grid Database System for OLTP and Big Data Applications

Li-Yan Yuan
Department of Computing Science
University of Alberta
yuan@cs.ualberta.ca

Lengdong Wu
Department of Computing Science
University of Alberta
lengdong@cs.ualberta.ca

Jia-Huai You
Department of Computing Science
University of Alberta
you@cs.ualberta.ca

Yan Chi
Shanghai Shifang Software
chiyan@rubatodb.com

ABSTRACT

This paper proposes a new formula protocol for distributed concurrency control, and specifies a staged grid architecture for highly scalable database management systems. The paper also describes novel implementation techniques of Rubato DB based on the proposed protocol and architecture. We have conducted extensive experiments which clearly show that Rubato DB is highly scalable with efficient performance under both TPC-C and YCSB benchmarks. Our paper verifies that the formula protocol and the staged grid architecture provide a satisfactory solution to one of the important challenges in the database systems: to develop a highly scalable database management system that supports various consistency levels from ACID to BASE.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Transaction processing*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

Keywords

Scalable Architecture; Concurrency Control; ACID

1. INTRODUCTION

Big data applications demand and consequently lead to developments of various highly scalable data management systems. The trend of using lower-end, commodity servers to scale out configurations has also become popular, due to the drop in hardware prices and the improvement in performance and reliability. Meanwhile, many large-scale data management systems such as key-value stores [14] and Bigtable-like stores [8] represent a recent evolution in building infrastructure by making trade-off between scalability and consistency. That is, highly scalable systems often renounce ACID and turn to BASE for high scalability [10, 14, 19]. However, BASE properties make only liveness guarantee rather than safety guarantee, and thus applications managing critical data cannot depend on BASE for correctness and security.

The trade-off between achieving scalability and preserving strong consistency faced by existing databases leads us to consider the following questions:

- **Is it possible to achieve high scalability with the ACID properties?**
- **Is it possible to scale out the commonly used single server database system design?**

In this paper, we provide positive answers to both questions by presenting Rubato DB, a NewSQL [24, 25] database management system that supports ACID properties.

Rubato DB is designed and implemented with two distinguishing features:

- **F1**: Rubato DB applies a novel grid database architecture based on the staged event-driven architecture [31].
- **F2**: Rubato DB introduces a new formula protocol for distributed concurrency control.

One of the main challenges in designing and developing a highly scalable database management system using a collection (cluster) of commodity servers is how to distribute a large volume of data into tens or hundreds of smaller dedicated DAS devices attached to commodity servers. It is **F1** that enables Rubato DB to resolve this problem.

Another challenge is how to develop an effective concurrency control protocol for thousands of concurrent users that are accessing data distributed over commodity servers. We have developed the formula protocol for concurrency (**F2**) under the staged grid architecture, which guarantees the serializability of transactions processed by Rubato DB.

The formula protocol for concurrency is a variation of the Multiversion Timestamp Concurrency Control Protocol [26], which reduces the overhead of conventional implementation by the following three optimization mechanisms:

- Instead of using multiple versions of updated data items[1], simple formulas are stored in the memory associated with each updated data item.

[1]We use data item as a general term; all definitions and algorithms in this paper go through for any granularity of data substituted for this term.

CIKM'14, November 03–07, 2014, Shanghai, China.
Copyright 2014 ACM 978-1-4503-2598-1/14/11 $15.00.
http://dx.doi.org/10.1145/2661829.2661879.

- A dynamic timestamp ordering is used to increase the concurrency and reduce unnecessary blocking.
- Caching and delaying the transaction operations within formulas before committing.

We carried out comprehensive experiments to evaluate the performance of Rubato DB under the TPC-C and YCSB benchmarks. Especially, the performance of Rubato DB under TPC-C benchmark test can achieve 363K tpmC (with 325,000 concurrent clients) running on a collection of 16 commodity servers (as demonstrated in Figure 6(f)). Rubato DB strictly complies with the TPC-C specification. Particularly, remote guest access is fully supported and no simplifed mechasims such as stored procedures are used to enhance the performance.

The main contributions of this paper are:

- We propose the formula protocol for concurrency in the distributed environment ensuring serializability.
- We define a highly scalable staged grid database architecture, and outline its implementation techniques, based on which Rubato DB is implemented.
- We describe the main technologies and novel features used in the implementation of Rubato DB.
- We conducted extensive experiments focused on the performance of Rubato DB, which fully testify that Rubato DB can support both ACID and BASE properties with scalability.
- Our experiments not just confirm our solution but also provides much needed insights towards highly scalable database implementations.

In Section 2, we review the background and related work. We present the staged grid architecture and formula protocol in Section 3, 4. The experiments are reported in Section 5 and we conclude the paper in Section 6.

2. BACKGROUND AND RELATED WORK

In the last decade, the importance of shared-nothing architecture was enhanced in the design of web services. One representative design is the well-known MapReduce framework for processing large data sets [13]. Another design is the Staged Event-Driven Architecture (SEDA) [31]. The SEDA decomposes the whole execution into a series of stages that represent a set of individual tasks. [31].

There has been some recent work on bringing ideas from MapReduce/SEDA to database management systems. The SEDA has been applied to improve the staged database performance through exploiting locality in the memory hierarchy at the hardware level [16]. Hive [28] and Scope [7] integrate query compiling into the MapReduce framework at the query language level. At the system level, Greenplum [9] and HadoopDB [1] systematically leverage technologies to enable queries across multiple nodes to be executed in the MapReduce style. Dremel [22] uses a multi-level serving tree to execute queries. Each query is rewritten at each level and gets pushed down to the next level in the tree. Dryad [17] constructs the data flow into a direct acyclic graph with communication channels. Rubato DB adopts the similar idea by integrating the staged event-driven architecture into the grid shared-nothing infrastructure.

One common implementation to provide serializability in systems, such as Megastore [4], Spinnaker [23], etc., is based on distributed two-phase locking and two-phase commit protocol. To ensure serializability, all of a transaction's locks must be held for the entire duration of the two-phase commit. However, since multiple network round-trips are needed for the commit, the extra time that locks are held can considerably reduce the overall transactional efficiency, and the distributed locks that are held by the incomplete transations of a failed client prevent others from making progress [27]. As well, the deadlock detection and resolution may further prohibit the concurrency and scalability [20].

Some systems (Sinfonia [2] and H-Store [18], etc.) attempt to reduce the locking overhead by constraining the scope of transactions whose accesses are limited to a small subset of the database. Transactions that can be executed in parallel to completion without requiring communication with others are optimized, but other transactions are aborted or executed with coarse-grained locks on each partition. Another simplified variation is to write transactions as stored procedures [25, 30] without concurrency control. Stored procedure invocation is executed sequentially within different partitions to take advantage of parallelism. Despite the high performance and scalability induced by restricted transactional scope, the concurrency and applicability can be hindered for generalized workloads [4, 18, 27].

Calvin [27] introduced deterministic ordered locking. By using a pre-ordered agreement for acquiring locks in the presence of distributed transactions, distributed commit protocols can be eschewed. However, extra efforts are required to analyze advance knowledge of all transactions' read/write sets before execution. Instead of using any locking mechanisms, Rubato DB implements similar protocols to Spanner [12], that is, combining the timestamp-based concurrency control with two-phase commit.

Snapshot Isolation is also an attractive isolation level. However, reading from a snapshot means that a transaction never sees the partial results of other concurrent transactions and write skew may occur as well. Similar to the recent work [15] on snapshot isolation where transactional dependencies are detected for serialization, concurrency control in Rubato DB deals with read/write conflicts using a list of transactional stored facts (i.e., read before, read by) per data item with operation performed.

3. STAGED GRID DATABASE

3.1 A Staged Grid Architecture

Rubato DB is implemented using a staged architecture, which has been introduced and studied for various applications, such as Dynamic Internet Servers and high-performance DBMSs [16, 31]. The basic idea of this architecture is that a system is constructed as a network of staged modules connected with explicit queues. A staged module (or stage) is a self-contained module consisting of an incoming event (request) queue, a thread pool and an event handler, as described in Figure 1 [31]. The staged architecture provides a satisfactory design for a scalable grid (or distributed) database management system, since the staged modules can be easily arranged to run on various grid servers. As a matter of fact, the MapReduce Framework [13] can be modeled as a two-staged architecture.

Now we specify the staged grid architecture by integrating the staged approach into the grid shared-nothing envi-

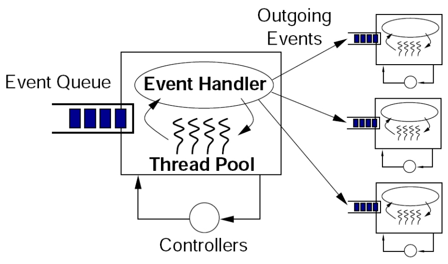


Figure 1: Description of a Staged Module

ronment. The definition facilitates easy implementation and efficient operation.

Definition 1. A *staged grid module* is a staged module that runs on a grid node. A staged grid module is *encapsulated* (*grid encapsulated module*), if it accesses only the data stored in the DAS (Direct-Attached Storage) of the node.

Definition 2. A *staged grid architecture* is a software system architecture such that:

(a) The system is constructed as a network of grid encapsulated modules on the shared-nothing infrastructure, i.e., all grid nodes are connected by a (high speed or otherwise) network.

(b) Communication between two modules is by means of the event queues through the network.

(c) When a staged module pushes a request to the next module, the request will always be accepted and kept in the event queue of the destination module.

(d) Each staged module attempts to read requests from its input event queue. The module will wait if no available request can be obtained.

(e) Each request to the system will be processed in a relay-style by passing the request from one staged grid module to the next one until it is completed.

3.2 Implementation of Staged Grid Database

Rubato DB uses the staged grid architecture for its query engine. The essential components of Rubato DB are depicted in Figure 2.

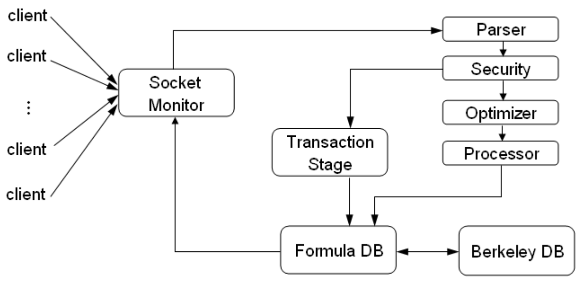


Figure 2: Rubato DB system architecture

System Deployment

Rubato DB runs on a collection of servers as one database instance with a single socket monitor stage module to establish and record the connection states for all client requests. Unlike systems such as VoltDB [30], Rubato DB provides a service as a general OLTP store in that all the transactions are received as a sequence of SQL requests, not just a stored procedure. When a request is received by the socket monitor stage, it is assigned a transaction id (i.e., timestamp).

Rubato DB's SQL engine is used to process all SQL queries. The SQL engine is built on top of the staged grid architecture where each component is implemented as a staged module. Data independent staged modules such as parser and security can be deployed on arbitrary grid nodes, and each node can even consist of multiple such stages for scaling up. Optimizer and processor stage modules (i.e. query stage, update stage) are deployed corresponding to tables per node. Particularly,

(a) If a query involves one table on one node, one query stage is set up for each table.

(b) If a query involves a join on multiple tables on one node, the join will be carried out by one join stage.

(c) If a query involves one table that distributed over different nodes without join, one query stage per node will issue the requests, and one aggregation stage in one of the nodes will collect all result sets and combine them as one result set.

(d) If a query involves a join on two or more tables distributed over different nodes, several stages are needed to perform the semi-join based query optimization (not implemented yet[2]).

(e) One update stage for each table to perform all update/insertion/deletion operations on each node.

With such deployment, different types of stages can work together for a single query in a relay-style to achieve pipelined execution. The same type of stages associated with different table partitions can also work simultaneously to achieve parallelism. A Transaction stage is deployed on each node, being responsible for coordinating data access on multiple nodes based on a novel formula protocol for concurrency (FPC) (in Section 4).

Software Instruction Set

To facilitate communication among staged modules on different nodes, a set of *software instructions* is introduced to specify all basic operations and/or data packets. Each instruction carries all necessary information required for a request or a data packet, including its *transaction id*, *operation id*, *stage id*, *partition id*, *table id*, *destination node*. Any stage will receive a sequence of incoming instructions from its previous stages, and forward the processed result (encapsulated in an instruction or a sequence of instructions) to its successive stages for further processing. The instructions with a uniform format are the only packets flowing through stages in the system.

By properly multiplexing the concurrent requests of multiple instructions, there is a potential of increasing instruction reusability to optimize the memory usage. Rubato DB utilizes a stack as a pool of instructions in each grid node: one instruction is popped up to serve a request when needed, and will be pushed back to the pool after being used.

Multiple Communication Channels among Stages

Rubato DB utilizes different communication channels among all stages, depending on locations of stages and/or the sys-

[2]So far, we can avoid such case by distributing tables with joining on the same node.

tem resources. Assume the stage S_i is going to send an instruction I into the stage S_j. Two different channels can be used for different cases:

- **I1.** S_i and S_j are in the same node. Since each node maintains one instruction stack pool, it only needs to push the address of I into the operation queue of S_j.
- **I2.** S_i is in Node N_i and S_j is in Node N_j, and N_i, N_j are on a shared-nothing infrastructure. The standard TCP/IP protocol pipes are used to send the content I from N_i into a new instruction I_j in N_j and then push the address of I_j into the operation queue of S_j.

4. CONCURRENCY CONTROL PROTOCOL

In this section, we will first present the *Formula Protocol for Concurrency* (FPC), the concurrency control protocol used in Rubato DB; and then outline our schema for the implementation of the protocol. The FPC is a novel implementation of the classical Multiversion Timestamp Concurrency Control Protocol [26] with logical transformation formula caching and dynamic timestamp ordering.

The **logical transformation formula caching** approach has the advantages over storing actual multiple versions of data items mainly in the following cases:

(a) For all non-key updates, storing multiple versions need to maintain physical copies of numerous rows [21], but the formula approach still uses one simple formula. This will significantly reduce overhead of multiple versions of all update data items and simplify the garbage collect process.

(b) Formula enables us to use commutative conflict-free operations [3] such as increment/decrement[3], instead of updates, which is much easier than if other wise, in terms of much less conflict potential.

(c) The implementation of multiversion uses the fine-grained page as the minimal unit in the memory [5, 6]. The page size can affect the latency greatly since the full bandwidth can only be achieved if data is flushed in pages of relatively large size(e.g. 4KB, 8KB) [6]. To save page space, multiple versions of different data are clustered into pages, and complex and error-prone mechanism is required to ensure update operation does not overwrite each other even though they access different data [5, 6]. The manipulation on the unit of formula can reduce the complexity of storing multiple versions on the page-level. FPC is suitable for both numeric values and string values without costs of pages.

The **dynamic timestamp ordering** is used to achieve:

(a) avoiding unnecessary blocking or waiting in order to increase the degree of concurrency.

(b) clearing the formulas as early as possible, similarly to database management systems using the multiversion timestamp protocol that delete any version of updated data item as soon as it is not needed.

In the conventional multiversion timestamp protocol, the commit order of operations conforms with the timestamp initially assigned to each transaction. This mechanism is considered to be static. However, the FPC respects the initial timestamp ordering while permitting an equivalent schedule that differs from the static timestamp ordering, as long as it ensures serializability. The timestamp ordering of all the transactions may be altered to allow a transaction with older (smaller) timestamp to read the data item updated by a later (larger) transaction on condition that the serializability is respected.

[3]The execution order of operations does not affect the result. Thus we consider such operations conflict-free even though they write the same data item.

The following example demonstrates the two distinct features of the FPC.

Example 1. Let A, B, C denote three values in one data item[4]. We use "W", "R" to denote "write" and "read" operation respectively, e.g. $W(B = B + 10)$ means update B by increasing the value of B by 10; $R(B)$ means read the value of B. Consider the schedule in Table 1 where T_{10}, T_{20}, T_{30} are three transactions with subscripts as their timestamp:

	T_{10}	T_{20}	T_{30}	**A**	**B**	**C**
t_1				90	100	80
t_2	W(B=B×1.1)			90	100	80
t_3			W(B=B+10)	90	100	80
t_4			W(C=C+10)	90	100	80
t_5			commit	90	110	90
t_6		R(B=121)				
t_7	commit			90	121	90
t_8		commit		90	121	90

Table 1: Transactions Schedule

In this schedule, the updates issued by T_{10}, T_{30} at t_2, t_3, t_4 will be executed by caching formulas rather than multiple versions of replica. When T_{30} requests to commit at t_5, all formulas issued by T_{30} (i.e., $W(B = B{+}10)$, $W(C = C{+}10)$) will be force-written and eliminated from the memory. After T_{30} commits, the disk values of B and C are updated to 110 and 90 respectively. Further, when T_{20} issues $R(B)$ at t_6, it will first read the value of B from disk as 110, and then the update formula $W(B = B * 1.1)$ issued by T_{10} earlier will be applied. Hence, T_{20} reads the value of B as 121.

This schedule that is not allowed in the classical multiversion timestamp protocol, is indeed serializable and its equivalent serial schedule is T_{30}, T_{10}, T_{20}. The schedule does not comply with the intial timestamp ordering, but it does permit an equivalent serial schedule.

Now we present the formula protocol for concurrency formally as follows.

4.1 Formula Protocol for Concurrency (FPC)

As with the timestamp protocol, each transaction under FPC is assigned a unique timestamp, $TS(T)$, when it is initiated in the socket monitor stage on one dedicated server. The FPC guarantees the existence of an equivalent serial schedule in which transactions are ordered by their timestamps (subject to dynamic timestamp ordering).

The FPC stores with each data item, x, on relevant node, the following pieces of information:

- $lrt(x, Ni)$: the largest timestamp of active (not committed) transactions that have read x on the node N_i;
- $list(x, N_i)$: the list of update formulas of the form: $uf(x, T_{u1}, N_i), \ldots, uf(x, T_{un}, N_i)$, where $uf(x, T_j, N_i)$ represents an update formula on x by transaction T_j on Node N_i, and $TS(T_{u1}) \le TS(T_{u2}) \le \cdots \le TS(T_{un})$.

[4]For simplicity, we consider only one data item but all the discussions are valid for a set of data items.

If such an active transaction does not exist, $lrt(x, Ni)$, and $list(x, N_i)$ are set to 0 and $\emptyset$ respectively.

Read/Write Operation

When a transaction, T_1, makes a request to read x on node N_i, $read(x, T_1, N_i)$ will first retrieve the value of x from disk on N_i and then update the retrieved value using stored update formulas in $list(x, N_i)$ if needed. More specifically,

- **R1.** T_1 read x on N_i.

  Let $v_0(x)$ be the disk value of x on N_i, and

  $uf(x, T_{u1}, N_i), \ldots, uf(x, T_{um}, N_i)$ be the list of update formulas in $list(x, N_i)$ such that (1) $TS(T_{um}) \leq TS(T_1)$ and (2) $TS(T_{u(m+1)}) > TS(T_1)$. Then $v_1(x)$ be the value obtained by applying $uf(x, T_{u1}, N_i)$ on $v_0(x)$, $v_2(x)$ be the value obtained by applying $uf(x, T_{u2}, N_i)$ on $v_1(x)$, and $v_{um}(x)$ be the value obtained by applying $uf(x, T_{um}, N_i)$ on $v_{u(m-1)}(x)$.

  $v_{um}(x)$ is the value to be retrieved by $read(x, T_1, N_i)$.

  If $TS(T_1) > lrt(x, N_i)$, $TS(T_1)$ is assigned to $lrt(x, N_i)$.

  To facilitate the cascading rollbacks and $wait\_for\_commits$, the FPC protocol also records a fact, $read\_by(T_{uk}, x, T_1)$ for $1 \leq k \leq m$, indicating the value x updated by T_{uk} is read by T_1.

Obviously, T_1 retrieves the value that is obtained by sequentially applying all update formulas on x issued by transactions with older timestamps, and if $TS(T_1) < swt(x, N_i)$, where $swt(x, N_i)$ is the smallest timestamp of a transaction contained in $list(x, N_i)$, it retrieves the disk value of x directly. This is the same as the multiversion timestamp protocol, except that the update formulas are used instead of actual multiple versions of data items.

When a transaction, T_1 with timestamp $TS(T_1)$, makes a request to write x on node N_i, i.e., $write(x, T_1, N_i)$, the FPC performs the following action:

- **W1.** If $TS(T_1) < lrt(x, Ni)$, there must exist another transaction T_2, which should follow T_1 according to the equivalent serial order on the timestamp, has read the value of x before. Thus T_1 is too old to write x, and must be aborted. This may also trigger cascading rollbacks as in **K1**.

- **W2.** If $lrt(x, Ni) = 0$ or $TS(T_1) = lrt(x, N_i)$, the write request is processed by simply adding a new update formula $uf(x, T_1, N_i)$ into $list(x, N_i)$.

- **W3.** If $TS(T_1) > lrt(x, N_i) > 0$, the write request is processed by (1) simply adding a new update formula $uf(x, T_1, N_i)$ into $list(x, N_i)$.

  Further, there must exist another transaction T_2 such that (a) $TS(T_2) < TS(T_1)$, and (b) T_2 has read the value of x before. The FPC protocol also (2) records a fact $read\_b4(T_2, x, T_1)$.

In **W3**, $read\_b4(T_2, x, T_1)$ is recorded to facilitate the dynamic timestamp ordering. Assume that $TS(T_2) < TS(T_1)$, and T_1 issues commit before T_2 does. The dynamic timestamp ordering allows T_1 to commit, but update formulas of T_1 will be retained in the memory rather than cleared if there exists any $read\_b4(T_2, x, T_1)$ stored in the system. Only when all the stored facts $read\_b4(T_i, x, T_1)$ for T_1 are removed as in **C2**, **K1**, update formulas of T_1 can be cleared and force-written to disks.

Commit/Rollback Protocol

To facilitate committing a transaction that has accessed data items distributed over different nodes, the FPC maintains a list $PN(T)$ of participating nodes for each active transaction T. That is,

$$PN(T) = \{N_i \mid T \text{ reads } x \text{ on } N_i \text{ or } T \text{ writes } x \text{ on } N_i\}.$$

When a transaction, T_1, makes a request to commit, the FPC performs a variation of two-phase commit protocol through a coordinator. In the 1st-phase, pre-commit(T_1,N_i) pre-checks if N_i is ready to commit for each $N_i \in PN(T_1)$. In the 2nd-phase, a consentaneous action will be taken based on the response from each node. There are two kinds of approaches to present the dynamic timestamp ordering: one is pessimistic that forces transactions to $wait\_for\_commit$ for $read\_b4$ facts; the other is optimistic that allows transactions to commit immediately in presence of $read\_b4$ facts, but retain the formulas without being cleared until all $read\_b4$ facts are removed later. The pessimistic approach is optimal for read intensive workloads where $read\_b4$ facts are rare; while the optimistic approach is optimal for write intensive workloads where $read\_b4$ facts are frequent.

- **P1. (pessimistic)** pre-commit(T_1,N_i)
  1. If there exists $read\_by(T_2, x, T_1)$ or $read\_b4(T_2, x, T_1)$ on N_i, suspend T_1 with $wait\_for\_commit(T_1, T_2)$.
  2. Otherwise, return T_1 is ready to commit on N_i.

- **P2. (optimistic)** pre-commit(T_1,N_i)
  1. If there exists $read\_by(T_2, x, T_1)$ on N_i, suspend T_1 with $wait\_for\_commit(T_1, T_2)$.
  2. Otherwise, return T_1 is ready to commit on N_i.

The FPC waits until it receives the ready-to-commit message from all nodes in $PN(T_1)$, then performs commit(T_1,N_i) on all nodes $N_i \in PN(T_1)$.

- **C1. (pessimistic)** commit(T_1,N_i), i.e., T_1 commits on the node N_i: Force-write all update formulas issued by T_1 on N_i, and remove all stored facts, such as $read\_by$, $read\_b4$, and uf involving T_1.

- **C2. (optimistic)** commit(T_1,N_i) i.e., T_1 commits on the node N_i: The formulas issued by T_1 are cleared and force-written to database disks only if there exists no $read\_b4(T_{rb}, x, T_1)$ fact; otherwise formulas are reserved in the memory. When T_{rb} commits or rollbacks and $read\_b4(T_{rb}, x, T_1)$ is the only $read\_b4$ stored fact for T_1, force-write and clear all update formulas issued by T_1 on N_i in **W3**.

After T_1 commits in all participating nodes in $PN(T_1)$, the FPC wakes up all waiting transactions T_w to resume pre-commit(T_w,N_i), if $wait\_for\_commit(T_1, T_w)$ recorded in **P1**(1) or **P2**(1).

When a transaction T_1 makes a request to rollback, or is forced to rollback as in **W1**, or times out to receive any response from N_i due to network failure, the FPC performs rollback(T_1,N_i) on all $N_i \in PN(T_1)$.

- **K1.** rollback(T_1,N_i). Send cascading rollbacks to all T_r if $read\_by(T_r, x, T_1)$ is recorded; and remove all stored facts, such as $read\_by$, $read\_b4$, and uf involving T_1 on N_i. If there exists $read\_b4(T_1, x, T_{cb})$ and it is the only $read\_b4$ stored fact for T_{cb}, force-write and clear all update formulas issued by T_{cb} on N_i in **W3**.

After T_1 rollbacks in all participating nodes in $PN(T_1)$, the FPC wakes up all waiting transactions T_w to be rolled back. if $wait\_for\_commit(T_1, T_w)$ is recorded in **P1**(1) or **P2**(1).

As indicated after **C1**, **C2** and **K1**, when T_1 terminates (commit or rollback), all transactions that are waiting for T_1 resume and restart to pre-commit or cascading rollback. The transition for transaction states is demonstrated in Figure 3.

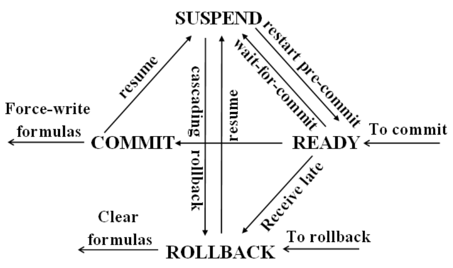


Figure 3: Transitions between states

It is not difficult to prove that the FPC guarantees serializability and atomicity since it is just a novel implementation of the multiversion timestamp protocol, except for the dynamic ordering. As a matter of fact, its proof follows the following facts:

(a) The orders of all conflict pairs of operations are compatible with the timestamp ordering of all the involving transactions.

(b) The dynamic ordering rearranges a transaction T to be effectively a smaller timestamp only if T has no operations that are conflict to any operations of T_i such that $TS(T_i) < TS(T)$.

(c) Update transactions commit corresponding to timestamp ordering. If T_1 and T_2 are update transactions that commit, then if $TS(T_1) < TS(T_2)$ (after dynamic ordering if possible), then T_1 commits before T_2.

(d) No transaction commits in a state where it has read uncommitted data. That is, transactions that issue commit will wait for any uncommitted update transactions from which they read.

(e) Any transaction that reads data written by an aborted transaction itself aborts.

(f) Availability is achieved by rolling back transactions with timeout due to node failure or network partition.

4.2 Implementation of the FPC

Rubato DB uses Berkeley DB[5] with B-tree indexes for basic data operations (e.g. *put*, *get*, *insert*, *delete*, etc.). All table partitions and their secondary index files if any are stored on the local disk as Berkeley DB files.

The basic operations of the FPC are implemented as a layer, called *Formula DB* on top of Berkeley DB. *Formula DB* is a thread-free package compatible with Berkeley DB, such that all disk accesses are through Formula DB[6].

In addition to all Berkeley DB operations, *Formula DB* also supports the following operations and functionalities:

(a) FDB→ pre-commit(T_1): to perform the pre-commit operation on FDB, as specified in **P1** and **P2**,

(b) FDB→ commit(T_1): to perform the commit operation on FDB, as specified in **C1** and **C2**.

(c) FDB→ rollback(T_1): to perform the rollback operation on FDB, as specified in **K1**.

(d) FDB→ update(T_1, N, F, W): to update the Nth column of the FDB according to the formula F for all records satisfying the boolean value expression W.

Formula DB uses (a), (b), (c) to achieve serializability over distributed nodes; and uses (d) to store logic formulas instead of multiversions of updated data items for FPC.

Formula DB implements a fine granularity of control on the manipulation of data at row level or even finer by using formula unification.

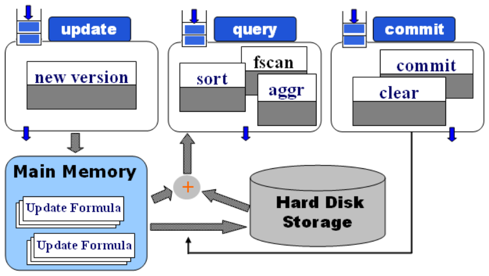


Figure 4: Query/Update engine module

As shown in Figure 4, an internal main memory Formula DB is used to store logic formulas of all the updates[7]. Each update operation is stored as one formula. Selection or read operation is evaluated against both formulas and disk. Update or write operation is done only after the transaction is committed. All associated formulas are removed from the main memory when the related transaction's state is cleared.

The implementation of *Formula DB* is decentralized in that every node contains a transaction stage to initialize the two-phase commit/rollback. The node where the two-phase commit protocol is initialized is called the coordinator node for the transactions. Other participating nodes are cohort nodes, respectively. *Formula DB* also includes measures for dealing with various kinds of failures and/or network partition that might occur.

(a) The cohort times out while waiting for a pre-commit message. The cohort will roll back the transaction. The cohort can be certain that no commit has been taken at any node, since it has not replied ready-to-commit yet.

(b) The coordinator times out while waiting for a ready-to-commit response from cohorts. Even though cohorts might have sent read-to-commit response, but one of replies might not have been delivered during the timeout period. The coordinator decides to roll back, and sends a rollback request to each achievable cohort.

(c) The cohort times out while waiting for a commit/rollback message. The cohort attempts to communicate with the coordinator or other cohorts to determine the outcome of the transaction. The cohort is blocked until it can determine if the coordinator has made a decision and if so what that decision is.

Similar to the timestamp management approach of Spanner [12] that avoids transactions from being executed with an invalid timestamp, Rubato DB adopts a loading control

[5]The transaction support of Berkeley DB itself is turned off.

[6]We set the buffer size of Formula DB same as Berkeley DB that is fine-tuned based on the data size.

[7]So far, system crashes are not considered and thus formulas are durable in the memory without being flushed to disks.

schema implemented in the socket monitor stage based on the following principles:

At any time, the system should process the requests with least conflict potential. When all the current requests have higher conflict potential, the socket monitor stage would wait awhile for new requests with lower potential to arrive.

The conflict potential is evaluated by the number of active clients and priorities among transactions. The socket monitor maintains one list of active clients (with an active transaction) and one list of requesting clients (whose requests are waiting to be processed). Assume the oldest timestamp in the requesting list is TS_R, and the number of active clients is N_A, and N_O is the number of active clients whose transaction timestamp $T_S \leq TS_R$. The conflict potential at any moment is determined by $\frac{N_O}{N_A}$. If the transaction T_{TS_R} with timestamp TS_R is the oldest among all active transactions, then the ratio is $\frac{1}{N_A}$, and processing T_{TS_R} has the least conflict potential. On the other hand, if T_{TS_R} is the youngest transaction, then the ratio is $\frac{N_A}{N_A} = 100\%$, and processing T_{TS_R} has the highest conflict potential. Rubato DB regulates the socket monitor not process any request when the oldest waiting transaction is not among the older 20%.

5. PERFORMANCE EVALUATION

In this section, we report various experiments, focusing on the performance evaluation of Rubato DB. The main purposes of experiments are:

1. What is scalability of Rubato DB for the OLTP applications requiring ACID properties?
2. Is Rubato DB capable of handling both OLTP and big data applications using a collection of commodity servers?
3. What are performance comparisons between Rubato DB and NoSQL systems in big data applications?

Since Rubato DB is developed using the proposed new formula protocol for concurrency and the staged grid database architecture, the answers to aforementioned questions also provide an assessment to the FPC and the new architecture.

We also present some interesting experiments conducted during the development of Rubato DB that provide much needed insights to FPC and the staged architecture. Particularly, answers are given to the following questions:

4. If numerous conflict operations access data items distributed over multiple nodes in an OLTP application, what is the impact on the performance of FPC?
5. In developing an application using the staged architecture, what is a better choice of using either single-thread or multiple-threads?

5.1 Experimental Setup and Benchmark

All the experiments reported in this paper use a collection of (up to 16) commodity servers connected with a Gigabit LAN with low network latency. More specifically,

1. Each server has dual quad-core Intel Xeon CPUs, 32 GB of main memory, SATA disks configured in RAID0.
2. All of the servers are running Linux Ubuntu 12.04 LTS.
3. A Rubato DB server runs on the collection of servers as one database instance.

In order to measure performance of Rubato DB and compare its performance with other systems, two standard benchmarks are used. The **TPC-C benchmark** [29] is a comprehensive database benchmark test that continues to be a popular yardstick for comparing OLTP performance on various hardware and software configurations. The **Yahoo Cloud Serving Benchmark** (YCSB) [11] is a data serving benchmark widely used to measure throughput and latency with varying operations distribution for big data systems.

5.2 Scalability of Rubato DB under TPC-C

The first set of experiments are conducted to measure scalability of Rubato DB[8] for OLTP applications using the TPC-C Benchmark. TPC-C performance is measured in tpmC (i.e., the number of New-Order transactions per minute). The benchmark can be scaled up by increasing the number of warehouses and hence the number of concurrent clients.

We firstly investigate the capacity of a single server system by scaling up the RAM size (i.e., 32 GB, 64 GB, 128 GB). We deploy all staged modules in one server. The test is conducted by increasing the number of clients from 1000 to 20000 (each warehouse has 10 clients as per the TPC-C specification). Figure 5 shows the CPUs can achieve high utilization with 128 GB RAM. Considering we only have a collection of commodity servers with 32 GB RAM, in the following tests, we strictly comply with TPC-C specification with one exception that is setting 50 clients per warehouse in order to make the best of the computing resources we have. Figure 5 shows that our results will still stand if we fully comply with TPC-C specification, i.e., 10 clients per warehouse if we had 16 servers with 128G memory each.

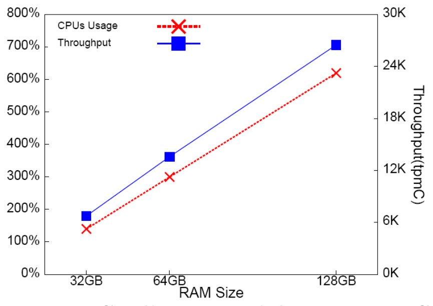


Figure 5: Scaling up with Memory Size

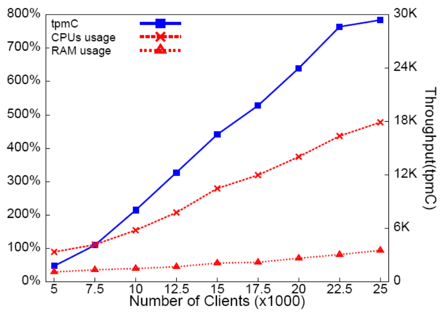
To test scalability of Rubato DB, we continue the experiments by increasing the number of concurrent clients from 25,000, at the full capacity of one server, to 47,500 using two servers, to 85,000 using four servers, and all way to 325,000 clients using 16 servers[9]. The latency of all TPC-C tests satisfies the benchmark specification, and as a matter of fact, all transactions are completed within 0.6 second. The performance of scalability is summarized in Table 2, and all details are presented in Figure 6. There are two types of nodes when the system size is greater than one. Compared with non-main nodes, the main-node also includes an additional socket monitor stage and simulates the client requests of the TPC-C test program[10].

The tests show that the system works smoothly by allocating more computing resources gradually to handle growing client base. The CPU and RAM usage percentage increases

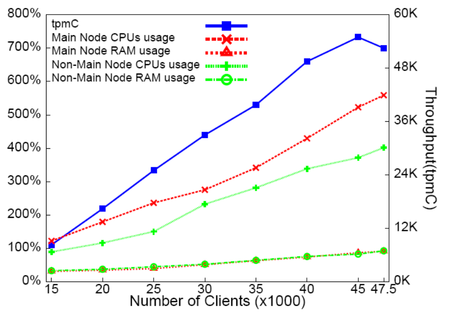
[8]By using "SET TRANSACTION ISOLATION LEVEL SERIALIZABLE", Rubato DB provides serializable transaction semantics guarantee.

[9]Rubato DB is currently implemented as a research system in the university lab, and its performance may still be improved dramatically comparing with those commercial ones.

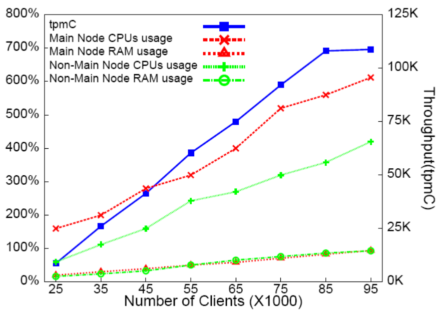
[10]We do not need dedicated client servers for simplicity and minimal hardware cost.



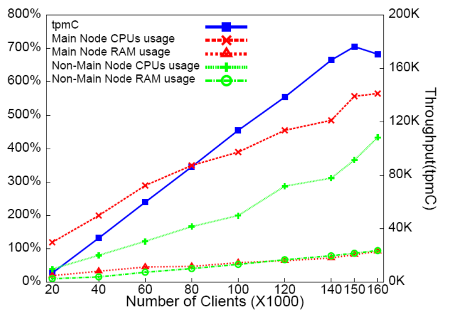
(a) 1-node



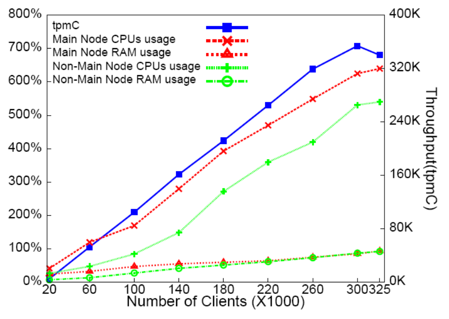
(b) 2-nodes



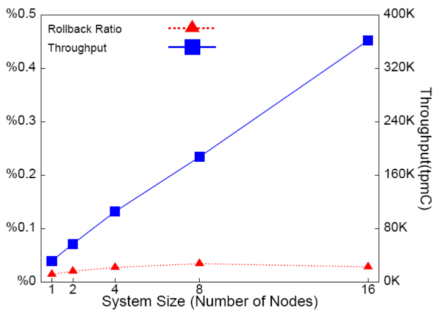
(c) 4-nodes



(d) 8-nodes



(e) 16-nodes



(f) scalability

Figure 6: TPC-C Benchmark Performance

wareh.	size	clients	servers	tpmC
500	72 GB	25000	1	28935
950	135 GB	47500	2	55390
1700	245 GB	85000	4	105572
3200	464 GB	160000	8	184524
6500	943 GB	325000	16	363759

Table 2: TPC-C results

linearly with the growing client number. However, when the system reaches its full capacity, increasing the number of clients will decrease the tpmC. Then the system needs to scale out by adding more servers. The results clearly demonstrate that Rubato DB is scalable for OLTP applications in that the throughput of the benchmark satisfies a linear growth with the increase of the number of servers used. The overall throughput of Rubato DB with different system sizes is plotted in Figure 6(f), which shows the achieved throughput (tpmC) and the rollback ratio on top of the commodity servers cluster. The rollback ratio is stable at a low level of 0.05% as the number of nodes increases. Our experiments also verify that the FPC is scalable in distributed database environment without decreasing the performance.

Another interesting observation is that by using the proposed formula protocol for concurrency and the new staged grid database architecture, we can develop a large scale of database applications running on a collection of commodity servers, without using expensive network-attached storage (NAS) systems and/or cluster servers.

5.3 Conflict Operations over Different Nodes

This set of experiments are design to investigate the impact on the performance of OLTP applications using the formula protocol for concurrency when a large percentage of conflict operations accessing data items distributed over multiple different grid nodes.

We execute a workload derived from the TPC-C benchmark by involving remote guest clients. With remote guest clients, transactions will need to access multiple table partitions across multiple nodes, adding additional network overhead. The performance of the TPC-C benchmark tests with the remote guest accesses varying from 1%, 10%, 20%, and 30% are presented in Figure 7.

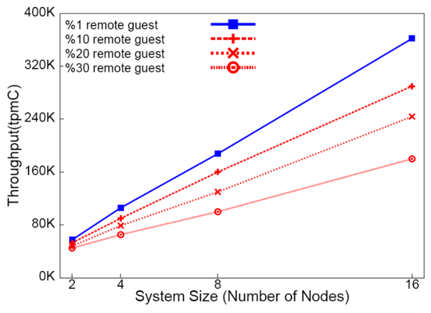


Figure 7: Impact of Remote Guest Clients

Rubato DB runs smoothly and correctly[11] with various percentages of remote guest accesses. As expected, the implementation of FPC does not limit the scalability, unless extensive conflicts occur over distributed nodes, as shown by the performance declining with the increase of remote guest accesses in Figure 7: with 10% remote guest accesses, the throughput of the TPC-C test reduces to about 80% on 8 and 16 grid nodes. The performance further reduces to 65% with 20% remote guest accesses. With 30% remote guest accesses, the performance may decrease nearly by half. We believe that the performance repercussions increase relative to the number of nodes mainly because of the communication cost among distributed nodes, which leaves room for improvement of the FPC.

[11]TPC-C check program is conducted to verify all constraint conditions are passed.

5.4 Stages vs. Threads

Different from traditional databases using multi-threads for parallelism, and staged event-driven architecture applications using multi-threads in their stages, Rubato DB is implemented with a single-thread in each and every staged module in its architecture, partially due to the results demonstrated by the experiment below.

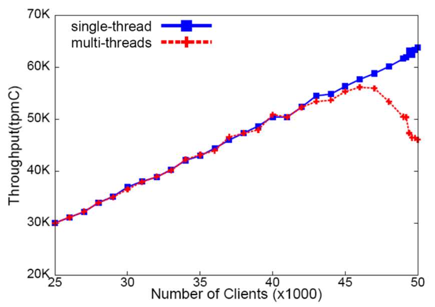


Figure 8: Throughput Comparison

We conducted a sequence of experiments for performance comparison of Rubato DB using single-thread stages vs. multi-threads stages running on a collection of 4 commodity servers. The results are presented in Figure 8. It clearly demonstrates that the superior scalability of the system using a single thread delivers much better performance than that of the system using multiple threads. It has been reported that the multiple threads may pay a heavy price in context switching, especially in transactional processes involving high code foot-print and exhibiting irregular data access patterns [16, 25].

5.5 Performance on YCSB Benchmark

In this set of experiments, we use YCSB benchmark to evaluate the performance of Rubato DB and other typical systems for big data applications including Cassandra[12] (an open-source key-value store), HBase[13] (a open-source Bigtable [8] system) and MySQL Cluster[14] (a scalable relational data store). By setting "AUTO COMMIT ON" option in Rubato DB, each statement issued from any client will be committed automatically regardless of the state of transactions. We use the implemented YCSB client programs[15] for testing two kinds of workloads: read intensive workload (95% read and 5% write operations) and write intensive workload (50% read and 50% write operations). The size of the data is set to 100 million 1KB records for each node, resulting around 120GB of raw data per node. In the experiments, a continuous mixed workload is submitted into the system as per the specification, and the benchmark then measures the performance in terms of throughput (operations/sec) and latency (in milliseconds).

The experiments are conducted on all four systems with the number of nodes as 1, 2, 4, 8, and 12[16]. The system overall throughput comparison is plotted in Figure 9(a), 10(a) and the corresponding latency of read and write operation is shown in Figure 9(b),(c), 10(b),(c) respectively.

Our experiments clearly demonstrate the following:

(a) Rubato DB scales well with increasing throughput and flat latency. That is, more workload can be handled by adding more server nodes into the system.

(b) Overall, the performance of Rubato DB is comparable with that of other popular big data storage systems.

Readers may also find out that the scalability of Rubato DB is comparable to Cassandra, and better than MySQL Cluster. Further, the write latencies for Rubato DB are higher than read latencies, since the update mechanism in the Rubato DB system is designed to provide low read latency at the cost of high write latency.

As a matter of fact, the soft state in BASE properties presents challenges for developers, because it requires complex and error-prone mechanisms to reason about the correctness of the system state. With the FPC, Rubato DB can achieve a higher consistency level-BASIC (**B**asic **A**vailability, **S**calability, **I**nstant **C**onsistency) to get rid of soft state.

5.6 Experimental Summary

The performance experiments of Rubato DB reported in this entire section not just provide answers to questions listed earlier but also provide much needed insights to challenges of scalability and ACID/BASE properties.

1. Rubato DB is highly scalable and efficient for OLTP applications supporting the ACID properties.
2. The performance of Rubato DB is comparable with many NoSQL systems for key-value store applications supporting BASE properties.
3. Rubato DB is capable of handling both OLTP and big data applications on a collection of commodity servers.
4. Rubato DB is considerably cost effective.
5. It is better to use a single thread for all stages in the staged architecture.
6. The scalability of OLTP applications is not limited by using FPC, unless a large percentage of conflict operations access data items distributed over different grid nodes.

6. CONCLUSIONS

In this paper we have introduced a highly scalable database management system, namely Rubato DB, that can be elastically deployed in the grid distributed environment. Extensive experiments on different typical benchmarks verify that the staged grid architecture and the FPC protocol make Rubato DB a satisfactory solution for achieving scalability with both ACID and BASE properties.

7. REFERENCES

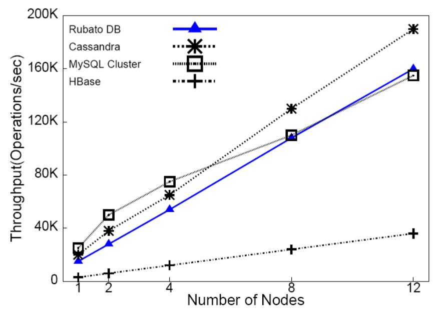
[1] A. Abouzeid and etc. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *PVLDB*, pages 922–933, 2009.

[2] M. K. Aguilera, A. Merchant, and etc. Sinfonia: A new paradigm for building scalable distributed systems. volume 27, pages 5:1–5:48.

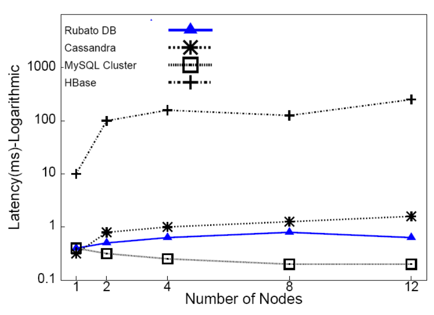
[3] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[4] J. Baker, C. Bond, J. Corbett, and etc. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

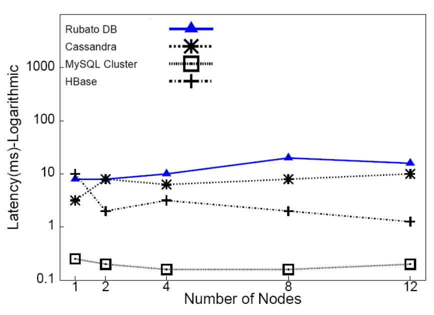
[12]http://cassandra.apache.org/
[13]http://hbase.apache.org/
[14]http://www.mysql.com/cluster/
[15]https://github.com/brianfrankcooper/ycsb/
[16]Our experiments do not use replications. With replications the performance may be improved due to parallel read, the writing effect varies with different update synchronization/asynchronization strategies.



(a) Throughput

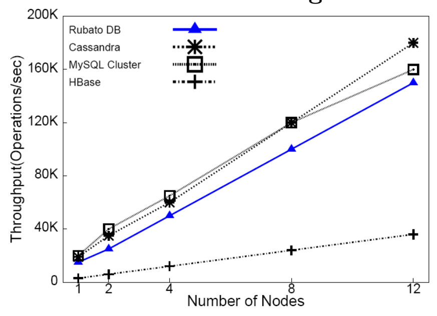


(b) Read latency

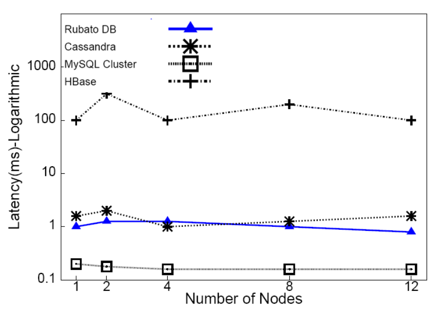


(c) Write latency

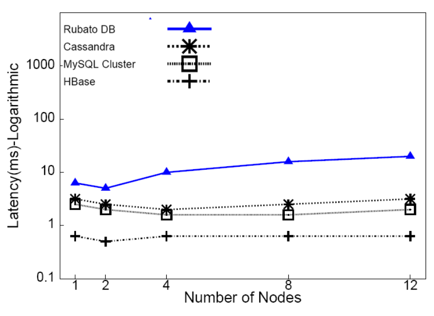
Figure 9: Performance Results for Read-intensive Workload



(a) Throughput



(b) Read latency



(c) Write latency

Figure 10: Performance Results for Write-intensive Workload

[5] S. Blott and H. F. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *PVLDB*, pages 706–717, 2002.

[6] M. Brantner, D. Florescu, and etc. Building a database on s3. In *SIGMOD*, pages 251–264, 2008.

[7] R. Chaiken, B. Jenkins, and etc. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[8] F. Chang, J. Dean, S. Ghemawat, and etc. Bigtable: A distributed storage system for structured data. In *ACM TOCS*, volume 26, pages 1–26, 2008.

[9] J. Cohen, B. Dolan, and etc. Mad skills: new analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[10] B. F. Cooper and etc. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, pages 1277–1288, 2008.

[11] B. F. Cooper and etc. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[12] J. C. Corbett and etc. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Commun.of ACM*, volume 51, pages 107–113, 2008.

[14] G. DeCandia, D. Hastorun, and etc. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[15] A. Fekete, D. Liarokapis, and etc. Making snapshot isolation serializable. In *TODS*, volume 30, pages 492–528, 2005.

[16] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[17] M. Isard, M. Budiu, and etc. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS*, 41(3):59–72, 2007.

[18] R. Kallman and etc. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, pages 1496–1499, 2008.

[19] A. Lakshman. Cassandra: a decentralized structured storage system. In *SIGOPS*, pages 35–40, 2010.

[20] P.-A. Larson, S. Blanas, and etc. High-performance concurrency control mechanisms for main-memory databases. In *PVLDB*, volume 5, pages 298–309, 2011.

[21] K. Manassiev and etc. Exploiting distributed version concurrency in a transactional memory cluster. In *ACM SIGPLAN*, pages 198–208. ACM, 2006.

[22] S. Melnik and etc. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[23] J. Rao and etc. Using paxos to build a scalable, consistent, and highly available datastore. volume 4, pages 243–254. VLDB Endowment, 2011.

[24] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80, June 2011.

[25] M. Stonebraker, S. Madden, and etc. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[26] R. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Digest of papers IEEE COMPCON spring*, pages 56–62, 1984.

[27] A. Thomson, T. Diamond, and etc. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[28] A. Thusoo, J. S. Sarma, and etc. Hive: a warehousing solution over a map-reduce framework. volume 2, pages 1626–1629. VLDB Endowment, Aug. 2009.

[29] TPC-C. http://www.tpc.org/tpcc/. 2010.

[30] VoltDB. http://voltdb.com/products/technology.

[31] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.