MaaT: Effective and scalable coordination of distributed transactions in the cloud

Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi
University of California,
Santa Barbara, CA, USA
{hatem, vaibhavarora, nawab, agrawal,amr}@cs.ucsb.edu

ABSTRACT

The past decade has witnessed an increasing adoption of cloud database technology, which provides better scalability, availability, and fault-tolerance via transparent partitioning and replication, and automatic load balancing and fail-over. However, only a small number of cloud databases provide strong consistency guarantees for distributed transactions, despite decades of research on distributed transaction processing, due to practical challenges that arise in the cloud setting, where failures are the norm, and human administration is minimal. For example, dealing with locks left by transactions initiated by failed machines, and determining a multi-programming level that avoids thrashing without under-utilizing available resources, are some of the challenges that arise when using lock-based transaction processing mechanisms in the cloud context. Even in the case of optimistic concurrency control, most proposals in the literature deal with distributed validation but still require the database to acquire locks during two-phase commit when installing updates of a single transaction on multiple machines. Very little theoretical work has been done to entirely eliminate the need for locking in distributed transactions, including locks acquired during two-phase commit. In this paper, we re-design optimistic concurrency control to eliminate any need for locking even for atomic commitment, while handling the practical issues in earlier theoretical work related to this problem. We conduct an extensive experimental study to evaluate our approach against lock-based methods under various setups and workloads, and demonstrate that our approach provides many practical advantages in the cloud context.

1. INTRODUCTION

The rapid increase in the amount of data that is handled by web services, as well as the globally-distributed client base of those web services, have driven many web service providers towards building datastores that provide more scalability and availability via transparent partitioning and replication, at the expense of transactional guarantees. For example, systems like Google's Bigtable [16], Apache Cassandra [24], and Amazon's Dynamo [19] do not guarantee isolation or atomicity for multi-row transactional updates, so as to avoid the cost of distributed concurrency control and distributed atomic commitment, in order to provide more scalability. Such systems are usually referred to as *cloud datastores* because the location of the data, as well as the partitioning scheme, are totally transparent to the application. However, while cloud datastores relieve the application from the burden of load balancing and fault-tolerance, the lack of transactional support throws the burden of data consistency onto the application. In an attempt to mitigate this issue, major service providers have developed cloud datastores that provide restricted transactional support. For example, systems like Google's Megastore [10], Microsoft's Cloud SQL Server [12], and Oracle's NoSQL Database [34] provide ACID guarantees for transactions whose data accesses are restricted to subsets of the database. Moreover, some research efforts have been directed towards developing methods for partitioning databases in a manner that reduces the need for multi-partition transactions [18, 37, 29]. Restricting transactional guarantees to single-partition transactions takes an important feature out of cloud databases, namely transparent partitioning, which allows non-transactional cloud datastores like Cassandra and Dynamo to automatically repartition data for load balancing. Besides, there are still applications whose data access patterns do not lend themselves easily to partitioning, and for those applications isolation and atomicity of multi-partition transactions are still the job of application developers, resulting in slower and more complex application development. As a result, there has been a renewed interest in distributed transactions for the past decade, stimulating research and development efforts in the industry (e.g., Spanner [17]), in the academic community (e.g., Calvin [41]), and in the open source community (e.g., MySQL Cluster [1]), towards building scalable and highly-available database systems that provide ACID guarantees for distributed transactions; we refer to those systems as *transactional cloud databases*. Transactional cloud databases are still different from the traditional partitioning solutions that all major database management systems provide as options in their recent releases, in the sense that transactional cloud databases are designed with transparent partitioning in mind. A typical cloud database performs automatic re-partitioning and live migration of partitions for

Proceedings of the VLDB Endowment, Vol. 7, No. 5

load balancing at runtime, as well as synchronous replication for automatic fail-over, thus concurrency control needs to be optimized for distributed transactions.

Most transactional cloud databases use some form of locking for concurrency control, either dynamic locking as the case in Spanner and MySQL Cluster, or static locking as the case in Calvin. In fact, aside from Megastore, all major production database management systems, whether designed for the cloud or not, use lock-based concurrency control [13, 36]. However, while lock-based concurrency control is widely-used for implementing transactions at the database layer, optimistic concurrency control (OCC) [22] has been a favorite choice when implementing transactions at the application layer, specially for RESTful web applications that are stateless by definition. For example, Microsoft's .Net framework [6], and the Ruby on Rails framework [2] both use OCC at the application layer, regardless of the concurrency control implemented at the database layer. Also, Google's App Engine [3] still uses Megastore as its database layer, which uses OCC. Optimistic locking is a common way of implementing OCC at the application layer on top of lock-based concurrency control at the database layer [13]. In optimistic locking, each application-level transaction is broken down into two database-level transactions: (1) a read-only transaction that fetches data, and (2) a read-write transaction that checks whether the data fetched by the first, read-only transaction has been changed, and commits updates only if the read data has not been changed. Many database management systems (e.g., IBM DB2 [4] and Oracle DB [5]) provide native support for optimistic locking at the database layer. In general, OCC is very practical for interactive user-facing applications that involve arbitrary user stalls, in which most data reads are not followed by updates, because OCC allows an application to release the database connection after retrieving data, then requesting the connection again if the application needs to perform updates. This saves database resources and allows more application instances to use the database concurrently (e.g., via connection pooling). However, when implementing transactions at the database layer, disk and network stalls are measured in milliseconds, so lock-based pessimistic concurrency control performs well enough, while lock-free OCC wastes relatively high portion of the processing power of a database server aborting transactions that have already been processed [9].

Optimistic concurrency control involves even more practical challenges in the context of distributed databases. Although a fair volume of research has been conducted in the area of distributed OCC [15, 33, 23, 7, 14, 39] soon after the original OCC algorithm was published [22], distributed OCC has rarely been used in practice when implementing transactions at the database layer [42]. In addition to the inherent issue of resource waste due to transaction restarts in OCC, almost all proposed distributed OCC schemes do not eliminate the need for exclusive locking during the final phase of the two-phase commit. When a transaction updates data items on multiple servers, the database needs to lock those data items between the prepare and commit phases of two-phase commit, thus blocking all reads except those made by read-only transactions that can read stale versions of the data. One proposed distributed OCC scheme [23] attempts to work around exclusive locking during two-phase commit using techniques like ordered sharing of locks [8], which is still a form of locking that has its own disadvantages (e.g., cascading aborts). Another proposed distributed OCC scheme [14] eliminates the need for locking during two-phase commit but requires each transaction to perform validation on all data servers in a distributed database system, including data servers not accessed by the transaction, thus adds significant messaging and processing overhead, and sacrificing many of the advantages of distributed databases such as load distribution and fault-tolerance.

Megastore [10] is a rare example of a production distributed database management system that implements OCC. In Megastore, a database is partitioned into *entity groups*, and the system guarantees serializability only among transactions accessing the same entity group by restricting updates on an entity group to only one transaction at a time, while aborting and restarting other concurrent updates. Since transaction latencies in Megastore are measured in hundreds of milliseconds (because updates need to be synchronously replicated across datacenters), the update throughput of each entity group in Megastore is only a few transactions per second [10]. Using finer-grained conflict detection is a possible solution to improve the throughput of Megastore, as shown in [28]. However, Google eventually switched to lock-based multi-version concurrency control in Spanner [17], citing the need to handle long-lived transactions as the motivation behind abandoning OCC and adopting lock-based concurrency control [17]. We question the validity of that reasoning though, since multi-version OCC also performs well in terms of throughput in the presence of long-lived transactions [25]. Instead, we claim that the actual value added by implementing lock-based concurrency control in Spanner is the ability to guarantee atomicity for multi-partition transactions using two-phase commit, since none of the existing distributed OCC mechanisms provide a practical way to perform two-phase commit without locking.

Aside from the practicality issues of OCC, there are advantages of using OCC instead of lock-based concurrency control. First, unlike lock-based concurrency control, systems that use OCC to implement transactions do not experience thrashing when the number of concurrent transactions grows very large [38, 13]. Second, when enough surplus resources are made available to account for aborted transactions, OCC consistently performs better than lock-based concurrency control in experimental evaluations [9]. In this paper, we tackle the practicality issues associated with OCC when implementing transactions at the database layer in a distributed setting. We present a novel re-design of OCC that (1) eliminates the need for any locking during two-phase commit when executing distributed transactions, (2) reduces the abort rate of OCC to a rate much less than that incurred by deadlock avoidance mechanisms in lock-based concurrency control, such as wait-die and wound-wait, (3) preserves the no-thrashing property that characterizes OCC, and (4) preserves the higher throughput that OCC demonstrates compared to lock-based concurrency control. Thus, our novel re-design of OCC overcomes the disadvantages of existing OCC mechanisms, while preserving the advantages of OCC over lock-based concurrency control. We implement our proposed approach as part of a transaction processing system that we develop; we refer to this transaction processing system as MaaT, which is an abbreviation of "Multi-access as a Transaction." We conduct an extensive experimental study to evaluate the performance of MaaT under various workloads and compare its performance against dis-

tributed two-phase locking and traditional optimistic concurrency control.

The rest of the paper is organized as follows. In Section 2 we specify the requirements that a practical transaction processor for the cloud should satisfy. In Section 3, a description of MaaT is given followed by a proof of its correctness and an analysis of its characteristics in Section 4. In Section 5 we explain our implementation of MaaT and show experimental results. We conclude in Section 6.

2. REQUIREMENTS SPECIFICATION

In this section, we discuss the requirements that an ideal cloud database should satisfy. We set these requirements for ourselves to guide our design of a practical transaction processing system based on optimistic concurrency control.

2.1 High throughput

The motivation behind partitioning in cloud databases is to achieve higher throughput by distributing load among multiple data servers. Therefore, high throughput is a key requirement in any cloud database. If all transactions are short, performing few small reads and updates, with few conflicts and no stalls, achieving high throughput is a very easy task; in fact, concurrency control is arguably unnecessary in that case [40]. Instead, we are interested in high throughput that is resilient to long queries, data contention, and potential stalls (e.g., network communication stalls). Single-version concurrency control is known to be problematic under these settings [25]. When using single-version concurrency control with workloads that involve long transactions or data contentions, performance degrades rapidly as many transactions get blocked, when lock-based concurrency control is used, or aborted and restarted, when restart-based concurrency control is used.

2.2 Efficient CPU utilization

Cloud databases are horizontally partitioned to achieve higher throughput by distributing the processing load over multiple data servers. Efficient CPU utilization is crucial to reduce the load on each data server in a system, so as to reduce the need for partitioning, re-partitioning, and migration. Optimistic concurrency control is known for wasting a lot of resources by aborting and restarting transactions after processing them [9, 30]. Also, timestamp ordering [31] is known for incurring many unnecessary rollbacks [7]. A practical concurrency control mechanism should reduce the abort rate as much as possible; for example, by checking conflicts at finer granularities and/or developing techniques that do not abort transactions for mere conflicts.

2.3 Scalability

The trend in the computing industry for the past decade has been to scale out horizontally; that is, to scale out by adding more computing nodes, rather than adding more computing power to a single computing node. We are particularly interested in concurrency control mechanisms that can be implemented with minimal or no mutual exclusion among database threads/processes running over multiple servers. Experimental studies evaluating widely-used database systems on multi-core architectures (e.g., [32]) demonstrate that contention over mutexes in many cases causes the database performance to drop as the number of cores increases, instead of taking advantage of the increasing computing capacity. The performance penalty is even worse in the case of distributed mutual exclusion over a network if the database is distributed over multiple servers.

2.4 No thrashing

Thrashing is a well-known issue in lock-based concurrency control mechanisms [38, 13]. When a system thrashes, its throughput drops rapidly after the number of transactions running concurrently on the system exceeds a particular limit. Thrashing can be avoided by configuring the database so that the concurrency degree (i.e., the number of transactions that run concurrently on the system) does not exceed a particular limit. However, determining an optimum concurrency degree, that neither under-utilizes the system nor causes it to thrash, is not always an easy task and depends on the characteristics of different workloads. Having a concurrency control mechanism that does not thrash relieves database operators from the burden of determining the optimum concurrency degree for each different workload.

2.5 Liveness

Indefinite blocking may occur in case of deadlock, or in case of a crashing client that hold locks. When considering lock-based concurrency control, deadlock detection by constructing wait-for graphs is impractical in a distributed environment. Non-preemptive deadlock avoidance mechanisms, like wait-die, do not allow a transaction to release the locks held by another transactions; thus, if a transaction crashes while holding locks, these locks remain in place, blocking other transactions that access the same data. Preemptive deadlock avoidance, like wound-wait, allows transactions to release the locks of other (possibly crashed) transactions; however, releasing a lock is still not possible after a transaction that owns that lock has started the prepare phase of two-phase commit. The possibility of indefinite blocking is a well-know problem in two-phase commit. Most existing distributed optimistic concurrency control algorithms require locking during two-phase commit, thus encounter the same problem of potentially indefinite blocking. Three-phase commit [35] eliminates indefinite blocking using timeouts. As timeouts are very sensitive to fluctuations in network latency, and in order to handle the case of network partitioning, more sophisticated atomic commit protocols, such as enhanced three-phase commit [21] and Paxos commit [20], have been proposed to eliminate indefinite blocking without relying on timeouts. We are interested in concurrency control mechanisms that avoid indefinite blocking in a distributed setting, or at least lend themselves to existing techniques that eliminate indefinite blocking.

3. MAAT TRANSACTION PROCESSING

3.1 Design Overview

Motivation. We avoid multi-version concurrency control so as to make efficient use of memory space. In fact, the design decision of using single-version concurrency control has been made by several commercial main memory databases as well, such as Oracle TimesTen and IBM's solidDB. In addition to deadlock-related issues, lock-based concurrency control also experiences thrashing when the number of clients accessing the database concurrently grows very large. Concurrency control techniques that require full knowledge of global system state, such as serializability graph checking,

are also problematic since they do not lend themselves to horizontal scalability. Our requirements specification has motivated us to consider re-designing single-version optimistic concurrency control to make it the practical solution for cloud databases, by reducing the abort rate significantly, and by eliminating the need for locking during two-phase commit.

In this section we explain how we re-design optimistic concurrency control in order to make it practical for distributed, high-throughput transactional processing in the cloud. A practical solution should scale out linearly; that is, the throughput of the system, measured in transactions per time unit, should grow as a linear function in the number of data servers. For example if we double the number of servers the throughput of the system should get doubled. In order for lock-free optimistic concurrency control to be practical as a concurrency control mechanism at the database layer for distributed, high-throughput, update-intensive transactional processing, a set of design issues need to be considered. First, to achieve high-throughput with update-intensive workloads, the verification phase of OCC needs to be re-designed so as to reduce the transaction abort rate as much as possible. In particular, a mere conflict between two transactions should not be enough reason to restart any of the two transactions; instead, the system should try to figure out whether this conflict really violates serializability, and should tolerate conflicts whenever possible. Second, in order to be practical for distributed, high-throughput systems, the validation phase of optimistic concurrency control needs to be fully-distributed over data servers, rather than relying on a centralized verifier. Almost all existing OCC mechanisms that are fully distributed involve some kind of locking or distributed mutual exclusion during two-phase commit. To the best of our knowledge, the only distributed OCC mechanism that eliminates the need for locking during two-phase commit is a theoretical proposal [14] that has many practical issues; for example, each transaction needs to be validated on each data servers, including data servers not accessed by the transaction. Third, related to the first issue of reducing transaction abort rate, checking for conflicts should be done at fine granularities (i.e., at row level). Performing this fine-grained conflict checking in optimistic concurrency control has been a challenge [27], and the lack thereof in production implementations of optimistic concurrency control (e.g., Megastore) results in very low update throughput [10].

Timestamp ranges. In order to handle these three design issues, we abandon the way verification has been traditionally implemented in optimistic concurrency control. The original proposal [22] presents a parallel validation mechanism that is based on comparing the write set of a transaction T that is being validated against the read and write sets of other active transactions in the system; thus any read-write or write-write conflict results in a transaction getting aborted. A later OCC protocol [14] uses dynamic timestamp ranges to capture and tolerate conflicts. Each transaction is associated with a timestamp range whose lower bound is initially 0 and whose upper bound is initially ∞. When validating a transaction T, the concurrency control mechanism adjusts the timestamp range of T, mainly by narrowing that timestamp range, to avoid overlapping with other timestamp ranges of transactions conflicting with T. Thus, instead of aborting conflicting transactions, conflicts are translated into constraints on the timestamp ranges of transactions, and abort occurs only when these constraints can not be satisfied. At commit time, if the constraints on the timestamp range of transaction T can be satisfied, the concurrency control mechanism picks an arbitrary timestamp for transaction T from the timestamp range that satisfies the constraints, and assigns that timestamp to T. The concept of dynamic timestamp ranges has been used in other contexts as well, such as pessimistic timestamp ordering [11], and transaction-time databases [26], where adjustments to timestamp ranges are done whenever a conflict occurs, rather than at commit time as the case in OCC. The advantage of deferring timestamp range adjustment to the end of a transaction in OCC is that the concurrency control mechanism can make more informed decisions to avoid as many transaction aborts as possible. In addition to reducing the abort rate, optimistic validation based on dynamic timestamp ranges has the advantage of lending itself to distribution without the need for locking during two-phase commit. However, the original proposal that presents distributed OCC validation based on dynamic timestamp ranges [14] is a theoretical proposal that has practicality issues; mainly, each data server needs to be involved in the validation of each transaction, even data servers that are not accessed by that transaction.

MaaT design. We develop a practical optimistic concurrency control algorithm that still uses dynamic timestamp ranges, yet only the data servers accessed by a transaction are involved in the validation of that transaction. Our optimistic concurrency control algorithm utilizes concepts from pessimistic timestamp ordering and lock-based concurrency control to eliminate the need for validating every transaction on every data server, without being pessimistic or blocking. First, as in the case in pessimistic timestamp ordering [31], our algorithm associates each data item x with two timestamps: (1) a timestamp $ts_r(x)$ of the transaction that last read x, and (2) a timestamp $ts_w(x)$ of the transaction that last wrote x. However, unlike pessimistic timestamp ordering, the two timestamps $ts_r(x)$ and $ts_w(x)$ must refer to committed transactions since only committed transactions have timestamps in our case; in other words, a read (resp. write) operation done by a transaction T on a data item x can not change the read timestamp $ts_r(x)$ (resp. the write timestamp $ts_w(x)$) until T actually commits, if at all. Moreover, whenever a transaction T reads (resp. writes) a data item x whose write timestamp $ts_w(x)$ (resp. read timestamp $ts_r(x)$) is not less than the lower bound of the timestamp range of T, the transaction T does not abort; instead, this conflict results in an adjustment of the lower bound of the timestamp range of T to a value greater than $ts_r(x)$ (resp. $ts_w(x)$), and an abort may occur only if this adjustment makes the lower bound of the timestamp range of T greater than the upper bound of the timestamp range of T. Second, we also use the concept of soft locks to enable our solution. Soft locks are either read or write locks; however, unlike conventional (hard) locks, soft locks do not block transactions. Instead, soft locks act as markers to inform transactions accessing a data item x of other transactions that have read or written x but have not committed yet. Since transactions are not assigned timestamps until they commit, soft locks refer to transactions using unique identifiers that we call *transaction IDs*. A transaction ID is assigned to a transaction when the transaction is initialized (i.e., before any

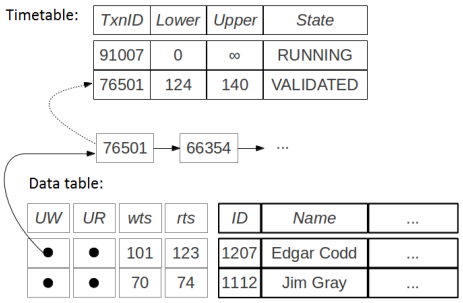

Figure 1: An example of a state of MaaT's data and timetable in a data server

reads or writes) and remains unchanged during the life time of the transaction. By utilizing these two concepts from pessimistic timestamp ordering and lock-based concurrency control, we are able to use dynamic timestamp ranges for optimistic concurrency control without having to perform validation on each data server for each transaction.

Figure 1 shows an example of the state of one data server in MaaT. To facilitate MaaT's concurrency control two tables are maintained. The timetable is a main-memory hash table on each data server that is indexed by the transaction ID, and holds for each transaction the lower bound $lower(T)$ and upper bound $upper(T)$ of the timestamp range of the transaction, as well as the current state of the transaction $state(T)$; there are four possible states: RUNNING, VALIDATED, COMMITTED, or ABORTED. The Data table containing the data objects also maintins the following: (1) Uncommitted Writes (UW) maintains per object a list of transactions that hold soft write locks on the data object, (2) Uncommitted Reads (UR) contains the soft read locks, (3) write timestamp (wts) contains the timestamp of the last committed write on that object, and (4) read timestamp (rts) is the timestamp of the transaction that last read the object. In the example, two data objects are shown, where the object is described by an ID, a name, and other omitted values. Data object with ID 1207 for example, was last written by a transaction with timestamp 101. The Figure also shows a list maintaining transaction IDs of transactions holding soft write locks on data object 1207. Transaction 76501 is shown in the timetable where it is VALIDATED and has a lower bound of 124 and upper bound of 140. The next section will show how the state is changed in MaaT and how transactions are committed.

3.2 Concurrency Control

In MaaT, each transaction T undergoes the following phases.

3.2.1 Initialize

During initialization, a transaction T is assigned to a data server that acts as the *home data server*. Typically the home data server should be the data server where the transaction performs most of its reads and updates, but that is not necessary. The home data server assigns to the transaction a globally unique transaction ID, that we refer to as $tid(T)$, by appending the current local time according to the ID of the home data server itself. The home data server then creates a new entry for the transaction in the *timetable*. Initially, the lower bound is set to 0, the upper bound is set to ∞, and the transaction state is set to RUNNING.

3.2.2 Read

A transaction T may read data from the home data server of T, or other data servers. Whenever the transaction T reads data from a remote data server s, the data server s creates an entry for the transaction T in the local timetable at s, if such an entry does not exist already. The data server s initializes the timetable entry of T to the same values used in the initialization phase; that is, the lower bound equals 0, the upper bound equals ∞, and the transaction state is RUNNING. Let x denote the data item that T reads from s. Whether the data server s that T reads from is the home data server of T or a remote data server, the read operation triggers the following operations at the transaction processor of the data server s.

- The transaction processor places a soft read lock on the data item x that is read by T. The transaction ID of T is attached to this soft read lock, to inform subsequent transactions that T has read x.

- When the transaction processor returns the value of x back to T, the transaction processor also returns the transaction IDs of all transactions that have soft write locks placed on x; we refer to the set of transactions with soft write locks on x as $UW(x)$. Soft write locks are placed by transactions during their validate-and-prewrite phase to indicate that a particular data item will be updated at commit time without actually updating the data item. The transaction T needs to collect those transaction IDs, so that when T commits, T should be assigned a commit timestamp that is less than the commit timestamp of any of the transactions in $UW(x)$, because T did not see the updates made by any of the transactions in $UW(x)$.

- The transaction processor returns to T the write timestamp $ts_w(x)$, which represents the largest commit timestamp of a transaction that updated x. The transaction T needs to get the write timestamp $ts_w(x)$, so that when T commits, T should be assigned a commit timestamp that is greater than $ts_w(x)$, because T saw the updates made by the transaction whose commit timestamp is the write timestamp $ts_w(x)$.

During the read phase, all updates made by T are buffered locally at the buffer space of T and do not affect the actual content of the database until the transaction T is done with all its reads, then the transaction processor moves T to the prewrite-and-validate phase.

3.2.3 Prewrite and Validate

After all reads are done, the transaction T sends prewrite-and-validate messages to relevant data servers. Relevant data servers include data servers from which T has read data items during the read phase, and data servers to which T needs to write data items. Typically, the write set of T is subset of the read set of T, but this is not required. When sending a prewrite-and-validate message to a data server s, the transaction T attaches to the prewrite-and-validate message any information returned by the transaction processor of s during the read phase; that is, for each data item x read

by T from s, the set of uncommitted writes $UW(x)$ and the write timestamp $ts_w(x)$.

When a data server s receives a prewrite-and-validate message from a transaction T, the transaction processor at s checks that the transaction T has a timetable entry at s; if not, the transaction processor initializes a timetable entry for T. Next, the transaction processor at s performs the following set of operations for each data item y that T needs to write on s, if any. We refer to the following set of operations as prewrite operations.

- The transaction processor places a soft write lock on y to inform subsequent transactions that T will update y at commit time.
- The transaction processor logs the update operation, as well as the value to be assigned y by the transaction T when T commits.
- The transaction processor collects the transaction IDs of all transactions with soft read locks on y; we refer to the set of transactions with soft read locks on y as $UR(y)$. The transaction T needs to collect those transaction IDs, so that when T commits, T should be assigned a commit timestamp that is greater than the commit timestamp of any of the transactions in $UR(y)$, because none of the transactions in $UR(y)$ saw the update made by T on y.
- The transaction processor fetches the read timestamp $ts_r(y)$, which represents the largest commit timestamp of a transaction that read y. The transaction T needs to get the read timestamp $ts_r(y)$, so that when T commits, T should be assigned a commit timestamp that is greater than $ts_r(y)$, because the transaction whose timestamp is $ts_r(y)$ did not see the updates made by T. The read timestamp $ts_r(y)$ is guaranteed to be greater than the write timestamp $ts_w(y)$, thus there is no need to fetch the write timestamp as well.
- The transaction processor collects the transaction IDs of all transactions with soft write locks on y; we refer to the set of transactions with soft write locks on y as $UW(y)$. The order of the commit timestamps of those transactions with respect to the commit timestamp of T is not determined at this point, and will be determined by the transaction processor during the validation phase. However, there must be a total order between the timestamps transactions in $UW(y)$, including T, because they are conflicting.

Once the transaction processor at the data server s finishes all prewrite operations for all data items that T needs to write on s, the transaction processor performs a set of operations to validate transaction T. Validation involves adjusting the timestamp range of T and/or the timestamp ranges of other transactions to ensure that the timestamp ranges of conflicting transactions do not overlap. The outcome of these validation operations is to determine whether the constraints on the commit timestamp of T can be satsified or not; that is, whether T can commit or not. If T is a distributed transaction, other data servers accessed by T may make different commit decisions; thus a final commit decision can be made only when the client that executes T receives commit decisions from all data servers accessed by T. We explain the validation operations as follows.

- For each data item x read by T on s, the transaction processor checks that the lower bound of the timestamp range of T is greater than the write timestamp of x; that is, $lower(T) > ts_w(x)$. If not, the transaction processor adjusts $lower(T)$ to enforce this inequality.
- For each transaction T' in $UW(x)$, for each data item x read by T, the transaction processor checks that the lower bound of T' is greater than the upper bound of T; that is, $lower(T') > upper(T)$. If yes, the transaction processor does nothing. Otherwise, the transaction processor needs to adjust either the lower bound of T' or the upper bound of T to enforce this inequality. If the state of T' is VALIDATED or COMMITTED, the timestamp range of T' can not be modified, thus the transaction processor adjusts the upper bound of T. For example, in Figure 1 say transaction 91007 reads record 1207, which has a soft write lock held by transaction 76501 which has been VALIDATED. This will lead to the upper bound of transaction 91007 to be set to less than the lower bound of transaction 76501, i.e 124. If the state of T' is ABORTED, the transaction processor adjusts the lower bound of T'. Otherwise, if the state of T' is RUNNING, the transaction processor has the choice to adjust either or both of the timestamp ranges of T and T' to avoid their overlapping; the transaction processor defers this decision to a latter step, and adds T' to a set $after(T)$ of transactions whose lower bounds need to be greater than the upper bound of T, but whose adjustments will be determined later.
- For each data item y written by T on s, the transaction processor checks that the lower bound of the timestamp range of T is greater than the read timestamp of y; that is, $lower(T) > ts_r(x)$. If not, the transaction processor adjusts $lower(T)$ to enforce this inequality. For example, in Figure 1, if transaction 91007 writes record 1112, then its lower bound would be greater than the read timestamp of record 1113, i.e 74.
- For each transaction T' in $UR(y)$, for each data item y written by T, the transaction processor checks that the upper bound of T' is less than the lower bound of T; that is, $upper(T') < lower(T)$. If yes, the transaction processor does nothing. Otherwise, the transaction processor needs to adjust either the upper bound of T' or the lower bound of T to enforce this inequality. If the state of T' is VALIDATED or COMMITTED, the timestamp range of T' can not be modified, thus the transaction processor adjusts the lower bound of T. If the state of T' is ABORTED, the transaction processor adjusts the upper bound of T'. Otherwise, if the state of T' is RUNNING, the transaction processor adds T' to a set $before(T)$ of transactions whose upper bounds need to be less than the lower bound of T, but whose adjustments will be determined later.
- For each transaction T' in $UW(y)$, for each data item y written by T, the transaction processor checks the state of T'. If the state of T' is ABORTED, the transaction processor ignores T'. If the state of T' is VALIDATED or COMMITTED, the transaction processor needs to ensure that the upper bound of T' is less than

the lower bound of T; that is, $upper(T') < lower(T)$. If this inequality does not hold, the transaction processor adjusts the lower bound of T to enforce this inequality. Otherwise, if the state of T' is RUNNING, the transaction processor adds T' to a set $after(T)$ of transactions whose lower bounds need to be greater than the upper bound of T, but whose adjustments will be determined later.

After performing the previous operations, the transaction processor checks if the lower bound of T is still less than the upper of T. If no, the transaction processor changes the state of T to ABORTED. Otherwise, the transaction processor changes the state of T to VALIDATED. Although the transaction processor has not decided yet how the transactions in $before(T)$ and $after(T)$ will be adjusted, the transaction processor never aborts a transaction that is being validated to save a transaction that is still running. A simple-minded policy would be to leave the timestamp range of T as it is at this point, and adjust the timestamp ranges of all transactions in $before(T)$ and $after(T)$ accordingly, aborting as many transactions of them as it takes to keep the timestamp range of T intact. Instead, to reduce the abort rate, the transaction processor assigns to T a sub-range of the range from $lower(T)$ to $upper(T)$ such that the number of transactions in $before(T)$ and $after(T)$ that are forced to abort is minimized. The transaction processor then sends a response back to the client executing T to indicate whether T should commit or abort. If the validation decision is to commit, the transaction processor attaches to the response the timestamp range of T as set locally in the timetable of the data server s.

3.2.4 Commit or Abort

The client collects validation decisions, and the attached timestamp ranges, from all data servers on which T is validated. If one or more of the validation decisions is an abort, the transaction T is aborted. If all validation decisions indicate a commit, the client computes the intersection of all timestamp ranges returned by all data servers accesed by T. This intersection needs to be a valid range; that is, a range whose lower bound is no more than its upper bound. If the intersection is not a valid range the transaction T gets aborted; otherwise, T is committed, and the client picks an arbitrary timestamp from the intersection range to be the commit timestamp of T. The client forwards the final decision, whether to commit or abort, as well as the commit timestamp if any, to all data servers accessed by T. Whenever the client sends a commit message to a data server, the client attaches again all buffered writes that need to be done on that data server, since these updates are not applied to the database during the prewrite-and-validate phase.

Whenever a data server s receives an abort message from T, the transaction processor at s sets the state of T to ABORTED, removes all soft locks placed by T on any data items, and logs the abort operation. If s receives a commit message from T, the transaction processor at s performs the following operations.

- The transaction processor sets the state of T to COMMITTED, removes all soft locks placed by T on any data items, logs the commit operation.
- The transaction processor sets both the lower bound and the upper bound of the timestamp range of T in the timetable at s to the commit timestamp of T.
- For each data item x read by T from s, the transaction processor compares the read timestamp $ts_r(x)$ against the commit timestamp of T. If the commit timestamp of T is greater than the read timestamp of x, the transaction processor sets the read timestamp of x to the commit timestamp of T; otherwise, the read timestamp of x remains unchanged.
- For each data item y written by T on s, the transaction processor compares the write timestamp $ts_w(y)$ against the commit timestamp of T. If the commit timestamp of T is greater than the write timestamp of y, the transaction processor sets the write timestamp of y to the commit timestamp of T, and applies the write of T to the database (i.e., sets the value of y to the value written by T); otherwise, the write timestamp of y, as well as the value of y, remain unchanged.

3.3 Garbage collection

For any given transaction T, as long as the state of T on a data server s is set to RUNNING or VALIDATED, the timetable entry of T on s can not be garbage collected. If the state of T on s becomes ABORTED, the timetable entry of T on s can be garbage collected immediately because the lower and upper bounds of the timestamp range of T can be set to any arbitrary values, thus there is no need to save these bounds in the timetable; other transactions conflicting with T on s recognize that T is ABORTED when they can not find the entry of T in the timetable. If the state of T becomes COMMITTED on a data server s, the timetable entry of T on s can be garbage collected once it is no longer needed for the validation of other transactions on s. Let x be any data item in the read set of T on s, and let T' be any transaction that places a soft write lock on x after T places its soft read lock on x, and before T removes its soft read lock from x. T' needs the timetable entry of T to perform validation. Thus, if T commits before T' is validated, the timetable entry of T should not be garbage collected until T' performs validation. Similarly, let y be any data item in the write set of T on s, and let T' be any transaction that places a soft read lock or a soft write lock on y after T places its soft write lock on y, and before T removes its soft write lock from y; if T commits before T' is validated, the timetable entry of T should not be garbage collected until T' performs validation.

One possible approach is to use a garbage collection mechanism that is based on reference counting by adding a reference count field to each timetable entry. When the state of T becomes COMMITTED on s, the transaction processor sets the reference count of T on s to the number of RUNNING transactions that will need the timetable entry of T on s during their validation. Whenever a transaction T' uses the timetable entry of T on s for validation, and recognizes that the state of T is COMMITTED, T' decrements the reference count of T on s. When the reference count of T on s becomes 0, the timetable entry of T on s is garbage collected by the transaction processor. Alternatively, it is possible to use a more eager approach in which the timetable entry of T on s is garbage collected immediately after the state of T on s become COMMITTED. To see this, let T'

be any RUNNING transaction that will need the timetable entry of T during validation; note that the constraints imposed by T on T' are determined at the moment the state of T becomes COMMITTED, and do not change thereafter. Thus, the transaction processor can proactively apply the constraints imposed by T on T' as soon as T commit, then the transaction processor can remove the timetable entry of T immediately. When T' performs validation on s, T' will not find the entry of T in the timetable, thus will ignore T; this still does not affect the correctness of the transaction history because the constraints imposed by T on T' has already been applied. The difference between the two garbage collection approaches, the reference counting approach and the eager approach, is analogous to the different between backward and forward validation in traditional optimistic concurrency control.

4. ANALYSIS

In this section, we analyze the various aspects of our concurrency control algorithm explained in Section 3.2.

4.1 Correctness

The following theorem states the correctness of MaaT.

THEOREM 4.1. *Transaction histories generated by the algorithm in Section 3.2 are conflict serializable.*

Proof: Assume for contradiction that a transaction history H generated by MaaT is not in conflict serializable. Then there exists a cycle of n transactions $T_1, ..., T_n$ in H, such that for each two consecutive transactions in the cycle, say T_1 and T_2, there exists an object x that was either (1) updated by T_1 then updated by T_2, (2) updated by T_1 then read by T_2, or (3) read by T_1 then updated by T_2. We prove that such a cycle never exists by showing that for all three types of conflicts between T_1 and T_2, the commit timestamp of T_1 is less than the commit timestamp of T_2; that is, $ts(T_1) < ts(T_2)$, thus a cycle is impossible. Assume a conflict between T_1 and T_2 on object x. Let s be the data server hosting x, note that x is updated only when s receives a commit message from a transaction, not when s receives a prewrite message.

Consider Case (1) when x is updated by T_1 then read by T_2. When x gets updated by T_1, the write timestamp of x becomes at least the commit timestamp of T_1. The write timestamp of x is fetched by T_2 when x receives the read request from T_2, thus the commit timestamp of T_1 becomes a lower bound on the timestamp range of T_2. Similarly, in Case (2) when x is updated by T_1 then updated by T_2, when x gets updated by T_1, the write timestamp fo x becomes at least the commit timestamp of T_1, and the read timestamp of x is set to be at least the write timestamp of x. The read timestamp of x is fetched by T_2 when x receives the update request from T_2, thus the commit timestamp of T_1 becomes a lower bound on the timestamp range of T_2. Consider Case (3) when x is read by T_1 then updated by T_2. In this case, either T_1 commits before x receives a prewrite from T_2, or T_1 commits after x receives a prewrite from T_2. If T_1 commits before the prewrite of T_2, the read timestamp of x becomes at least the commit timestamp of T_1, then the read timestamp of x is fetched by T_2 when x receives the prewrite of T_2, thus the commit timestamp of T_1 becomes a lower bound on the commit timestamp of T_2. Otherwise, if x receives the prewrite of T_2 before T_1 commits, either (3-a) T_1 encounters the soft write lock placed by T_2 on x, if x receives the prewrite of T_2 before the read of T_1, or (3-b) T_2 encounters the soft read lock placed by T_1 on x, if x receives the read of T_1 before the prewrite of T_2. In either case, one of the two transactions is aware of the conflict. During validation, the transaction that is aware of the conflict either imposes a constraint on the timestamp range of the other transaction, if the other transaction has not undergone validation yet, or on itself if the other transaction has already undergone validation. In particular, in Case (3-a) if T_1 gets validated before T_2, T_1 imposes a constraint on T_2 to force it to pick a timestamp greater than that of T_1; otherwise, if T_2 gets validated before T_1, T_1 imposes a constraint on itself to pick a timestamp less than that of T_2. Similarly, in Case (3-b) if T_2 gets validated before T_1, T_2 imposes a constraint on T_1 to force T_1 to pick a timestamp less than that of T_2; otherwise, if T_1 gets validated before T_2, T_2 imposes a constraint on itself to pick a timestamp greater than that of T_1. Thus, in both cases, the timestamp of T_1 is less than that of T_2.

Having proved that in Cases (1), (2), and (3), the commit timestamp of T_1 is less than the commit timestamp of T_2 (i.e., $ts(T_1) < ts(T_2)$), we apply this inequality to each two consecutive transactions in the cycle $T_1, ..., T_n$. From the transitivity of the $<$ operator, we infer that the commit timestamp of T_1 is less than itself (i.e., $ts(T_1) < ts(T_1)$), which is a contradiction. Thus, a cycle is impossible, and the transaction history H is guaranteed to be conflict serializable. ■

4.2 Memory utilization

The memory needs of our concurrency control algorithm arise from (1) the timetable at each data server, (2) the (soft) lock table at each data server, and (3) the read and write timestamps associated with each data item in the database. The timetable space reserved for any given transaction T is constant, but the size of the lock table entry of T is proportional to the number of data items accessed by T, which is a small number in typical transactional workloads. The memory allocated to a transaction T can be freed as soon as T commits, but read and write timestamps remain attached to data items.

4.3 CPU utilization

Unlike most existing concurrency control algorithms, our algorithm neither aborts nor blocks a transaction because of a mere conflict with another transaction. In MaaT, aborts occur only in the following case. Consider a transaction T that conflicts with two transactions, T_1 and T_2, such that the type of conflict between T and T_1 is different from the type of conflict between T and T_2; for example, the conflict between T and T_1 could be write-read while the conflict between T and T_2 could be read-write or write-write. If both T_1 and T_2 perform validation before T performs validation, both T_1 and T_2 impose their constraints on the timestamp range of T. In this example, T_1 imposes a lower bound on the timestamp range of T, while T_2 imposes an upper bound on the timestamp range of T. If the lower bound imposed by T_1 is greater than the upper bound imposed by T_2, then T must abort. The probability that this scenario occurs is much less than the probability of a simple conflict. In fact, as we show in our experiments, the abort rate of our algorithm is much

less than the abort rate of deadlock avoidance mechanisms such as wait-die.

4.4 Fault-tolerance

Since MaaT is totally lock-free, an aborting transaction never causes indefinite blocking of other transactions. The main concern when a transaction aborts is to release soft write locks held by this aborted transaction as soon as possible. This concern is due to the fact that soft write locks impose upper bounds on the timestamp ranges of conflicting transactions, and these upper bounds when left indefinitely become eventually unsatisfiable, and may lead to unnecessary aborts of other transactions. For example, if a transaction T aborts while holding a soft write lock on a data item y, a subsequent transaction T' that reads or updates y has to pick a timestamp that is less than the lower bound on the timestamp range of T. The transaction T' may or may not be able to obtain a timestamp less than the lower bound of T, depending on the read and write timestamps of other data items that T' update and read. For example, if T' needs to read or update another data item z whose write timestamp $ts_w(z)$ is not less than the lower bound of T, then T' has to abort. As in lock-based methods, timeouts and three-phase commit can be used to ensure that locks are never held indefinitely. However, in lock-based mechanisms, locks left by aborted transactions affect the throughput of the system as other transactions queue up waiting for these locks to timeout, causing cascading blocking; meanwhile, in MaaT, no cascading blocking occurs since transactions never block at all.

5. EXPERIMENTS

We begin by explaining our framework and implementation of MaaT. Then we perform an evaluation of performance of MaaT. In this evaluation, one of our aims is to compare MaaT, which has an optimisitc concurrency control scheme to pessimistic concurrency control techniques to evaluate their differences. MaaT will be compared with two reference implementations of locking protocols. The first part of the evaluation will compare with distributed two-phase locking, which is the basis of concurrency control schemes in many current database management systems executing distributed transactions. The next part will compare with a deterministic locking scheme.

5.1 Framework

In MaaT, a database is partitioned based on key ranges, and each partition is managed by a single-threaded database server process, thus no latches or mutexes are required. An application accesses the database through a statically-linked client library that exchanges messages with database server processes using BSD sockets. The workload of transaction processing is divided between database server processes and the client library that runs in application process space. The database server process is responsible for storing and retrieving data, logging transaction operations, and maintaining the timetable, the lock table, and the timestamp cache. The client library is responsible for mapping keys to data servers, sending read and write requests to data servers, receiving responses, and performing commits and aborts. The application passes to the client library a set of callback functions to invoke upon receiving responses from data servers; these callback functions perform the necessary processing on

Figure 2: Througput Analysis

data and may request more reads and writes. Although the client library is single-threaded, an application may execute multiple concurrent transactions through the client library, which keeps track of reads, writes, and callbacks of each transaction.

Our evaluation uses the standard benchmark TPC-C and focuses on the New Order transaction, which models the purchase of 5-10 items from one or more partitions. We implement MaaT and distributed two-phase locking in C++, and run our experiments on Amazon EC2 machines. We use small 32-bit EC2 machines (m1.small), which promise 1 EC2 compute unit per machine; that is, equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. In all our experiments, we place one TPC-C warehouse partition on each server machine, and execute a workload of 10,000 TPC-C New Order transactions on each server. The number of transactions running concurrently on each servers, as well as the percentage of distributed transactions, differ from one experiment to another. Each client is assigned a home server. A distributed transaction accesses items in servers other than the home server. For each item in the transaction, the probability that it accesses a partition other than the one in its home server is equal to the percentage of distributed transactions. If an item was assigned to access a different partition, then it will choose a partition other than the one in the home server using a uniform distribution.

5.2 Overall evaluation with distributed two-phase locking

In the distributed two-phase locking implementation, each data server maintains a lock table of items. When data are read, a read lock is requested. If all read locks are acquired, the write set is sent to all accessed servers. If all write locks are successfully acquired, a commit message is sent to all servers. In the implementation a wait/die approach is used to avoid deadlocks.

5.2.1 Scaling Up

Figure 2 illustrates the performance of MaaT, in terms of throughput, compared to distributed two-phase locking, as the number of clients per server increases from 1 to 50. The number of clients per server is usually also referred to as the multi-programming level (MPL). In this set of experiments, the number of servers is fixed at 50, with one database process per server, managing one warehouse of the TPC-C database. Transactions timeout within 100 milliseconds for both MaaT and distributed two-phase locking. As shown in Figure 2, when the percentage of distributed transactions

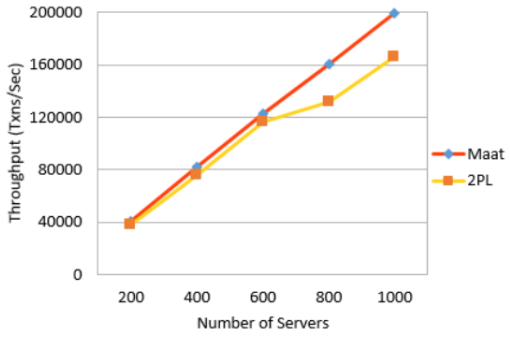

Figure 3: Scale-out Throughput

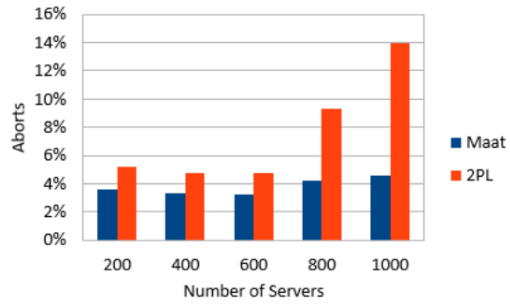

Figure 4: Scale-out Aborts

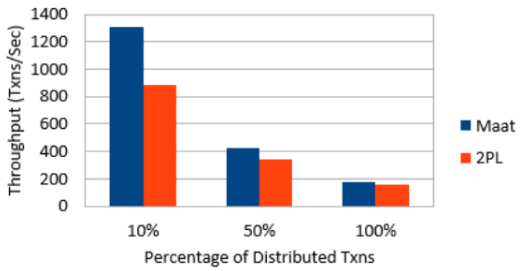

Figure 5: Fault-tolerance Throughput

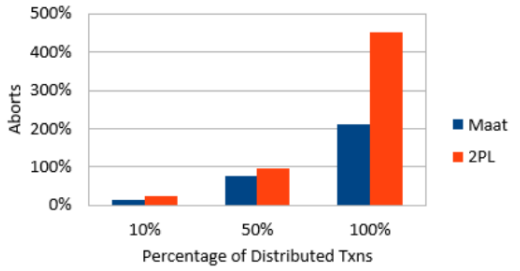

Figure 6: Fault-tolerance Aborts

is as low as 10%, which is the standard percentage stated by the TPC-C benchmark, there is barely any difference between the performance of MaaT and the performance of distributed 2PL; for both mechanisms the throughput grows rapidly as the number of clients per server increases from 1 to 5, then the throughput remains unchanged as the number of clients per server increases beyond 5. As we increase the percentage of distributed transactions to 50% and 100%, the throughput of both mechanisms decreases; however, starting from MPL=10, the throughput of MaaT remains mostly unchanged, while the throughput of distributed 2PL drops rapidly. This set of experiments demonstrates that MaaT is more resilient to variability in workload compared to lock-based distributed concurrency control.

5.2.2 *Scaling Out*

In this set of experiments we run the TPC-C workload with MPL=5 on a number of servers that ranges from 200 to 1000. Figure 3 illustrates the performance of MaaT compared to distributed two-phase locking, as the number of servers increases from 200 to 1000. As shown in Figure 3, the throughputs of MaaT and 2PL grow linearly. However, as the number increases over 600 servers, 2PL is not able to scale as efficiently as MaaT. Figure 4 shows the percentage of aborts incurred by each of the two concurrency control mechanisms. As shown in Figure 4, the percentage of aborts incurred by distributed 2PL remains slightly higher than MaaT, until the system reaches a tipping point when the percentage of aborts incurred by distributed 2PL starts to increase as the number of servers increases, while the number of aborts incurred by MaaT remains relatively stable. This set of experiments shows that MaaT is more effective than lock-based concurrency control when it comes to very large scale databases, and scales out much more smoothly.

5.2.3 *Fault-tolerance*

In this set of experiments, we evaluate the resilience of both MaaT and distributed two-phase locking to failures. We run our TPC-C workloads on 10 servers, with MPL=5, and with different percentages of distributed transactions, and evaluate the performance of the system when one of the 10 servers is faulty.

Figure 5 shows the throughput of both MaaT and 2PL for different percentages of distributed transactions. In most cases, MaaT's throughput is higher than 2PL. However, it is especially worth noting that in the 10% distributed transaction case, and in contrast to Figure 2 (the no failure case) where both throughput were comparable, here, with one failure the throughput of MaaT is over 25% higher than 2PL. The difference in throughput decreases as the percentage of distributed transactions increases. In contrast, in Figure 6, which shows the percentage of aborts, the percentage of aborted transactions in 2PL increases significantly more than MaaT as the percentage of distributed transactions increases. In particular, in the case of 100% distributed transactions the number of aborted transactions is almost double in 2PL over MaaT.

5.3 Evaluation with deterministic locking

The scalability exhibited by MaaT is an attractive feature sought by distributed transaction systems. A design model that has been proven to be effective in scaling with a large number of machines is the deterministic concurrency model. One notable example is Calvin [41, 40]. In Calvin, there are three layers of execution: (1) sequencers are used to define a global ordering of transactions, (2) schedulers receive transactions from sequencers and acquire locks deterministically according to the global order, and (3) execution threads are used to process transactions once all locks

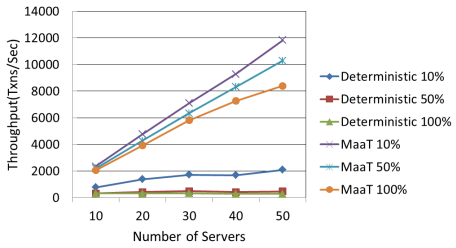

Figure 7: Scale-out experiment with deterministic locking

are acquired. This design is enabled by a transaction model that requires a transaction read and write sets to be initially declared together when contacting the sequencers. This is different from MaaT's flexible framework where the complete read and write sets don't have to be determined at the beginning.

Given the success of deterministic protocols, we have designed a deterministic locking protocol that is built on top of our existing framework. Our aim is to study the effect of a deterministic deployment on top of our framework. Each client will act as a sequencer and will assign unique IDs to transactions. These are ensured by assigning IDs as multiples of the client ID. The lock manager, rather than granting locks in the order of received transactions, will grant locks according to the global order of the received transactions. However as mentioned, unlike in Calvin, in our model, clients do not generate both the read and write-sets at the start of the transaction. Thus, it is not possible to deliver the whole transaction to the sequencer and the servers. Thus, in our case, clients need to send the read and write operations to the servers. Reads are grouped in one package and then sent as a whole. Once the read locks are acquired, the client sends the write-set as a whole to the accessed servers. For example, if a client is accessing data objects x, y, and z, each residing at a different server, the client will send three packages each containing one object. One challenge that arises is that each server needs to know which transactions do not access it so that it can proceed to serve the next transactions. To do this, the client sends notifications to servers that are not accessed by a transaction.

This deterministic deployment was tested to study its applicability to our framework. Figure 7 shows the results of the deterministic deployment compared to MaaT while varying the number of servers from 10 to 50. One client runs at each server. These experiments are run on larger EC2 machines ($c1.medium$). This is because the overhead exhibited in our deterministic deployment made us unable to run experiments on smaller machines. This overhead will be analyzed in this experiment. MaaT is shown to scale as the number of servers increases to achieve a throughput of 12000 transactions per second for 50 servers compared to 2000 transactions per second for 10 servers in the case with 10% distributed transactions. Our deterministic deployment on the other hand achieved a maximum throughput of 2000 transactions per second for the case of 10% distributed transactions on 50 servers, which is only 16.67% of what is achieved by MaaT. One reason for this is that in our framework, a limited number of clients are run and each client only has a limited number of allowed concurrent transactions. Since clients are waiting for the read data objects to be read before writing and moving to the next transaction, transactions are not batched in large numbers to the servers. This causes the communication overhead to affect each transaction rather than distributing the overhead among a large number of transactions in a batch. This study demonstrates that significant changes need to be made to the deterministic approach for it to succeed in a more general framework where (1) the clients run a limited amount of concurrent transactions , and (2) the complete read / write sets of a transaction cannot be known at the beginning of the transaction.

6. CONCLUSION

The past decade has witnessed an increasing adoption of cloud database technology; however, only a small number of cloud databases provide strong consistency guarantees for distributed transactions, due to practical challenges that arise because of distributed lock management in the cloud setting, where failures are the norm, and human administration is minimal. Most distributed optimistic concurrency control proposals in the literature deal with distributed validation but still require the database to acquire locks during two-phase commit, when installing updates of a single transaction on multiple machines. In this paper, we re-design optimistic concurrency control to eliminate any need for locking during two-phase commit, while handling the practical issues in earlier theoretical work related to this problem. We conducted an extensive experimental study to evaluate MaaT against lock-based methods under various setups and workloads, and demonstrate that our approach provides many practical advantages over lock-based methods in the cloud context.

7. ACKNOWLEDGEMENTS

This work is partially supported by a gift grant from NEC Labs America and NSF Grant 1053594. Faisal Nawab is partially funded by a fellowship from King Fahd University of Petroleum and Minerals. We would also like to thank Amaozn for access to Amazon EC2.

8. REFERENCES

[1] http://www.mysql.com/products/cluster/.
[2] http://api.rubyonrails.org/.
[3] http://developers.google.com/appengine/.
[4] http://www.ibm.com/software/data/db2/.
[5] http://www.oracle.com/database/.
[6] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ado.net entity framework. In *SIGMOD*, 2007.
[7] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.
[8] D. Agrawal, A. El Abbadi, R. Jeffers, and L. Lin. Ordered shared locks for real-time databases. *The VLDB Journal*, 4(1):87–126, Jan. 1995.
[9] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, Nov. 1987.

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

[11] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In *Proceedings of the Second International Symposium on Distributed Data Bases*, DDB '82, pages 9–20, 1982.

[12] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *ICDE*, 2011.

[13] P. A. Bernstein and E. Newcomer. *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[14] C. Boksenbaum, M. Cart, J. Ferrié, and J.-F. Pons. Certification by intervals of timestamps in distributed database systems. In *VLDB*, 1984.

[15] S. Ceri and S. S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Berkeley Workshop*, pages 117–129, 1982.

[16] 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. In *OSDI*, 2006.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[18] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[19] 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. In *SOSP*, 2007.

[20] J. Gray and L. Lamport. Consensus on transaction commit. Technical Report MSR-TR-2003-96, Microsoft Research, 2004.

[21] I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *J. Comput. Syst. Sci.*, 57(3):309–324, Dec. 1998.

[22] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[23] M.-Y. Lai and W. K. Wilkinson. Distributed transaction management in jasmin. In *VLDB*, 1984.

[24] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *PODC*, 2009.

[25] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[26] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. In *ICDE*, 2012.

[27] C. Mohan. Less optimism about optimistic concurrency control. In *Research Issues on Data Engineering, 1992: Transaction and Query Processing, Second International Workshop on*, pages 199 –204, feb 1992.

[28] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: concurrency control and availability in multi-datacenter datastores. *Proc. VLDB Endow.*, 5(11):1459–1470, July 2012.

[29] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, 2012.

[30] P. Peinl and A. Reuter. Empirical comparison of database concurrency control schemes. In *VLDB*, 1983.

[31] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[32] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *EuroSys*, 2011.

[33] G. Schlageter. Optimistic methods for concurrency control in distributed database systems. In *VLDB*, 1981.

[34] M. Seltzer. Oracle nosql database. In *Oracle White Paper*, 2011.

[35] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed systems. *IEEE Transactions on Software Engineering*, pages 219–228, 1983.

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

[37] A. Tatarowicz, C. Curino, E. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, 2012.

[38] A. Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Trans. Database Syst.*, 18(4):579–625, 1993.

[39] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Trans. on Knowl. and Data Eng.*, 10(1):173–189, Jan. 1998.

[40] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1-2):70–80, Sept. 2010.

[41] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[42] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.