# Dynamic Timestamp Allocation for Reducing Transaction Aborts

Vaibhav Arora, Ravi Kumar Suresh Babu, Sujaya Maiyya, Divyakant Agrawal, Amr El Abbadi
vaibhavarora, ravikumar, sujaya_maiyya, agrawal, amr@cs.ucsb.edu
Department of Computer Science, University of California, Santa Barbara
Xun Xue, Zhiyanan, Zhujianfeng
xun.xue, zhiyanan, zhujianfeng@huawei.com
Huawei

***Abstract*—CockroachDB is an open-source database, providing transactional access to data in a distributed setting. CockroachDB employs a multi-version timestamp ordering protocol to provide serializability. This provides a simple mechanism to enforce serializability, but the static timestamp allocation scheme can lead to a high number of aborts under contention. We aim to reduce the aborts for transactional workloads by integrating a dynamic timestamp ordering based concurrency control scheme in CockroachDB. Dynamic timestamp ordering scheme tries to reduce the number of aborts by allocating timestamps dynamically based on the conflicts of accessed data items. This gives a transaction higher chance to fit on a logically serializable timeline, especially in workloads with high contention.**

## I. Introduction

CockroachDB [2] is an open-source distributed SQL database built on a transactional and strongly-consistent key-value store. The transactional guarantees are provided over data, which is synchronously replicated using a distributed consensus protocol, Raft [18].

CockroachDB provides two transaction isolation levels: SI (Snapshot Isolation) [9] and Serializable [10]. Snapshot Isolation can detect write-write conflicts among transactions and provides efficient performance, but does not guarantee serializability [15]. Applications needing stricter correctness guarantees than snapshot isolation can use the Serializable isolation level, which is provided using a Serializable Snapshot Isolation (SSI) [23] based technique. To implement SSI, CockroachDB employs a lock-free multi-version timestamp ordering scheme. The timestamp ordering scheme in CockroachDB uses a fixed timestamp allocation scheme, and assigns timestamps at the start of each transaction. These timestamps are used as commit timestamp for the transaction. This commit timestamp is used to order the transactions in logical timestamp order, and hence enforce serializability. SSI, unlike SI, detects and handles read-write conflicts but the restrictive fixed timestamp ordering leads to higher number of aborts under contention. Figure 1 illustrates that with increasing contention in the workload, the number of aborts increases, leading to a drop in throughput with timestamp ordering concurrency-control employed in CockroachDB.

To reduce the number of aborts at high contention, a possible strategy is to use dynamic timestamps for allocating the commit timestamp to a transaction. In this technique, the system tries to dynamically allocate a timestamp range to each transaction, based on the conflicts of data items accessed by the transaction. At commit, a timestamp is allocated to the transaction from that range, to fit that transaction on a logical serializable timeline. We integrate CockroachDB with the dynamic timestamp allocation technique we designed in our previous work MaaT (Multiaccess as a Transaction) [17]. MaaT changes the validation phase of the optimistic concurrency control mechanism to allocate a commit timestamp dynamically by keeping track of the items accessed and conflicting transactions accessing the items using read and write markers. By performing dynamic timestamp allocation during validation and commit, MaaT is able to reduce aborts under contention.

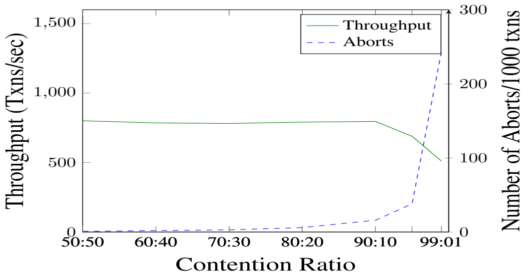


Fig. 1: Performance of Fixed Timestamp Ordering Scheme under Contention. Contention ratio x:y specifies that x percent of the transactions access y percent of the items.

Adapting the dynamic timestamp allocation technique designed in MaaT with the SSI implementation in CockroachDB can eliminate unnecessary aborts. Our technique targets cases where transactions are aborted due to being non-serializable because of a fixed timestamp ordering, rather than being aborted because of actual conflicting order of access of items (in other words an ordering that would lead to cycle in a conflict graph). Some of these cases are serializable when timestamps are dynamically allocated to fit them on a logicial timeline, based on the order of access of data items.

CockroachDB is modified to integrate the dynamic timestamp ordering based concurrency-control mechanism. As the concurrency-control layer interacts with the lower layers of the database for ensuring strong consistency of data, multiple components had to be modified to integrate the technique.

Past techniques proposed to reduce aborts in lock-free concurrency control techniques [26], [17], [25] have been implemented and evaluated in standalone prototypes. Implementing the dynamic timestamp ordering technique in a full-featured database system like CockroachDB gives an insight into the performance characteristics of the dynamic timestamp ordering like optimizations, and how to integrate such optimizations in existing systems.

A benchmark is also developed to extensively evaluate the performance of dynamic timestamp ordering based concurrency-control in CockroachDB. The benchmark evaluates the performance while varying contention, the degree of concurrent access and the ratio of read-only and read-write transactions. The source-code changes as well as the benchmark are available on Github [4], [1].

The rest of the paper is organized as follows. Related work is discussed in Section II. In Section III, we describe CockroachDB's architecture. Section IV describes the dynamic timestamping technique employed by MaaT. An abstract overview of the integration of dynamic timestamp allocation in CockroachDB is presented in Section V. The details of the integration and the implementation are provided in Section VI. Evaluation results are presented in Section VII. Section VIII concludes the paper.

## II. Background

Snapshot Isolation (SI) has been implemented in major database systems, like Oracle and PostgreSQL. Fekete et al. [15] illustrated that SI does not provide serializability and then subsequently Fekete et al. [14] studied the transaction patterns occurring in SI violations. Various techniques have been proposed to make Snapshot Isolation serializable. Some of the techniques [14], [16] perform static analysis of application code and detect SI violation patterns. The potential violations are translated into write-write conflicts, which are then detected by snapshot isolation. However, these techniques are limited in scope and cannot be applied to systems dynamically generating transactions. Cahill et al. [12] develop a SSI (Serializable Snapshot Isolation) methodology to detect SI violations at run-time and implement the technique over existing snapshot isolation providing database, BerkleyDB. CockroachDB's technique for providing serializable snapshot isolation is inspired by a multi-version timestamp ordering [19] variant proposed by Yabandeh et al. [23].

Various mechanisms have been proposed to reduce aborts in pessimistic as well as optimistic concurrency control (CC) algorithms. For locking algorithms, variants of 2PL such as Order-shared [6] locking, Altruistic Locking [20] and Transaction chopping [21] techniques have been proposed to reduce aborts and achieve higher throughput. Recent works [24] have also explored performing static analysis of application code to order locking requests, such that lock contention is minimized.

Among the lock-free concurrency control schemes, many pessimistic, as well as optimistic techniques employ timestamps to enforce serializable order. The timestamp allocation may be done at the beginning or at the end of the transaction. If a transactional operation violates the timestamp order, the transaction is aborted. Dynamic timestamp allocation [11] schemes have been developed to allocate transactions dynamically, rather than using a fixed timestamp allocation scheme. MaaT [17] employs dynamic timestamps in a distributed setting. MaaT dynamically allocates logical timestamps during validation phase and utilizes soft read and write locks to avoid locking of items during two-phase commit (2PC) between the prepare and commit phase. Our proposed technique builds on MaaT. Tic-toc [25] uses the idea of dynamic timestamp allocation in a single-server setting. BCC [26] defines essential patterns which occur in non-serializable patterns in Optimistic concurrency control (OCC) algorithms. Rather than aborting a transaction on detecting an anti-dependency, as in OCC, BCC tracks dependencies to abort transactions only when these non-serializable patterns are detected. Although a BCC like technique would reduce the number of aborts, it adds extra overhead to track dependencies.

Deterministic transaction scheduling [22] has also been proposed to eliminate transaction aborts and improve performance under high contention. However, such techniques need a priori knowledge of read-write sets and do not work in an ad-hoc transaction access setting, like in CockroachDB.

CockroachDB's [2] SSI technique employs timestamp allocation, is lock-free and can support distributed transactions. Hence, dynamic timestamping proposed in MaaT is a good fit for CockroachDB, since it is timestamp based and lock-free. The SI implementation in CockroachDB also provides a mechanism to push commit timestamps for distributed transactions. However, this technique cannot be applied to SSI [3].

## III. CockroachDB overview

CockroachDB [2] is an open-source distributed cloud database built on top of a transactional and consistent key-value store. Its primary design goals are scalability and strong consistency. CockroachDB aims to tolerate disk, machine, rack, and datacenter failures with minimal disruption and no manual intervention, thus being survivable (hence the name).

CockroachDB achieves strong consistency by synchronous replication of data. It replicates data over multiple nodes and guarantees consistency between replicas using Raft [18] consensus protocol. Cockroach provides transactional access to data. It supports two isolation levels: Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI). While both provide lock-free reads and writes, SI allows write-skew [9], and does not guarantee a serializable history. SSI eliminates write-skew but introduces a performance hit in cases of high contention.

### A. Architecture

CockroachDB implements a layered architecture as shown in Figure 2. The highest level of abstraction is the SQL Layer, which acts as an interface to the application clients. Every SQL statement received at this layer, is converted to an equivalent key-value operation.

The Transaction Coordinator receives the key-value operations from the SQL layer. It creates the context for the transaction if it is the first operation of the transaction or else forwards the request to the Distributed Sender in the context of an existing transaction. The Transaction Coordinator also sends a begin transaction request to create transaction record if the operation is the first write operation of the transaction, and also keeps track of the keys written by the transaction.

The Distributed Sender communicates with any number of cockroach nodes. Each node contains one or more stores, one per physical storage device in that node. Each store contains potentially many ranges. Each range comprises a contiguous group of keys. A range is equivalent to a partition or a shard. Each range or shard can have multiple copies for providing fault tolerance. Ranges are defined by start and end keys. They are merged and split to maintain total byte size within

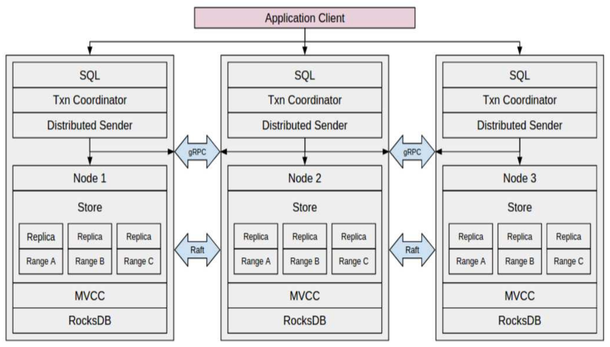


Fig. 2: Architecture of CockroachDB. Shows a cluster of 3 nodes (Node 1 to Node 3), and 3 ranges (Range A to Range C) with full replication. Changes are replicated using Raft.

a globally configurable min/max size interval. The data in the multiple copies is synchronized using Raft [18], [8], at the granularity of the range. The Distributed Sender forwards the requests from the Transaction Coordinator to the lease holder of the range (Lease Holder is an entity within Raft, which holds a non-overlapping lease interval for a range) having the key on which request is operating upon. The lease holder performs the read and writes operations, while ensuring that reads return the latest value, and writes are replicated to the majority of the servers, before successfully committing a transaction.

The logical entity called Replica at each range's lease holder is responsible for creating and maintaining transaction records, updating the timestamp cache, performing read and write operations by calling the the MVCC (Multi Version Concurrency Control) Layer and also replicating the transaction record, write intents and resolving those intents using Raft.

Data is stored in RocksDB [5]. RocksDB ensures efficient storage and access of data. Data stored in RocksDB includes the key-value data and all the versions associated to a key, as well as all the consensus state associated with Raft.

### *B. Transaction Processing and Concurrency Control*

CockroachDB uses a multi-version timestamp ordering protocol to guarantee that its transaction commit history is serializable. The default isolation level is called Serializable Snapshot Isolation(SSI). The SSI mechanism employed in CockroachDB is lock-free and ensures concurrent read-write transactions will never result in anomalies.

Every transaction is assigned a timestamp (by the node on which it starts) when it begins. This timestamp is used to resolve conflicts with respect to timestamp ordering. Each transaction has a transaction record, that stores the status of the transaction. Every transaction starts with the initial status PENDING. If a transaction is aborted due to data conflicts, the status is changed to ABORTED, or else its status is changed to COMMITTED on commit.

If a transaction is distributed, i.e., the data accessed by the transaction is spread across multiple ranges, then the transaction record is maintained only at one of the servers having data accessed by the transaction. All the operations of the transaction update the same transaction record according to the status of the operation at the particular data server.

CockroachDB keys store multiple timestamped versions of the values. Every new write of a committed transaction creates a new version of the value with a timestamp of that transaction. The write of an uncommitted transaction is added as an intent version with the timestamp of the transaction.

Read operations on a key return the most recent version with a smaller timestamp than the transaction. The timestamp of the transaction performing the read operation is recorded in a node-local timestamp cache. This cache returns the most recent timestamp of a transaction which read the key.

All write operations consult the timestamp cache for the read timestamps of keys they are writing. If the returned timestamp is greater than the transaction's timestamp, then this indicates a timestamp order violation. Hence, the transaction is aborted and restarted with a larger timestamp.

Operations in CockroachDB are only allowed to read or write committed values; operations are never allowed to operate on an uncommitted value. That is, if an uncommitted transaction wants to read or write a value of a key and if it sees a write intent for that key, one of those transactions is aborted based on their priorities.

SSI, due to its restrictive policy to avoid read-write conflicts, ends up aborting or restarting many transactions. With fixed timestamp allocation for transactions, SSI trades performance for correctness under high contention. Our interest is in increasing performance by eliminating the unnecessary aborts realized by SSI by dynamically assigning timestamps to transactions based on the conflicts, such that they follow a serializable timestamp order.

## IV. MaaT overview

MaaT (Multi-access as a Transaction) [17] re-designs the optimistic concurrency control (OCC) protocol in order to make it practical for distributed, high-throughput transactional processing systems. To achieve high-throughput with update intensive workloads, MaaT assigns dynamic timestamps to the transaction, instead of fixed timestamps, and re-designs the verification phase of OCC so as to reduce the transaction abort rate. In particular, a mere conflict between two transactions should not be enough to restart them; instead, the system should try to figure out whether this conflict really violates serializability, and should tolerate conflicts whenever possible.

While performing read and write operations, each transaction maintains and updates valid timestamp ranges it can commit in, based on the data items accessed and conflicting operations on those data items. At commit time, if there is a valid timestamp range for the transaction to commit, then a commit timestamp will be chosen from that valid range. Employing timestamp ranges allows the flexibility to shift the commit timestamp to fit it in the logical serializable order.

### *Design*

MaaT assigns a timestamp range with a lower and upper bound for each transaction, instead of a fixed timestamp. Initially, the lower bound is set to 0 and the upper bound is set to $\infty$. The lower and upper bounds are adjusted dynamically with respect to the data conflicts of the transaction. MaaT employs soft locks, which do not block transactions, to act as markers to inform transactions accessing a data item about other transactions that have read or written that data item but not committed yet.

Each transaction can either be single-sited or distributed. For a distributed transaction, one of the servers involved in the transaction is declared the transaction coordinator. Timestamp range is maintained at each of the servers accessed by the transaction. Next, we describe how the timestamp range are dynamically adjusted, and used to allocate commit timestamp to a transactions. This is one of the core components of MaaT, which leads to reduced aborts under contention.

If $T$ and $T'$ are two transactions, with lower bounds, *lowerbound(T)* and $lowerbound(T')$ and upper bounds, $upperbound(T)$ and $upperbound(T')$. Whenever $T$ reads a data item, which has a write soft lock by $T'$, MaaT adjusts the $upperbound(T)$ to be less than $lowerbound(T')$, so that $T$ executes as if did not see the updates made by $T'$.

Whenever $T$ writes a data item, which has a read soft lock by $T'$, MaaT adjusts the $lowerbound(T)$ to be greater than *upperbound(T')*, so that $T'$ executes as if it did not see the updates that will be made by $T$.

Whenever $T$ writes a data item, which has a write soft lock by $T'$, MaaT adjusts the $lowerbound(T)$ to be greater than *upperbound(T')*, so that $T'$ executes as if it did not see the updates that will be made by $T$.

In the validation phase, the timestamp range of $T$ and/or the timestamp ranges of other transactions are adjusted 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 satisfied or not; that is, whether $T$ can commit or not, by finding a timestamp to fit $T$ in the logical serialization order. Transaction $T$ is aborted if there is an overlap of the timestamp range with other concurrent transactions or if the lower bound of $T$ is greater than its upper bound; otherwise, $T$ is committed, and the client picks an arbitrary timestamp from the intersection range to be the commit timestamp of $T$.

If $T$ is a distributed transaction, each data server accessed by $T$ adjusts the timestamp range on the server based on the items accessed there. All these ranges are sent to the coordinator of the distributed transaction. If there is a valid intersecting range in the timestamp ranges sent by all the servers, the transaction is committed. Otherwise, the transaction aborts.

## V. Abstract Overview of Dynamic Timestamp Ordering Adaptation in Cockroach DB

Employing SSI in CockroachDB ensures correctness, but it decreases the performance at high contention. Adapting SSI approach in CockroachDB to work with dynamic timestamps will avoid aborting transactions that can be serialized by changing their timestamps with regards to their data conflicts. Next, an example is analyzed to illustrate how dynamic timestamping can lead to reduced aborts as compared to the current concurrency control technique employed in CockorachDB. We then describe the data structures introduced in CockroachDB to enable dynamic timestamping mechanism.

### *A. Reducing aborts with dynamic timestamping*

When the isolation level in CockroachDB is set to SSI, transactions are aborted in following cases:

1) Conflicts are analyzed at every write, to check whether any later transactions have read or written the data item currently being written by the transaction. If this is the case, the transaction is aborted.
2) When reading an item, if there is a write intent created with a lower timestamp, then the SSI approach in CockroachDB will abort one of the transactions. The transaction to abort will be decided based on the priority of the transactions involved.

Dynamic timestamp allocation can help reduce some of the aborts in the above cases. It allows more concurrent operations to commit by dynamically trying to commit the transaction on a logical timeline based on order of access of items. Consider the following example.

*Example 1:* Lets consider the following two transactions.

$$T_1 : r_1(\text{y})\ r_1(\text{x})$$
$$T_2 : r_2(\text{x})\ w_2(\text{x})$$

Suppose the execution history comprising the two transactions is as follows.

$H_1 : b_2\ r_2(\text{x})\ b_1\ r_1(\text{y})\ r_1(\text{x})\ c_1\ w_2(\text{x})\ c_2$

Transaction $T_2$ begins before $T_1$, and hence is assigned an earlier timestamp than $T_1$. At $w_2(x)$, the SSI approach in Cockroach DB sees that x is read by $T_1$, which is a transaction with a later timestamp. To ensure timestamp ordering given by the fixed timestamp allocation, transaction $T_2$ is aborted since it causes RW conflict with $T_1$, and violates the logical timestamp order according to the allocated timestamps.

In case of using the dynamic timestamping technique, suppose $lowerbound(T_1)$ and $lowerbound(T_2)$ are the lower bounds and $upperbound(T_1)$ and $upperbound(T_2)$ are the upper bounds of the transactions $T_1$ and $T_2$ respectively. On detecting the RW conflict at $w_2$, the $lowerbound(T_2)$ is made greater than $upperbound(T_1)$ so that history will be equivalent to the serialization order $T_1 \longrightarrow T_2$. Rather than aborting the transaction due to an initially allocated timestamp order, the dynamic timestamp allocation can reorder the logical transaction order, and lead to commitment of both the transactions in this case.

### *B. Data Structure changes in CockroachDB*

*Transaction Record*

CockroachDB maintains a transaction record for each transaction, which maintains the transaction ID and transaction state. We add two fields to hold the lower and upper bounds of the transaction. *CommitBeforeQueue* field is added to hold the list of transactions before which the current transaction has to commit. *CommitAfterQueue* holds the list of transactions after which the current transaction can commit.

*Timestamp cache*

For each data item CockroachDB maintains the timestamp of last read and last written transaction in its *timestamp cache*. In the SSI approach, this can be updated even by uncommitted transactions. We modify the mechanism updating the timestamp cache in CockroachDB to ensure timestamps in the read cache are updated only by committed transactions.

*Soft Locks and Soft Lock Cache*

Soft Locks are non blocking markers to inform other transactions about ongoing transactions. Soft Locks comprise transaction-metadata, which is used to locate the transaction record of the transaction that placed the soft lock. A read soft lock is placed while reading and a write soft lock is placed while writing the data items. The Soft Lock Cache holds the read soft locks and write soft locks per key.

## VI. Integrating dynamic timestamp ordering in CockroachDB

We now describe the integration of dynamic timestamping mechanism in CockroachDB. First, the handling of different operations of the transaction is described. Then, we describe the changes implemented at different layers in CockroachDB.

### A. Transaction Lifecyle

As described earlier, a transaction comprises of begin, read, write and commit operations. On every read operation, a soft read lock is placed on the key being read and soft write locks that are already placed on the key are collected. The soft write locks are then placed in the corresponding transaction record. The *CommitBeforeQueue* is populated with the transactions which correspond to the soft write locks on the data item, indicating that the transaction reading the item should commit before all the transactions, which have a soft write lock on the item. As the read of the item does not reflect the update of the transactions intending to write the item, its logical commit order should be before such transactions. *CommitBeforeQueue* is used during transaction validation to enforce the transaction ordering implied by the queue.

On every write, instead of aborting transactions on detecting a conflict based on fixed timestamps, a soft write lock is placed on the key being written and, read and write soft locks that are already placed on the key are collected and placed in the transaction record. The *CommitAfterQueue* is populated with the transactions which correspond to the soft read locks on the data item, indicating that the current transaction intending to write the data item, should commit after the transactions which have a soft read lock on the item (implying the intention to read the data item). As the read performed by the transactions, which have the soft read lock, does not reflect the write operation of the current transaction, the current transaction's logical commit order should be after the transactions with the soft read lock. *CommitAfterQueue* is also populated with the transactions which correspond to the soft write locks on the data item, indicating that the current transaction intending to write the data item, should commit after the transactions which have a soft write lock on the item.

Along with the above described processing, on every read or write operation, the lower bound of the transaction is adjusted to be equal to the last committed write timestamp (or read timestamp), which is retrieved from the timestamp cache.

When a commit request for the transaction is sent, a validation phase is executed. During the validation phase, the lower and upper bounds of the transaction are adjusted such that the transaction commits before all the entries in *CommitBeforeQueue* and commits after all the entries in *CommitAfterQueue*. The *CommitBeforeQueue* and *CommitAfterQueue* have been populated with all the transactions, which have conflicting operations with the given transactions. Hence, respecting the commitment order enforced by the queues, guarantees that the transaction orders all conflicting operations and preserves serializability. The upper bound of the transaction is updated to be the minimum of its current upper bound and the lower bound of each transaction in the *CommitBeforeQueue*. Similarly, the lower bound of the transaction is updated to be the maximum of current lower bound and the upper bound of each transaction in the *CommitAfterQueue*.

The timestamp range of the transaction is then checked to see if the lower bound of the transaction is less than its upper bound. If so, the transaction is committed by picking the lower bound, as the commit timestamp; otherwise the transaction is aborted. The transaction status in the transaction record is updated accordingly as COMMITTED or ABORTED.

On both commit and abort of the transaction, soft read locks and soft write locks held by that transaction are released. On commit, the write soft locks will be resolved to actual write operations in the DB. The read and write timestamps of the items accessed by the committed transaction will be updated in the timestamp cache.

If the transaction spans across multiple ranges, RPCs are used to update the transaction record during read and write operation and to validate the transaction at the remote range (employing the validation strategy to update transaction bounds described above) during validation phase. The soft locks in the remote ranges are resolved asynchronously using RPCs, while the soft locks local to the range are resolved synchronously during the end transaction request.

The dynamic timestamp allocation based proposed design makes use of the soft locks to detect the conflicting transactions. It then tries to reorder the transactions by analyzing these conflicts and allocating dynamic timestamps to the transactions to fit them in a logical serializable timestamp order.

### B. Implementation Details

Multiple components have been modified in CockroachDB to integrate the dynamic timestamp ordering approach.

The Transaction Coordinator is modified to send the begin transaction request to create transaction record on the first operation of the transaction rather than on first write of the transaction. Transaction coordinator now additionally tracks the keys read by the transaction along with the keys written by the transaction.

Transaction record is modified to hold the lower bound and the upper bound of the transaction, with zero and infinity being the initial values respectively. Two new queues, namely *CommitBeforeQueue* and *CommitAfterQueue*, are introduced in the transaction record. These queues are used at validation to adjust the lower and upper bounds of the transaction based on conflicts of the transaction.

At each replica, the timestamp cache is modified to hold only the timestamps of the committed transactions. A replica is also responsible for maintaining the soft lock cache for all the keys in that replica.

APIs for performing writes at the MVCC layer (such as MVCCPut, MVCCInitPut etc.) are modified to not place the intent, and instead place soft write locks against the key intended to write. Read operation APIs in MVCC (such as MVCCGet, MVCCScan etc.) are modified to place soft read lock on the key being read along with reading the value for the key. New MVCC APIs are created to resolve write soft lock on commit, and write the committed values to the key-value store and to garbage collect soft locks on abort.

New RPCs are created to update the transaction record at the remote range for distributed transactions. Additionally, these RPCs are also used to perform validation on the transaction record in the remote range. Additional RPCs are created to execute commit or abort processing after the validation phase and to resolve remote soft locks asynchronously on commit. Another RPC is introduced to garbage collect remote soft locks asynchronously on aborting a transaction.

The Store layer is modified to handle the asynchronous resolving and garbage collecting of remote soft locks, like the asynchronous resolving of write intents that was done before, for the pessimistic timestamp allocation mechanism.

The source-code changes are available on Github [4].

## VII. Evaluation

CockroachDB is extensively evaluated to compare the performance of the fixed timestamp allocation scheme, with the dynamic timestamp ordering scheme, under varying levels of concurrent access, contention and read-write ratios.

### A. Experimental Setup

#### 1) Benchmark Description

Yahoo Cloud Serving Benchmark (YCSB) [13] is a benchmark for evaluating different cloud datastores. A YCSB-like benchmark is designed for performing the evaluation. The benchmark provides support for transactions, rather than only key-value operations, like in YCSB. Every transaction generated by the benchmark can be a read-only or a read-write transaction. Every transaction consists of 5 operations. The benchmark has 3 configurable parameters that can be altered to test different scenarios. They are described below.

- **Concurrent transactions.** This parameter is used to define the number of transactions that occur concurrently, with a default value of 50. If the *concurrent transactions* is set to 50, the benchmark creates 50 threads and each thread runs a transaction sequentially i.e., if a transaction is performing 3 reads and 2 writes, each of the operations is blocking and the thread executes these operations one after the other.
- **Contention ratio.** In most real world applications, the entire dataset is not uniformly accessed. There can be some subset of data with a highly-skewed access pattern. To mimic similar behavior in our experiments, we define the contention ratio, which indicates skew on a subset of data. The contention ratio 70:30 indicates 70% of the data items will be accessed by 30% of the transactions, while the remaining 30% of the data items will be accessed by remaining 70% of the transactions. The default contention ratio is a uniform distribution of 50:50, which corresponds to the lowest contention of 50%.
- **Read-only ratio.** In order to see how the dynamic timestamp ordering works for various scenarios such as write intensive transactions or read-dominant transactions, we introduced the read-only ratio parameter. Read-only ratio defines the ratio or the percent of transactions that perform only read operations. For the default value 50:50, 50% of the transactions have only read operations (5 per transaction) and rest of the transactions will be read-write transactions with 3 read operations and 2 write operations. Before beginning a new transaction, we toss a coin with the defined bias based on the read-only ratio parameter value and decide whether the transaction is going to be read-only or read-write.

***Experimental Configuration***: Every experiment was performed on a cluster with 3 servers. Each machine in the cluster runs a 8-core Intel Xeon E5620 processor clocked at 3.8 GHz and has 16 GB of RAM. Each server had one instance of CockroachDB node running on it and the data was replicated three ways. The experiments were run with 100,000 key-value data items without data partition, hence the experiments did not have distributed transactions. Each data point in the benchmarking process corresponds to an experiment performed with 100,000 transactions accessing 100,000 data items. 10000 warm-up reads are performed before starting each experimental run, so as to have the system in a steady state. Each data point reported in the results is an average of 3 repetitions of such an experiment. The benchmark is available on Github [1].

#### 2) Experimental Methodology

As mentioned in V-B, for dynamic timestamp ordering, the creation of a transaction record happens on the first operation, be it a read or a write, rather than on the first write operation, as is the case with fixed timestamp ordering. In fixed timestamp ordering, if the transaction consisted of all read-only operations, no transaction record is created. Furthermore, the creation of the transaction record also encompasses the replication of the transaction record (to the replicas holding the range associated to the first key accessed by the transaction) using Raft, both at the begin and when the commit decision is made. Hence using this approach in a read-dominant system, adds significant delays compared to the original implementation of CockroachDB. There are many approaches to avoid the bookkeeping done by MaaT for read-only transactions. One such solution was proposed by Agrawal et al. [7], which can be implemented in CockroachDB along with MaaT. A read-frontier based on this approach proposed by Agrawal et al. [7] can be continuously maintained, which provides access to the latest commit timestamp below which no transaction can commit. The timestamp corresponding to the read frontier can be used as the timestamp to read the items accessed in the read-only transactions.

In order to apply optimized solutions for read-only transactions, we first need to identify if the transaction will be read-only. After a discussion with CockroachDB developers, we learnt that there is no provision as of now in the database to specify a transaction as read-only. Implementing this change will require modification in multiple layers of the database and is a complex task, and it is not in the scope of this paper. In order to overcome this issue, in our experiments, we assume that every read-only transaction is successful and will not have the overhead of creating and maintaining a transaction record. If the optimization was to be implemented, read-only transactions would hit only one server, which would respond to the client without creating and replicating transaction record among the replicas of the queried data. So we do not account for the time taken to process and perform read-only transactions. Because of the above mentioned assumption, the throughput displayed in the following sections will be an upper bound of the actual throughput for dynamic timestamp ordering. But the number of aborts or retries is accurate because if the read-only optimization is implemented, it does not cause any read-only transaction to fail; all the aborts will be due to read-write transactions which is captured in our experiments.

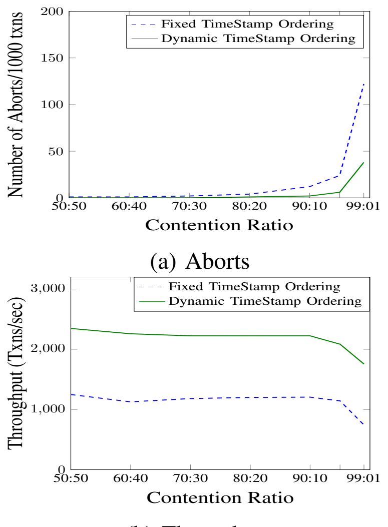


Fig. 3: Varying contention with 80% read-only transactions

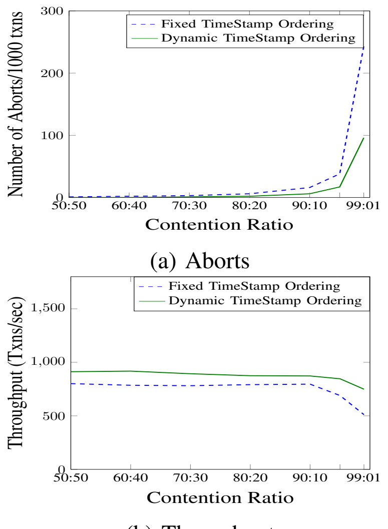


Fig. 4: Varying contention with 50% read-only transactions

## B. Evaluation Results

### 1) Varying Contention

First, we analyze the performance of the concurrency-control schemes under varying levels of contention on the data items. We performed two experiments with varying contention from 50% to 99%. In one, we set the read-only ratio and concurrency level with the default values of 80:20 and 50 respectively. Figure 3a plots the number of aborts for every 1000 transactions with increasing contention for both dynamic and static timestamp ordering. Even for 20% write transactions, when the contention is at its highest, the fixed timestamp ordering scheme has roughly 3 times more aborts for every 1000 transactions than dynamic timestamping. Figure 3b compares the throughput of both the techniques and we observe that dynamic timestamp ordering has significantly higher throughput. In the second experiment with varied contention, the read-only ratio was lowered to 50:50, while keeping the concurrency level to 50. Figures 4a and 4b illustrate the results. As mentioned in the methodology, the throughput for dynamic timestamp ordering is an upper bound, especially for higher read-only ratios.

In both Figures 3a and 4a, although both the techniques started with small abort numbers on low contention, dynamic timestamp ordering results in significantly lower number of aborts with the increase in contention. When contention is high, say 99:01, 99% of the transactions are trying to access 1% of data. Dynamic timestamp ordering ends up accomodating more transactions for commitment due to its flexibility in shifting the lowerbound and upperbound of commit timestamps for all contending transactions. In case of statically assigned timestamps, conflicts arise since large number of transactions are competing to access same set of data simultaneously and the timestamps are fixed, leading to higher aborts.

### 2) Varying Concurrency

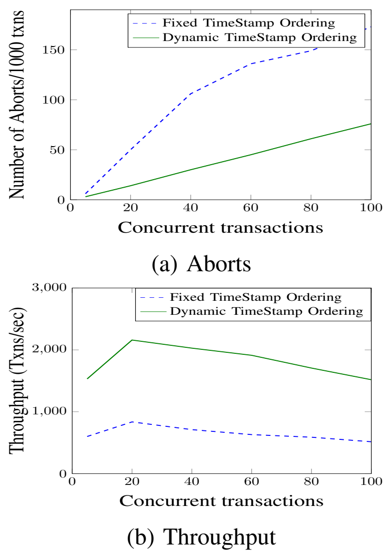
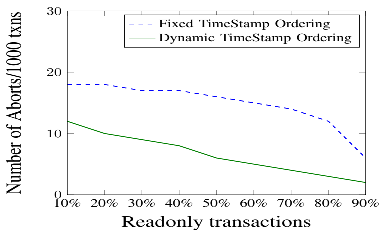


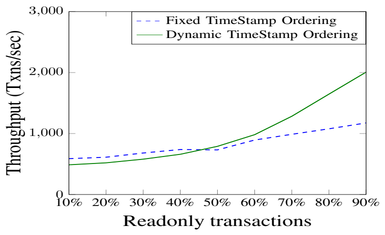
Fig. 5: Varying concurrency with 99% contention

Next, the performance of the concurrency control protocols is analyzed under varying concurrent number of transactions. The experiment is performed with the high contention ratio of 99:01 and 80% read-only transactions. Figure 5 illustrates the results. When the workload is highly concurrent (90 concurrent threads), fixed timestamp ordering has more than 2x aborts compared to the number encountered with dynamic timestamp ordering. This difference is seen because when many transactions are concurrently accessing small set of data, the dynamic timetamping technique adjusts the commit timestamp bounds of concurrent transactions, allowing many of those transactions to commit which would have failed otherwise.

*3) Varying Read-only ratio*



(a) Aborts



(b) Throughput

Fig. 6: Varying read-only ratio with 90% contention

In the next experiment, the performance of the concurrency-control protocols is analyzed under varying read-only transaction ratio. With 90% contention and 50 concurrent threads, increase in the read-write transactions (decrease in read-only transactions) leads to an increase in the number of aborts for both protocols. Figure 6a illustrates that in the case of increased write transactions, there is high number of aborts for the default pessimistic fixed timestamp ordering scheme as compared to the dynamic timestamp ordering. Figure 6b captures the throughput for decreasing read-only transactions percentage from 90% to 10%. At 10% read-only transactions, fixed timestamp ordering is performing slightly better compared to dynamic timestamp ordering due to the reduced bookkeeping in the former approach. But with higher ratio of read operations, the aborts decrease and throughput increases for both the techniques.

## VIII. Concluding Remarks

A dynamic timestamp ordering based concurrency-control scheme modeled on the MaaT protocol is integrated in the transaction processing mechanism of CockroachDB. Rather than allocating a fixed commit timestamp at the start of the transaction, the commit timestamp is dynamically allocated, in such a way that the commit timestamp can be made to fit on a logical serializable timeline. Code changes were made to integrate the protocol in the existing transaction processing codebase in CockroachDB.

Initial results show that the integration of dynamic timestamp ordering in CockroachDB leads to a decrease in the number of aborts. Integrating dynamic timestamp ordering based concurrency-control in a fully-featured database like CockroachDB needed extensive changes at multiple layers. We have implemented and evaluated the changes, and have open-sourced them on Github. We plan to extend and integrate the proposed optimizations for read-only transactions and other mechanisms of reducing the overhead of dynamic timestamping.

To the best of our knowledge, this is the first evaluation of dynamic timestamp allocation technique in a commercial and production cloud database such as CockroachDB.

## IX. Acknowledgments

This work is supported by a grant from Huawei and NSF grant CSR 1703560.

## References

[1] Benchmark for evaluating Dynamic Timestamp ordering based Concurrency Control in CockroachDB. https://github.com/vaibhavarora/cockroach/tree/dynamic_timestamp_benchmark.

[2] CockroachDB. https://www.cockroachlabs.com/.

[3] CockroachDB Design. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[4] CockroachDB Repository with Source-code Changes. https://github.com/vaibhavarora/cockroach/tree/dynamic_timestamp_ordering.

[5] RocksDB. http://rocksdb.org/.

[6] D. Agrawal and A. El Abbadi. Locks with constrained sharing. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 85–93. ACM, 1990.

[7] D. Agrawal and S. Sengupta. *Modular synchronization in multiversion databases: Version control and concurrency control*, volume 18. ACM, 1989.

[8] V. Arora, T. Mittal, D. Agrawal, A. El Abbadi, X. Xue, Zhiyanan, and Zhujianfeng. Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols. In *Usenix HotCloud*, 2017.

[9] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10, 1995.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[11] C. Boksenbaum, M. Cart, J. Ferrié, and J.-F. Pons. Certification by intervals of timestamps in distributed database systems. In *Proceedings of VLDB*, pages 377–387. Morgan Kaufmann Publishers Inc., 1984.

[12] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):20, 2009.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. pages 143–154, 2010.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, 2005.

[15] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004.

[16] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of VLDB*, pages 1263–1274, 2007.

[17] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proceedings of VLDB*, 7(5):329–340, 2014.

[18] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[19] D. P. Reed. Naming and synchronization in a decentralized computer system. 1978.

[20] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM TODS*, 19(1):117–165, 1994.

[21] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM TODS*, 20(3):325–363, 1995.

[22] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of ACM SIGMOD*, pages 1–12, 2012.

[23] M. Yabandeh and D. Gómez Ferro. A critique of snapshot isolation. In *Proceedings of ACM EuroSys*, pages 155–168, 2012.

[24] C. Yan and A. Cheung. Leveraging lock contention to improve oltp application performance. *Proceedings of VLDB*, 9(5):444–455, 2016.

[25] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of ACM SIGMOD*, pages 1629–1642, 2016.

[26] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, et al. Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of VLDB*, 9(6):504–515, 2016.