ConfluxDB: Multi-Master Replication for Partitioned Snapshot Isolation Databases

Prima Chairunnanda, Khuzaima Daudjee, M. Tamer Özsu
Cheriton School of Computer Science, University of Waterloo
prima.ch@gmail.com, {kdaudjee, tamer.ozsu}@uwaterloo.ca

ABSTRACT

Lazy replication with snapshot isolation (SI) has emerged as a popular choice for distributed databases. However, lazy replication often requires execution of update transactions at one (master) site so that it is relatively easy for a total SI order to be determined for consistent installation of updates in the lazily replicated system. We propose a set of techniques that support update transaction execution over multiple partitioned sites, thereby allowing the master to scale. Our techniques determine a total SI order for update transactions over multiple master sites without requiring global coordination in the distributed system, and ensure that updates are installed in this order at all sites to provide consistent and scalable replication with SI. We present ConfluxDB, a PostgreSQL-based implementation of our techniques, and demonstrate its effectiveness through experimental evaluation.

1. INTRODUCTION

Snapshot Isolation (SI) has become a popular isolation level in database systems. Scaling-up a database system usually involves placing the data at multiple sites, thereby adding system resources over which the workload can be distributed to improve performance. Providing SI over such a distributed system is challenging since a global, total order for update transactions needs to be determined so that updates can be installed in this order at every site. While it is relatively easy to determine a total SI order for consistent installation of updates in a lazily replicated system, this usually requires execution of update transactions at only one (master) site.

There have been proposals to provide global SI for both partitioned and replicated databases [5, 12, 30, 27, 18, 1, 19, 13, 26, 22]. However, these proposals do not consider how to scale-up under lazy replication a primary (single-site) database system without relying on physical clock synchronization or a centralized site or component to determine a global update order in the distributed system.

Having a single site or centralized component at which all update transactions execute is a restriction with significant drawbacks. As the workload scales-up, an increasing update load is placed on the single site. This leads to performance degradation as this site becomes a bottleneck, which can be seen in our experimental results in Section 6.

Prior work that addresses this bottleneck problem partitions the update workload a priori or restricts transactions to updating data at a single site [6, 7, 10]. A more conservative approach is the prediction of conflicts between transactions to partition the primary database a priori into conflict classes [17]. In this paper, we propose techniques to provide global SI over partitioned and distributed databases. To the best of our knowledge, this is the first proposal that avoids the above restrictions.

We present ConfluxDB, a PostgreSQL-based prototype that (i) implements a set of techniques that support scaling-up a database through partitioning while providing global SI by determining a global order for distributed update transactions (ii) uses a novel log merging based replication scheme of the partitioned database that allows transactions to see snapshots consistent with global SI at the replicas.

In ConfluxDB, distributed update transactions can execute on a database partitioned over multiple primary sites and are not restricted to updating data at a particular site. ConfluxDB's novel log-merging replication solution merges multiple update streams from the partitioned sites into a single, unified stream that is consistent with the SI ordering over the partitioned primary sites. The unified stream is then used to install updates in the same order at sites that hold database replicas to achieve global SI ordering for all transactions in the distributed system. Update transactions execute under the well-known two-phase commit (2PC) protocol at the primary sites and updates are propagated lazily to secondary sites. Our choice of 2PC stems from it being the most widely used protocol for coordinating transactions in distributed database systems [3]. Using the TPC-C and TPC-W benchmarks, we conduct performance experiments that demonstrate the scalability and efficiency of ConfluxDB.

2. SYSTEM OVERVIEW & BACKGROUND

2.1 System Architecture

In ConfluxDB, the master or *primary database* is partitioned over one or more sites (called *primary sites*), as shown in Figure 1. No restrictions are placed on how the primary database is partitioned. The primary sites comprise the *primary cluster*. A replica of each primary database partition is held at each *secondary site* (in Section 4.1.3 we show that this is not a strict requirement, but it greatly simplifies the presentation). Each site in ConfluxDB hosts an autonomous DBMS in a shared-nothing environment that locally guarantees SI.

Proceedings of the VLDB Endowment, Vol. 7, No. 11

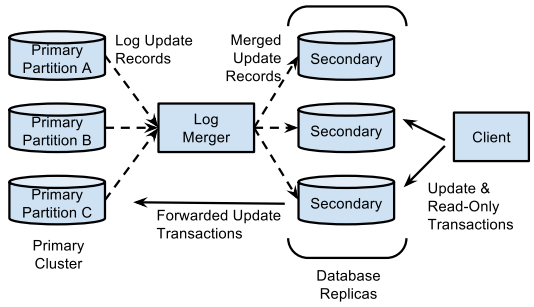

Figure 1: ConfluxDB Architecture

Clients connect to a secondary site and submit transactions for processing [11, 10]. We assume that a client's read-only transactions can be distinguished from its update transactions. Read-only transactions are executed at the secondary site to which they are submitted. Update transactions are forwarded by the secondary sites to the primary cluster for execution.

Update transactions execute at the primary sites using the 2PC protocol [3]. Update information can be extracted from the database logs, e.g. through log sniffing [16]. The updates from each log are merged into a single stream using the log merging algorithm described in Section 4 after which they are propagated lazily to the secondary sites in an order consistent with the order at the primary sites. At each secondary site, propagated updates are placed in a FIFO update queue. An independent process at each secondary site removes propagated updates from the update queue and applies them to the local database copy.

The architectural separation of responsibility between the primaries and secondaries gives users more flexibility to play to each of their strengths. The primaries can process update transactions common in OLTP applications while secondaries can serve OLAP-like queries [21, 25]. Popular benchmarks such as TPC-W and TPC-C that are modeled after real-world scenarios also contain such mixed workloads. A nice property of the ConfluxDB architecture is that more secondary sites can be added as the read-only workload scales up. With higher update transaction ratios, as is the case with OLTP workloads such as TPC-C, more primaries can be added to scale-up the system thereby increasing scalability and performance through this increase in system resources.

2.2 Terminology and Background

We now define the terminology relevant to update transactions at the primary cluster presented in Section 3. We introduce terminology relevant to read-only transactions and transactions that apply updates at secondary sites later in Section 4.

Transaction i will be denoted as T_i. The *participating sites* of T_i is the collection of sites T_i reads data from or writes data to. Throughout the paper, we use the term participating site s as any site from among the participating sites of a specified transaction. If there is only one participating site, then T_i is a *local transaction*; otherwise, it is a *global transaction*. Transaction T_i will have a *subtransaction* T_i^s at each participating site s.

The *coordinator* site for a transaction T_i is denoted as $coord_i$, and this will be the first participating primary site that starts its subtransaction of T_i. If T_i is a global transaction then $coord_i$ will also coordinate the 2PC.

We are specifically interested in the following events in the database system: *begin transaction*, *read* (item), *write* (item), and *commit transaction*. As is common, the *begin transaction* and *commit transaction* events are totally ordered at each site and we capture this order using the *happened-before* relation, where $e \prec e'$ means that event e happened before event e'. Existing mechanisms, such as Lamport timestamps [20], can be used to derive this order.

The begin and commit events of a subtransaction T_i^s are denoted as $begin(T_i^s)$ and $commit(T_i^s)$ respectively. A transaction T_i is said to have *happened-before* T_j ($T_i \prec T_j$) if and only if T_j sees T_i's effects but T_i does not see T_j's effects, i.e., $T_i^s \prec T_j^s$ if and only if $commit(T_i^s) \prec begin(T_j^s)$. If neither $T_i \prec T_j$ nor $T_j \prec T_i$ is true, then these transactions are *concurrent* with respect to each other.

In considering pairs of transactions, we use the notion of whether one transaction is dependent on another to derive an ordering of these transactions. We will use this definition to formulate global SI and to describe our protocol in Section 3.

Definition 1. Let s be some participating site of T_i, and $RS(T_i^s)$ and $WS(T_i^s)$ be the read-set and write-set, respectively, of T_i^s. The predicate $dependent^s(T_i, T_j)$ is true if and only if $(RS(T_i^s) \cap WS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap RS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap WS(T_j^s) \neq \emptyset)$.

Since at times we will be interested to know whether we have a dependent pair of transactions T_i and T_j at any of their participating sites, we define the non site-specific version of the dependent predicate as follows:

Definition 2. The predicate $dependent(T_i, T_j)$ is true if there exists some site t participating in T_i and T_j where $dependent^t(T_i, T_j)$ is true.

ConfluxDB relies on the update transactions in the workloads (in particular, TPC-C and TPC-W used for our experiments) to touch only rows with a particular key (e.g. customer id, warehouse id, etc.) making it straightforward to extract the read and write sets from their SQL statements during runtime. Techniques have been proposed if more sophisticated read/write set extraction is desired, e.g., [34].

2.3 Snapshot Isolation

Snapshot Isolation originates from multiversion concurrency control (MVCC) where multiple versions of the same data item may exist at any one time [2]. We use x_i to refer to the version of database item x installed by transaction T_i. A write operation by transaction T_i will always write its own version of x into the database (i.e. $w_i[x_i]$), but it may read data from any transaction, including itself (i.e. $r_i[x_j]$).

Under SI, a transaction T_i is guaranteed throughout its lifetime to read data items from the snapshot (or database state) S_i obtained when T_i starts. Specifically, this includes only changes made by all transactions T_j such that $commit(T_j) \prec begin(T_i)$. Any modifications made to the data items after S_i is read will not be visible to T_i, unless of course if T_i made the changes itself. All modifications made by a transaction become visible when the transaction commits. Concurrent transactions are not allowed to write into the same items, and when that happens, only the first commit is successful. This is called the "First-Committer Wins" rule[1] [2].

In practice, most implementations use the relative order of *begin* and *commit* events to decide whether two transactions are concurrent. More specifically, T_i and T_j are concurrent if and only if $begin(T_j) \prec commit(T_i)$ and $begin(T_i) \prec commit(T_j)$.

[1]Our implementation follows a variant rule used by PostgreSQL where the first updater to commit wins. In this rule, if T_i tries to update a row that has been updated by a concurrent transaction T_j, then T_i will be blocked until T_j's outcome is known, thereby guaranteeing that T_j is given chance to commit first.

Since concurrent transactions under SI do not see each other's effects, T_i will see the effects of some transaction T_k if and only if $commit(T_k) \prec begin(T_i)$. While systems that employ multi-versioning by definition store multiple versions of data items, the default is for T_i to see the latest committed version of data item x at the time T_i begins. In other words, T_i cannot request to see arbitrary older versions of a data item.

3. COLLABORATIVE GLOBAL SNAPSHOT ISOLATION

In this section, we present our proposal, Collaborative Global Snapshot Isolation (CGSI) to extend SI to a distributed environment. The CGSI protocol allows global transactions to see consistent snapshots while guaranteeing global snapshot isolation to transactions executing over the primary sites.

3.1 Global Snapshot Isolation

When a database is partitioned over multiple sites, the challenge is to ensure that SI holds *globally* for concurrent, distributed, transactions executing over the partitioned database sites. These transactions will independently issue *begin* and *commit* events at multiple autonomous sites. It is undesirable to rely on global clock synchronization, which can be expensive to achieve, to assign a total order on events in the distributed system.

In particular, for any transaction pair T_i and T_j running at sites s and t, if T_i^s does not see T_j^s's effects, we do not want T_i^t to see T_j^t's effects either. Selecting a snapshot S_i for T_i to access at all sites that host the partitioned database is a challenge. For example, it is possible that when T_i starts at site s, T_j is executing, thus $begin(T_i^s) \prec commit(T_j^s)$ and T_i^s will not see T_j^s's effects. Meanwhile, when T_i starts at site t, T_j may have executed and committed, which means that when T_i^t starts, it will not see the effects of T_j^t. In this context, we formalize the concept of a *consistent* snapshot through the following definition:

Definition 3. Snapshot S_i seen by transaction T_i is a consistent snapshot if for any transaction pair T_i and T_j executed at different sites s and t, it is not the case that $dependent(T_i, T_j) \wedge (T_i^s \prec T_j^s) \wedge (T_i^t \not\prec T_j^t)$.

Thus, before T_i is allowed to commit, the system needs to ensure that snapshot S_i^s read by T_i at site s is consistent with snapshot S_i^t read by T_i at site t, i.e. both snapshots should have been installed by the same committed transaction at all sites before T_i starts. In doing so, we exclude from consideration the set of transactions T_j where $\neg dependent(T_i, T_j)$ as in that case it is irrelevant whether one transaction can or cannot see the effect of the other. There are SI variants that allow transactions to request a snapshot of the database as of a specific time in the past [24]. However, they usually use real (physical) time to define a snapshot, which is challenging to enforce, manage, and synchronize in a distributed system.

The database state seen and committed by update transactions at the primary sites corresponds to the transactions' *begin* and *commit* events. Further, as can be seen from Definition 3, the consistency of these events' ordering across the primary sites also directly determines whether transactions see consistent snapshots. In general, to ensure that each and every committed transaction in the system sees a consistent snapshot, the following conditions (or invariants) must hold for each transaction pair T_i and T_j:

I1. If $dependent(T_i, T_j) \wedge (begin(T_i^s) \prec commit(T_j^s))$, then $begin(T_i^t) \prec commit(T_j^t)$ at all participating sites t.

I2. If $dependent(T_i, T_j) \wedge (commit(T_i^s) \prec begin(T_j^s))$, then $commit(T_i^t) \prec begin(T_j^t)$ at all participating sites t.

I3. If $commit(T_i^s) \prec commit(T_j^s)$, then $commit(T_i^t) \prec commit(T_j^t)$ at all participating sites t.

I1 and I2 are important to consistently determine whether T_j that satisfies $dependent(T_i, T_j)$ should be visible to a transaction T_i. For example, if site s observes the order $begin(T_i^s) \prec commit(T_j^s)$ while another site t observes the order $commit(T_j^t) \prec begin(T_i^t)$, then T_j happened before T_i at site t, but they are either concurrent at site s, or T_i happened before T_j at site s.

I3 is important to prevent two different snapshots from seeing partial commits that are incompatible with each other. Assume that the value of data items x and y are originally x_0 and y_0, respectively. Consider the order $commit(T_j^s) \prec begin(T_p^s) \prec w_k[x_k] \prec commit(T_k^s) \prec begin(T_q^s)$ at site s, and the order $commit(T_k^t) \prec begin(T_q^t) \prec w_j[y_j] \prec commit(T_j) \prec begin(T_p^t)$ at site t. Supposing that both T_p and T_q want to read x and y, T_p will read x_0 and y_j, implying $T_j \prec T_k$. On the other hand, T_q will read x_k and y_0, implying $T_k \prec T_j$. Clearly, both conditions cannot be true at the same time, and at least one of them needs to be aborted. I3 avoids the occurrence of such aborts.

If $\neg dependent(T_i, T_j)$, any possible ordering of $begin(T_i^s)$ and $commit(T_j^s)$ will not cause inconsistency in S_i or S_j. This is trivially true in the absence of any other transactions in the system. When there is some other transaction T_k such that $dependent(T_i, T_k)$ and $dependent(T_k, T_j)$, but $\neg dependent(T_i, T_j)$, we need to make sure that at least one of them is aborted. We do this by detecting the inconsistency between S_i and S_k, and also between S_k and S_j (details about this detection are covered in Section 3.2.1).

An important consequence of the absence of dependencies is that if the majority of transactions touch only a small number of non-intersecting tuples, it allows their *begin* events to be executed in any relative order with respect to other transactions' *commit* events, greatly increasing parallelization opportunities. Note that the relative order among *commit* events is still important even if the transactions involved are data-independent for the same reason discussed for condition I3.

3.2 CGSI Algorithms

We now describe how CGSI enforces global snapshot isolation through consistent snapshots. Without loss of generality, consider a global transaction T_i running at two sites s and t. CGSI does not require a priori knowledge of the participating sites of T_i. CGSI lets $begin(T_i^s)$ proceed independently from $begin(T_i^t)$. Before T_i can commit, checks are made to verify that the snapshot seen by T_i^s is consistent with the snapshot seen by T_i^t and vice versa. We refer to this check as *certification*. If an inconsistency is found, T_i is aborted.

CGSI performs certification over all participating sites of T_i since each participating site s knows only about the snapshot seen by T_i^s but not the snapshot seen by T_i^t running at another participating site t. With this strategy, CGSI is able to avoid having a potential bottleneck in a centralized certifier. Moreover, the distribution of the certification process spreads the load across the sites.

3.2.1 Distributed Certification Algorithm

To provide a consistent snapshot, I1–I3 need to hold. We describe the enforcement of I1 and I2 next, followed by how I3 holds. To preserve I1 and I2, it is necessary for CGSI to know which transactions have been committed, and which are still active. At each site s, CGSI keeps sets of active and committed transactions, and their begin and commit timestamps respectively. We call these sets $active^s$ and $committed^s$. Prior to committing T_i^s, CGSI conducts the certification at site s in the following manner. Each of

the transactions in $active^s$ and $committed^s$ is categorized into one of two sets with respect to T_i: concurrent transactions or serial transactions, denoted as $concurrent^s(T_i)$ and $serial^s(T_i)$ respectively. Concurrent transactions are those T_j where $begin(T_i^s) \prec commit(T_j^s)$; otherwise they are serial. Thus, T_i completely sees the effects of T_j if and only if $T_j \in serial^s(T_i)$. If there exists a T_k such that $dependent(T_i, T_k) \wedge (T_k \in serial^s(T_i)) \wedge (T_k \in concurrent^t(T_i))$ for some participating sites s and t, certification will fail as S_i is deemed to be inconsistent.

The **distributed certification algorithm** is shown in Algorithm 1. For clarity, we present the algorithm independently from the commit protocol, and defer the discussion of its integration with 2PC.

Algorithm 1 Distributed Certification Algorithm

1. $coord_i$ requests each participating site s to send the transactions in $concurrent^s(T_i)$.
2. Participating site s responds to $coord_i$ with its $concurrent^s(T_i)$.
3. All responses of $concurrent^s(T_i)$ are merged by $coord_i$ into a single set $gConcurrent(T_i)$. If $T_j \in gConcurrent(T_i)$, then T_j^t is concurrent with T_i^t for at least one participating site t.
4. $coord_i$ sends $gConcurrent(T_i)$ to all participating sites.
5. Each participating site s checks if there exists a T_j, such that $dependent^s(T_i, T_j) \wedge (T_j \in gConcurrent(T_i)) \wedge (T_j \in serial^s(T_i))$. If such T_j is found, it is proven that T_i is seeing an inconsistent snapshot S_i, and site s should send a negative certification response. Otherwise, site s sends a positive certification response.
6. $coord_i$ aggregates the certification results from all participating sites. If any participating site replies negatively, it means at least one pair of T_i's subtransactions see an inconsistent snapshot, and T_i should be aborted. Otherwise, T_i can proceed to commit.

THEOREM 3.1. *For a committing transaction T_i, if there exists a transaction T_j satisfying $dependent(T_i, T_j)$, such that $(T_j \in concurrent^s(T_i)) \wedge (T_j \in serial^t(T_i))$ for some sites s and t, the distributed certification algorithm (Algorithm 1) will detect T_i as seeing an inconsistent snapshot.*

PROOF. Let us assume that there is a transaction T_j where $(T_j \in concurrent^s(T_i)) \wedge (T_j \in serial^t(T_i))$ for some sites s and t, but the algorithm determines that T_i sees a consistent S_i. If the algorithm does not detect the inconsistency, it means that all certification responses from the sites are positive. Now recall that $T_j \in serial^t(T_i)$. If site t responded positively during the certification, it means site t could not find a data-dependent T_j, such that $(T_j \in gConcurrent(T_i)) \wedge (T_j \in serial^t(T_i))$. Then, in order to avoid the inconsistency from being detected, it needs to be the case that $T_j \notin gConcurrent(T_i)$. However, because $gConcurrent(T_i)$ is the union of all $concurrent^s(T_i)$ from all participating sites s, there cannot be some site s such that $T_j \in concurrent^s(T_i)$, a contradiction. □

The distributed certification algorithm is integrated into 2PC as part of "preparing" the transaction for commit. Step 1 of Algorithm 1 is initiated with the *prepare* message to the participants, while the certification response in Step 5 is piggybacked with the *prepare_ack* message to the coordinator. Step 6 is performed in conjunction with the 2PC global decision. The remaining steps (Steps 2-4) are internal to CGSI and do not change the 2PC state of the transaction.

A new transaction T_j that starts at site s after the site executed Step 2 of Algorithm 1 but before T_i commits is not required to be added to $concurrent^s(T_i)$. This is because at that point in time, T_i is already in *prepared* state and thus T_j could not have affected the snapshot S_i seen by T_i at site s. Though, T_i still needs to be added to $concurrent^s(T_j)$.

Several MVCC based systems, including PostgreSQL, already track the list of concurrent transactions to determine tuple visibility in a snapshot. What remains is to track the full set of committed transactions, which typically needs to be recovered from the logs. To avoid the need to consult the logs, CGSI keeps track of this set separately. To prevent the set from growing forever, we employ an expiration strategy which limits the size of the committed transactions set at each site and evicts the oldest transactions in the set if its size exceeds the limit. We describe this expiration strategy in more detail in [9] with the necessary modifications to the certification algorithm and the proof of its correctness.

The participating sites of T_i need not be known beforehand, because $coord_i$ will include in the set of participating sites each primary when data at that site is accessed by T_i. This is also when $coord_i$ can infer that T_i is a global transaction. Once the commit request is made to $coord_i$, 2PC is initiated and all participating sites vote on whether T_i can be safely committed without violating global SI. $coord_i$ makes the final decision and all participating sites commit or abort in accordance with this decision. Another important characteristic of ConfluxDB is that the outcome of a transaction T_i is determined only by the sites that are participating in executing it. Our protocol obviates the need for a single, centralized, coordinating site to decide whether a transaction should be committed or aborted.

Table 1: Some Frequently Used Notation

Notation	Description
$active^s$	Active transactions at site s
$committed^s$	Committed transactions at site s
$serial^s(T_i)$	Transactions happened before T_i at site s
$concurrent^s(T_i)$	Transactions concurrent to T_i at site s
$gConcurrent(T_i)$	Transactions that are concurrent to T_i at at least one participating site
$event\_clock^s$	The event clock at site s
$gct(T_i)$	The global commit timestamp of T_i

3.2.2 Transaction Commit Ordering

We now consider how CGSI preserves I3. In systems with a centralized controller, the responsibility to assign a deterministic global order usually falls to that controller. Since we do not rely on such a centralized component, all sites collaboratively work to achieve a global ordering. To avoid communication overhead between sites, we piggyback information onto existing messages where possible.

Under CGSI, each site s stores a monotonically increasing *event clock* (denoted as $event\_clock^s$), which is communicated and updated with each 2PC message in the following manner (we omit database actions to focus on the clock manipulation):

1. When the coordinator c sends the prepare message for a global transaction, the coordinator piggybacks its $event\_clock^c$ value onto the broadcasted *prepare* messages.

2. On a participant p receiving *prepare* message from coordinator, the participant updates its $event\_clock^p$ with the maximum value from $event\_clock^p$ and the one carried by the *prepare* message. Then, the participant sends back a *prepare_ack* piggybacking the updated $event\_clock^p$.

3. On coordinator c receiving *prepare_ack* message from participants, the coordinator updates its $event\_clock^c$ with the maximum value from $event\_clock^c$ and the one carried by the *prepare_ack* message. Once all participants have replied with positive *prepare_ack*, the coordinator atomically increases its $event\_clock^c$ and globally decides to commit the transaction. The incremented $event\_clock^c$ becomes the *commit timestamp* of the transaction.

4. On a participant p receiving *commit* message from coordinator, the participant updates its $event\_clock^p$ with the maximum value from $event\_clock^p$ and the one carried by the *commit* message.

All global transactions coordinated by a particular site will have a total order defined on them by virtue of atomically increasing the event clock. Thus, there cannot be two global transactions coordinated by site s with the same commit timestamp. Across sites, two global transactions may still be assigned the same commit timestamp. To break ties, we can choose some identifier unique to each transaction coordinator, such as IP or hostname. This scheme is not unlike the timestamp generation methodology used in [32]. Together, the commit timestamp and the coordinator's identifier form the *global commit timestamp* of a transaction.

Definition 4. The *global commit timestamp* of a transaction T_i, denoted as $gct(T_i)$, is an ordered pair $\langle commit\_ts(T_i), id(T_i)\rangle$, where $commit\_ts(T_i)$ is the commit timestamp of T_i, and $id(T_i)$ is the identifier of $coord_i$.

Definition 5. For every two distinct transactions T_i and T_j, we define $T_i \prec T_j$ if and only if $commit\_ts(T_i) < commit\_ts(T_j)$, or $commit\_ts(T_i) = commit\_ts(T_j)$ and $id(T_i) < id(T_j)$.

The second case in Definition 5 is used to break ties in the case of assigned commit timestamps that are the same. The database commit actions are executed strictly in the order of the commit timestamps. If each site locally commits its global transactions in the order defined by $\prec$, then for any global transaction T_i, there will always exist a snapshot of the database such that a transaction $\{T_k | T_k \preceq T_i\}$ has been committed and no transaction $\{T_k | T_i \prec T_k\}$ has been committed. As a consequence of this ordering requirement, there may be times when some transaction commits need to be postponed. More formally, site s has to postpone the commit of global transaction T_i until site s is sure that there cannot be any other uncommitted global transaction T_j in which s is a participating site with $T_j \prec T_i$.

To accomplish this, CGSI maintains a priority queue data structure called PreparedQueue that contains prepared global transactions. Each prepared global transaction has a $highest\_clock$ attribute, which stores the highest event clock value ever received regarding the global transaction. Additionally, a $ready\_to\_commit$ flag is used to indicate whether the site has received global-commit decision about the transaction from the coordinator, at which point the transaction is ready to be written to the database. When this flag is true, the stored $highest\_clock$ value corresponds to the transaction's $commit\_ts$ because the event clock value carried by a transaction's global commit message will always be at least one greater than the highest event clock value carried by any of its *prepare* or *prepare_ack* messages. The identifier of the coordinator is also saved so that the priority queue can order the global transactions by their global commit timestamps using the $\prec$ total ordering. This way, the root of PreparedQueue will be the transaction with the lowest global commit timestamp.

PROPOSITION 3.2. *When the global transaction T_i present at the root of the priority queue of site s has $ready\_to\_commit$ flag set to true, there cannot be any other uncommitted global transaction T_j with participating site s where $T_j \prec T_i$.*

PROOF. There are four cases to consider, depending on whether site s is the coordinator of T_j, and whether 2PC has been started for T_j:

1. Suppose $s = coord_j$ and T_j has been prepared. Hence, we know that $commit\_ts(T_i) \leq event\_clock^s$. As the coordinator increases its event clock value on commit, it will be the case that $commit\_ts(T_j) \geq event\_clock^s + 1$. Regardless of the coordinator's identifier, $T_i \prec T_j$.

2. Suppose $s \neq coord_j$ and T_j has been prepared. Since we find T_i at the root of the priority queue, it means for every other global transaction T_k in the queue, its $highest\_clock$ value must be at least $commit\_ts(T_i)$. As T_j has been prepared, then T_j must be in the queue, and its $highest\_clock$ value must refer to the event clock piggybacked in the *prepare_ack* message. Because the coordinator updates its event clock with the piggybacked one, the coordinator will eventually assign a $commit\_ts(T_j)$ value of at least $highest\_clock + 1$. Regardless of the coordinator's identifier, $T_i \prec T_j$.

3. Suppose $s = coord_j$ and T_j has not been prepared yet. When site s prepares T_j, the $highest\_clock$ will be at least $event\_clock^s$. Following the same argument as case (1), $commit\_ts(T_j)$ will be at least $event\_clock^s + 1$, thus $T_i \prec T_j$.

4. Suppose $s \neq coord_j$ and T_j has not yet been prepared. Then, $commit\_ts(T_i) \leq event\_clock^s$. When the *prepare* message for T_j arrives, site s will piggyback an event clock value of at least $event\_clock^s$. Eventually, the value of $commit\_ts(T_j)$ will be at least $event\_clock^s + 1$, thus $T_i \prec T_j$. □

Transaction T_i is not immediately committed after the global decision to commit has been made under 2PC. First, CGSI records the commit timestamp in $highest\_clock$, flips $ready\_to\_commit$ flag to true, and then inserts T_i into the PreparedQueue. The actual commit operations are issued from a separate committer thread which continuously monitors the root of PreparedQueue for any transaction with the $ready\_to\_commit$ flag set to true.

THEOREM 3.3. *If each primary site runs CGSI and enforces SI locally, update transactions will run under global SI.*

PROOF. The committer thread enforces I3 across all sites by processing commit in a deterministic order of the transactions' global commit timestamp. Therefore, two sites participating in the same transactions cannot commit them in different orders. Furthermore, the First-Committer Wins rule at each site will ensure there is no concurrent write by two different transactions to the same item. On a shared-nothing database, this also holds globally.

Next, since the local concurrency control enforces SI locally, a transaction T_i^s at site s reads from snapshot S_i^s obtained when T_i^s starts. By Theorem 3.1, T_i^s is allowed to commit only if every subtransaction T_i^t reads from a consistent snapshot that satisfies I1 and

I2, where t is in the set of participating sites of T_i. Consequently, globally, T_i sees a consistent snapshot S_i that satisfies SI.

Since both of these are true for all committed transactions[2] at the primaries, these transactions execute under global SI. □

Normally, 2PC participants are notified of the global decision after the transaction has been successfully committed at the coordinator site. In CGSI, the global decision and the coordinator commit do not necessarily happen at the same time. The coordinator sends the *global_commit* message immediately after the global decision has been made without waiting for the actual commit at the coordinator's site. The coordinator must also durably log the global commit decision separately from the actual commit operation so that in the event that the coordinator fails to commit, we can recover from the coordinator's transaction failure using the same error handling mechanism as when a participant fails to commit.

4. REPLICATION AND LOG MERGING

In this section, we describe how updates of committed transactions over the primary sites that hold the partitioned database are captured in the database logs. We then present an algorithm for merging the updates from these database logs to generate a single stream of updates that is consistent with the global SI order over the partitioned primary database sites.

We use a physical log-based approach for maintaining replicas, where log information of updates from committed transactions are transferred to the replicas. Typically, database systems store the sequence of operations performed for a transaction in the log so that the operations can be undone or redone to preserve ACID properties in the event of failure. Log records can be appended only to the end of the log so that it represents a valid sequential execution of operations. In general, it is not safe to replay log records out of order, even if they concern different transactions. This comes as a consequence of the tight coupling with the physical database layout.

4.1 Merging Log Streams from Multiple Masters

A straightforward solution to replicate a partitioned database is to replicate each partition at some secondary site. A disadvantage of this approach, however, is that running analytical or multi-join read-only queries involving multiple partitions will incur communication and coordination overhead as transactions will need to access multiple secondary sites. To avoid incurring this cost, we fully replicate the set of primary database partitions at each secondary site, i.e. each secondary site holds a copy of all the tables from every primary partition. We propose a novel log stream merging solution to maintain replicas of all partitions under one database instance per secondary site so that queries involving these partitions can be serviced locally. Our solution merges log streams from all masters though it can also be used to merge log streams selectively from only a subset of them if desired.

A table can be split over multiple partitions so that each partition hosts a distinct, non-overlapping chunk of the table. The chunk still appears as an ordinary table to the partition, but we can do global queries and updates by utilizing the "table group" feature available in popular database systems (e.g. MySQL and PostgreSQL [29]).

Our solution merges multiple log streams into a single *unified stream*, as shown in Figure 2. Each master server, which processes update transactions, generates a log stream. The log streams are

[2]Aborted transactions cannot cause any visible changes, thus we can safely exclude them from consideration.

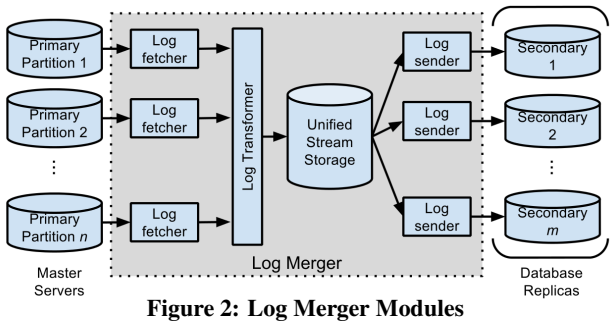

Figure 2: Log Merger Modules

then transferred to the *log merger* component, which reads log streams from all master servers and merges them into the unified stream. It consists of three main modules with distinct responsibilities. The *log fetcher* module uses multiple log fetcher threads to continuously fetch log records from the master servers. There is one log fetcher thread for each master server, and each thread operates independently from the other. This allows us to exploit parallelism in reading the log records from multiple master servers. Next, these log streams are fed into the *log transformer* to be processed, reordered, and merged into the unified stream, which is then durably written to the local persistent storage. Finally, the *log sender* module reads the unified stream from persistent storage and sends it to the replicas, one log sender thread per replica.

Each replica that receives the unified stream of updates is a database instance that continually replays log records as they become available. As the replicas serve only read-only transactions, they do not produce log records of their own. We have considered the option where a log merger exists at each replica, thus each replica can fetch and merge logs independently. However, we found this to be less scalable as it removes a portion of resources on each replica to repeat the log merging.

Our log merging solution has the following characteristics. First, replicas can replay the unified stream successfully as if it came from a single master. Second, we ensure that the replayed log records will not violate global SI. Finally, we avoid transaction inversion [12] such that subsequent transactions submitted by the same client see at least the preceding transaction's effects.

4.1.1 Forming A Unified Stream

The log streams are processed in round robin fashion. We take one record from a stream, process the record and then proceed to the next stream. Since a log record describes an operation performed by a transaction, it contains the *transaction identifier* (TId) of the transaction performing the operation, the type of operation itself, and the *object identifier* (OId) on which the operation is applied. Although identifiers are unique within a stream, they may collide between different streams. We resolve this collision in the unified stream as follows.

Each record in the unified stream has a TId field. To avoid ambiguity, we use the term *ReplicaTId* when specifically referring to the TId used in the unified stream. To resolve TId collision, for each master server, we maintain a mapping from the TId appearing in the source stream to its ReplicaTId. Each local transaction will have a unique ReplicaTId mapped for it. For a global transaction, it will appear as different TId on each stream, but those TIds must all be mapped to the same ReplicaTId.

As for OIds, we choose to avoid them from colliding in the first place. Each database item is assigned a unique OId when it is created. We avoid the collision by ensuring each master server assigns the same OId to the same object, that is by populating each master

server with the union of tables, indexes, and other database items from all master servers, but they are all empty. This way, we can simply leave the OIds as they are since the replicas will have consistent references to objects.

4.1.2 Ensuring Snapshot Consistency

We now describe the types of log records that are important to ensure snapshot consistency at the replicas. The master log stream contains a *begin* record to mark the start of a transaction, and a *commit* or an *abort* record to mark the end of a transaction. In addition, a global transaction may also have *prepare*, *abort prepared*, and *commit prepared* (instead of *commit*) records. In the unified stream, however, all global transactions will be reflected as local transactions, i.e. only *abort* and *commit* records will be present. Also, there will be only one *commit* record in the unified stream for a global transaction, even though there are multiple *commit prepared* records. Since a global transaction can update data items at multiple sites, the log merger must wait until it sees a *commit prepared* record from each of the participating sites before it can produce the equivalent *commit* record in the unified stream.

As the CGSI and local concurrency control at each master server guarantee snapshot consistency for update transactions, the multiple log streams can always be merged into a unified stream with a consistent global order. The log transformer needs to infer this order and then ensure that the *commit* records from various streams are replayed in the correct commit order. Recall that CGSI assigns a global commit timestamp to each transaction, and the transactions are committed in order of their timestamps. The log transformer can simply follow the same order for the unified stream; we simply tag the *commit* records with the global commit timestamps of the transactions.

Algorithm 2 Log Transformer Flushing Committed Queue

```
1:  procedure FLUSHCOMMITTEDQUEUE
2:      oldestMaster ← the master with the oldest state
3:      while CommittedQueue not empty do
4:          gTrans ← the root of CommittedQueue
5:          if oldestMaster.state ≺ gTrans.gct then
6:              break
7:          end if
8:          Place the combined gTrans.commitRec in
            unified stream
9:          Remove gTrans from CommittedQueue
10:     end while
11: end procedure
```

Sometimes, the log transformer may need to delay producing a *commit* record for transaction T_i if there is still some other site that can commit a transaction T_j with a lower global commit timestamp. This is because if $gct(T_j) < gct(T_i)$, then *commit* record of T_j must appear before *commit* record of T_i in the unified stream. To handle this, the log records of the master servers contain CGSI state information, such as the event clock and lowest global commit timestamp in the server's commit queue. Also, the log merger will queue all transactions eligible to be committed into a priority queue called the *CommittedQueue*. The root of CommittedQueue is the transaction T_i that has the lowest global commit timestamp among those eligible to commit. By comparing $gct(T_i)$ with the CGSI states of the master servers, the log transformer will be able to determine when it can safely produce the *commit* record of T_i. Pseudocode for this algorithm is shown in Algorithm 2.

Each participating site in a global transaction T_i may assign different TIds for T_i in their respective streams. To correlate among these TIds, $coord_i$ will generate a globally unique transaction id (*gTransId*) for T_i and communicate it to all participants. This value is then included in T_i's *begin* record in each of the participating sites' log streams. Since the gTransId is only used for log merging, it can be stripped off when writing the record to the unified stream.

Algorithm 3 Log Transformer Handling Transactional Records

```
1:  Let GTransList be a list of all global transactions known to
    the merger
2:  procedure PROCESSTXNLOG(master, rec)
3:      if rec.type is begin then
4:          Let gTrans be the entry in GTransList
            where gTrans.gTransId = rec.gTransId
5:          if no such gTrans in GTransList then
6:              Create new entry gTrans in GTransList
7:              gTrans.gTransId ← rec.gTransId
8:              gTrans.replicaTid ← next available ReplicaTId
9:              gTrans.commitRec ← ∅
10:         end if
11:         gTrans.tidByMaster[master] ← rec.tid
12:         return
13:     end if
14:     Let gTrans be the entry in GTransList
        where gTrans.tidByMaster[master] = rec.tid
15:     if rec.type is commit or commit prepared then
16:         gTrans.commitRec ← gTrans.commitRec ∪ rec
17:         if this is the last commit record for gTrans then
18:             gTrans.gct ← rec.gct
19:             Insert gTrans into CommittedQueue
20:         end if
21:     else if rec.type is abort or abort prepared then
22:         if this is the last abort record for gTrans then
23:             Place abort record in the unified stream
24:         end if
25:     else                                  ▷ Other record types
26:         Place rec in the unified stream
27:     end if
28: end procedure
```

Our overall algorithm[3] for processing transactional log records by the Log Transformer (described in Algorithm 3) works as follows. First, the record type is examined. If the type is not one we are interested in, the record is simply placed in the stream. If it is the *begin* record, the gTransId and its TId equivalent will be noted. If it is the first time we see this gTransId, we also initialize the $gTrans$ data structure.

For all other record types, we look up the transaction first by the TId appearing in the record. If it is a *commit* or *commit prepared* record, we combine the data from new commit records with any previously-seen commit records for this transaction. This is necessary because each commit record contains actions related to only one particular site but the commit record in the unified stream must contain the aggregate of all these data. We also count the number of commit records we have seen for this transaction, and if this record is the last one, we update the global commit timestamp of the transaction and place it in the CommittedQueue.

A transaction abort can produce either an *abort* or an *abort prepared* record depending on whether the transaction had entered the "prepared" state. There is no need to combine abort records because the purpose of an abort operation is to rollback previous

[3]We omit details of handling other record types as they do not affect snapshot consistency.

changes. We also delay placing the record in the unified stream until it is the last abort record of the transaction.

THEOREM 4.1. *Read-only transactions running at the secondary execute under global SI.*

PROOF. The merged unified stream contains a sequence of commit records ordered by the same deterministic total order used to enforce I3 at the primaries. As updates made by a transaction are visible only after that transaction commits, the secondary's database state will go through the same sequence of states as the primaries, a property called *completeness* [35].

Now, supposing a read-only transaction T_i starts at the secondary and the last replayed committed update transaction at that secondary is transaction T_j. Since the local concurrency control at the secondary guarantees SI locally, T_i will see a snapshot S_i that includes the effects of all committed update transactions up to T_j and none other. As the secondary goes through the same sequence of database states as the primaries, this is equivalent to running T_i at the primaries right after T_j is committed. Consequently, there cannot be a transaction T_k where $(T_k \in concurrent^s(T_i)) \wedge (T_k \in serial^t(T_i))$ for some sites s and t participating in T_k, and therefore S_i is a consistent snapshot that satisfies global SI. Further, T_i is a read-only transaction that cannot have *write-write* conflict, and thus running T_i at the secondary will execute under global SI. □

4.1.3 Parallelizing Log Merging

It is possible to enhance our system to have multiple log mergers, each one responsible for merging a subset of the tables. This is useful for both scalability and fault tolerance. As mentioned earlier, a log record contains the OId to which the log applies. Therefore, we can enhance the log merger so that it processes log records pertaining to only some OId values, effectively restricting the log merger to process only updates related to some subset of the database tables and indexes. Consequently, the generated unified stream will contain only records related to those tables and indexes, and the secondaries will contain only items in those tables and indexes as well. We can then designate those secondaries to answer read-only transactions that touch only those tables, freeing up other secondaries to service other transaction types. This setting also allows us to allocate more secondaries to serve more popular transaction types. Finally, a client no longer submits the request to a secondary, but rather to a router that will choose the appropriate secondary instance to execute the client's transaction type.

4.1.4 Fault Tolerance

Three failure scenarios can occur: (i) failure of a primary site (ii) failure of a secondary site (iii) failure of the log merger. We address each of these in turn. Global transaction failure at the primaries can be handled in accordance with the provisions of the 2PC protocol [3]. Also, the log merger keeps track of the last successfully received record from each primary (*primary markers*). When a primary goes down and then recovers, the log merger can reconnect back to the primary and simply resume to fetch log records starting from that primary's marker, and the log merging then resumes as usual. Failure of a secondary site has minimal impact on the client, which can always retry the read-only transaction at a different secondary. If a secondary fails, it can first perform local recovery and then bring its database state to the present time by fetching any missing log records from the log merger. Finally, if the log merger goes down, the log merger can resume the log merging from the persistent primary markers.

5. CONFLUXDB IMPLEMENTATION

We implemented our techniques using the ConfluxDB architecture described in Section 2 using PostgreSQL version 9.1.2 as the database system running at each primary and secondary site. We also implemented the log merger modules as described in Section 4.

In ConfluxDB, we build a layer, which we call Conflux Agent on top of PostgreSQL. PostgreSQL guarantees local SI, while the Conflux Agent guarantees global SI by running CGSI and performing distributed certification. We are able to precisely control the transactions while having access to system and transactional state. A nice feature of our approach is that the techniques we propose are built on top of PostgreSQL and require only small modifications, mostly to gather metadata from the internals of the PostgreSQL engine itself. This also allows us to instrument the system to generate a consistent stream of updates that can be installed on replicas at other sites in the distributed system. Another nice property is that it can be similarly layered on top of other database systems that provide local SI concurrency controls.

The Conflux Agent at each individual site coordinates global transactions with the help of Conflux Agents at the other sites. To initiate a global transaction T_i, the client can contact the Conflux Agent at any site. The site (or Conflux Agent) that is contacted will become the coordinator of T_i ($coord_i$).

We now discuss our implementation of the log merger in ConfluxDB. Each replica receiving the unified stream of updates is a PostgreSQL instance running in hot standby mode. In this mode, the database continually fetches and replays log records from a source (which we configured to be the log merger), while still being able to serve read-only transactions. The *Write Ahead Log receiver* component in PostgreSQL is responsible for fetching log records, while the *Startup* process will replay log records as they become available. The Startup process is also involved in initial database recovery by replaying log records from the last successful checkpoint. This Startup process is central to the progress of the replica in keeping up with the primaries because the updates are installed by replaying log records.

We also need to make the following modifications to implement Algorithm 3 for PostgreSQL. First, PostgreSQL does not have an explicit *begin* record; a transaction T_i does not appear in the log until after T_i updates a data item. Therefore we introduce a new *assignTId* record type that is inserted whenever PostgreSQL assigns a new TId for a transaction. As TId assignment is required before any log record related to a transaction T_j can be written, *assignTId* is guaranteed to be the first record seen to mention T_j. The *assignTid* record will simply contain the gTransId of the transaction and the equivalent TId used on that stream. Second, we extend PostgreSQL *commit* and *commit prepared* records to include the global commit timestamp of the transaction.

While read-only transactions never conflict with update transactions under SI, read-only transactions may *interfere* with update log installation at a replica at a secondary. The first type of interference is when an update will cause a read-only transaction to lose its consistent snapshot, such as when the log record wants to drop a table the read-only transaction is using. We can abort the read-only transaction, but doing so may significantly increase the occurrence of read-only transaction aborts. We can also block the update installation but doing so for a prolonged period of time may hamper the progress of update installation. There is a fine balance between the two, and the best combination may very well be workload dependent. The second type of interference is caused by resource contention. The higher the number of read-only transactions running at a secondary, the more resources they will consume, and

consequently, less resources will be available to the Startup process to install updates. Again, we want to balance the number of read-only transactions served with the update replay progress.

We deal with both these possible interferences by combining several PostgreSQL options with thread scheduling. On the PostgreSQL side, we activate the PostgreSQL "vacuum deferral" and "maximum streaming delay" features. The former avoids the first type of interference by postponing in-place vacuum while the latter imposes a limit on how long a read-only transaction can block an update replay. Thread scheduling boosts the progress of the Startup process. PostgreSQL creates one worker process for every connected client session. When the secondary is serving a lot of clients, there will be many more worker processes competing against the Startup process. To overcome this, we isolate the Startup process to one processor core in the system, while ensuring that all worker processes are not scheduled to use that processor core. The goal is to ensure that for most of the time, the Startup process will always have an idle core to run on.

6. PERFORMANCE EVALUATION

We use ConfluxDB described in Section 5 to evaluate the performance of our proposed techniques. We use two workloads, TPC-W and TPC-C, in our experiments.

6.1 Experimental Setup

We use the TPC-W workload Ordering mix (50% update transactions, 50% read-only transactions)[4] and the Shopping mix (20% update transactions, 80% read-only transactions). The initial database contains 1,000,000 items and 100,000 customers, resulting in a physical database size of approximately 1GB though this size grows over the experimental run. We used the default values specified in the TPC-W specification for client think time and session time. As TPC-W does not specify a particular partitioning scheme to follow, we partition the database as follows. Read-only tables are fully-replicated on all master servers. Each of the ITEMS, CUSTOMER, ADDRESS, SHOPPING_CART, and SHOPPING_CART_LINE tables are split into equal-sized partitions, and then spread over the primary sites. Thus, each primary will have a chunk of each of those five tables. To ensure that not all TPC-W update transactions are global transactions, the CC_XACTS, ORDER, and ORDER_LINE tables are hosted under one partition. The goal of our partitioning is to distribute the tables as evenly as possible over the primary sites since skewed partitioning is usually not the desired layout that any database system administrator would strive for. If a distribution proportional to the availability of hardware resources is desired, the partitioning could be changed accordingly. However, our techniques can support any arbitrary partitioning, i.e., they place no restrictions on how the database is partitioned. The default ratio for global transactions is about 30% of the update transaction workload. Note however that as the client workload containing update transactions scales up, this will result in an increasing update load serviced by the primaries so this parameter is simply for selecting an initial ratio.

We also use the TPC-C workload, which models store order processing across a number of warehouses, making it naturally partitionable. Transactions involving more than one warehouse (e.g. ordering an item or processing payment from a different warehouse) are natural candidates for global transactions. We set up our initial database with 12 warehouses, resulting in an initial database size of approximately 1.6GB that grows over the experiment measurement interval. We partition the warehouses such that each partition hosts the same number of warehouses, and each warehouse is hosted by exactly one partition. All of the TPC-C tables are split into equal partitions. All partitions contain the same tables but each one stores only data pertaining to its respective set of warehouses. In TPC-C, the remote warehouse probability refers to the probability that an update transaction is a global transaction. The standard TPC-C workload is update intensive with 92% of the transactions being update transactions. While we can use this mix to demonstrate the scalability of the primaries, the secondaries will not receive enough load. To demonstrate the scalability of the secondaries, we increase the proportion of read-only transactions so that it has a ratio of 70% read-only to 30% update transactions.

We ran the experiments on a dedicated cluster of 16 machines, each equipped with dual-core AMD Opteron 280 (2.4GHz) processors, 8GB of RAM and running PostgreSQL version 9.1.2 on Linux. For all experiments, we limit the amount of memory available for each PostgreSQL instance to 512MB to emulate a memory-constrained environment in which the initial database size does not fit in memory. The machines are connected by a router providing dedicated high-speed networking. Each machine in the 16-node cluster is utilized as follows. Each primary and secondary is hosted on a different machine. The log merger from Figure 2 is also hosted on a separate machine. All of the clients are hosted on another machine, separate from the primaries, the secondaries, and the log merger. All of the graphed experimental results are averages over at least 5 independent runs computed with 95% confidence intervals shown as error bars around each data point.

6.2 Experimental Results

We conduct the following experiments. First, we want to verify that scaling-up the primary through partitioning while providing global SI improves the performance of update transactions. Figures 3, 4, 5, and 6 confirm this, showing the total throughput (for all update transactions) achieved without partitioning by having one primary (1P) database, and by partitioning the database over 4 primaries (4P) and 12 primaries (12P) as the workload (number of clients) is scaled-up while guaranteeing global SI. As the database is partitioned over more sites, the performance gains over an unpartitioned primary database increase. With 12 primaries, ConfluxDB can produce almost 12 times the throughput of a single primary for the TPC-W workload. The TPC-C workload is highly update intensive so the contention at the single primary (1P) bottlenecks the system. When the primary is scaled-up through partitioning, two factors, i.e., alleviation of contention and the distribution of data and workload (through partitioning) combine to produce larger gains in performance where the 12P system has 12 times the throughput of the 1P system. The average response time observed for TPC-W (Figure 4) shows that the 12P system can handle at least 7 times more clients for a response time of 0.3s than the 1P system while the TPC-C response time results vastly outstrip the performance of 1P and 4P systems. While we plot all response time data, in practice, a response time above 3s is undesirable, and we include those for completeness only.

The second goal of our experiments is to measure the performance of the replicated system as both the workload and system resources scale-up. We measured the average read-only transaction response time for 3 ConfluxDB configurations: (i) 1 primary with y secondaries (labeled 1PyS), (ii) 2 primaries with y secondaries (labeled 2PyS), and (iii) 4 primaries with y secondaries (labeled 4PyS). During experiments, we observed that for the TPC-W workload, 1 secondary could support 150 clients at moderate load while for TPC-C, 1 secondary could serve 500 clients at moderate load and we use these ratios to scale-up the number of clients and

[4]We represent each web interaction by default as a transaction.

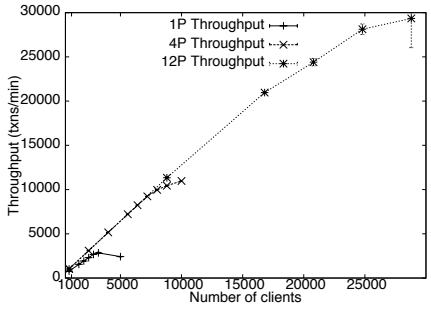

Figure 3: 1P/4P/12P Throughput (TPC-W 80/20)

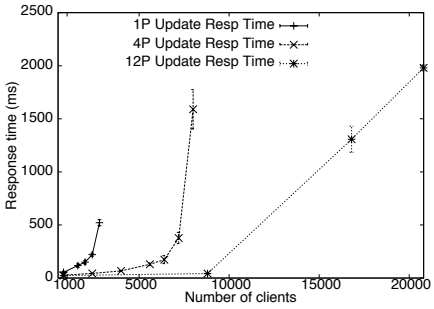

Figure 4: 1P/4P/12P Response Time (TPC-W 80/20)

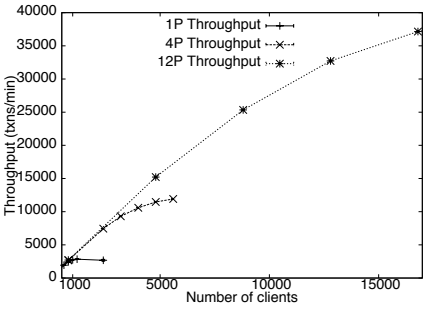

Figure 5: 1P/4P/12P Throughput (TPC-C 8/92)

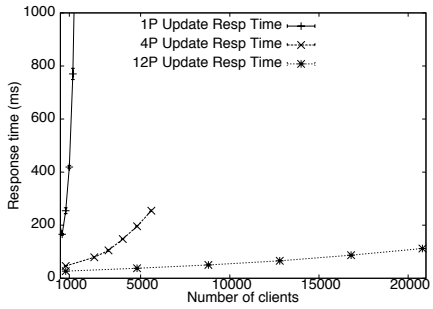

Figure 6: 1P/4P/12P Response Time (TPC-C 8/92)

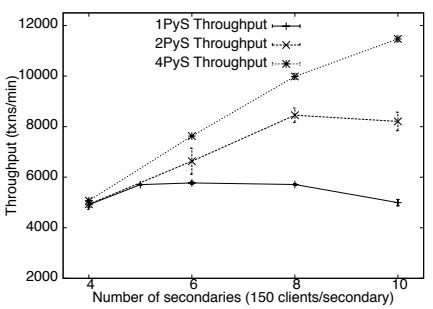

Figure 7: xPyS Throughput (TPC-W 50/50)

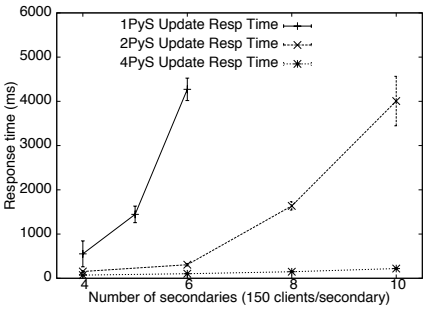

Figure 8: xPyS Update Trans Resp Time (TPC-W 50/50)

secondaries accordingly. Results of these experiments are shown in Figures 7, 8, 9 and 10.

For both workloads, scaling up the number of primary sites results in significant throughput gains and a reduction in update transaction response time for the whole system (Figures 7 and 8; due to space constraints, we omit showing the same trends on graphs observed for TPC-C experiments). The 1PyS curves in Figures 9 and 10 are flat because the single primary saturates quickly, thereby choking the rate at which read-only transactions are processed at the secondaries. As the number of primaries is doubled from 2 to 4, the log merging and protocol overhead is more than offset by increased parallelism in reading more primary log records. Moreover, this continued addition of primary system resources through partitioning spreads the load over more sites, contributing to a significant reduction in read-only transaction response time. In the case of TPC-W, the read-only transaction response time curve flattens out all the way to 8 secondaries (1200 clients) while in the case of TPC-C, it does the same (but to 4000 clients). This demonstrates that scaling-up the primary by increasing the number of partitions significantly alleviates the resource bottleneck for the complete system.

The throughput, update transaction response time and read-only transaction response time curves shown in Figures 7, 8, 9 and 10 demonstrate that large performance gains can be achieved by scaling-up the primary while providing global SI, and that these gains far outweigh the cost of this provision.

We also measure abort rates for both workloads. For TPC-W, the abort rates are low; none of the primary configurations abort more than 3% of transactions submitted for execution. For TPC-C, the abort rates are shown in Figure 11. With only 1P, resource contention results in a very high response time as the workload scales-up. This, in turn, results in a high abort rate on the 1P system compared to 4P and 12P as in TPC-C, a majority of update transactions contend on writing to the same set of rows. However, as seen in Figure 11, the abort rate halves with the addition of 4P from 1P, and is 6 times lower with 12P as the addition of more primary resources significantly reduces the contention window by reducing transaction lifetimes.

We also measure the sensitivity of performance to the percentage of global transactions in the workload. As expected, lower proportions of global transactions lead to better performance while higher proportions result in lower performance. Our performance studies show that irrespective of the proportion of global transactions, increased contention at the primaries ultimately limits the performance of update transactions and the system. Across all experiments, we did not encounter the log merger site to be a bottleneck, which can be alleviated by parallelizing log merging (described in Section 4.1.3) to increase performance if required.

6.3 CGSI and Log Merging Latency

We measure latencies due to enforcing CGSI and log merging in ConfluxDB. For these experiments, we modified several TPC-C and TPC-W benchmark parameters to isolate and measure specific aspects of the system. Since these modify the benchmarks, we will refer to their workloads as "queries" when discussing these results.

To measure the latency of CGSI, we augment Conflux Agent to record the duration that a transaction takes to (i) execute its statements, (ii) perform 2PC, and (iii) perform distributed certification. We set the proportion of global transactions to 100% forcing all update transactions to be global and to go through 2PC and CGSI protocols. We then run TPC-W and TPC-C queries on 2 primaries so that every global transaction will involve every primary. Experimentally, we determined from 1P results that having between 400 to 800 clients for TPC-C and 250 to 500 clients for TPC-W

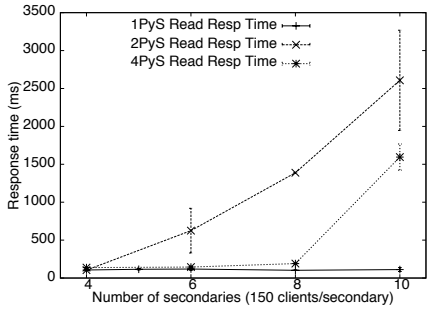

Figure 9: xPyS Read-only Trans Resp Time (TPC-W 50/50)

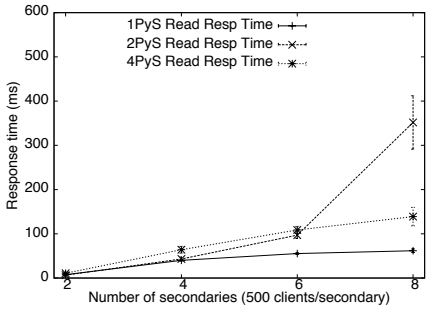

Figure 10: xPyS Read-only Trans Resp Time (TPC-C 70/30)

No. Clients	1P (%)	4P (%)	12P (%)
800	12.54	6.99	1.90
4800	59.47	32.88	10.15

Figure 11: TPC-C Abort Rates

generates load without underloading or overloading the primaries. Our measurements show that at both the lower bound and upper bound number of clients for both workloads, 2PC consumes approximately 55% of the total update transaction response time, the certification algorithm takes only about 4% while the remainder is attributed to executing the SQL statements themselves. CGSI certification adds not more than 7% overhead to 2PC as it leverages 2PC effectively to provide global SI over the primary sites.

Next, to determine the latency introduced by log merging as a proportion of the total replication latency, we set think time to zero to ensure that the read-only transaction response time reflects the replication latency. This forces read-only transactions to block until updates corresponding to the snapshot of the preceding update transaction in the same session have been propagated and installed. This change means that the number of active clients needs to be reduced to ensure that the system is not underloaded or overloaded to remove any load imbalance/skew effects. Experimentally, we determined this number to be 300 clients and 100 clients respectively for TPC-C and TPC-W uniformly distributed over eight secondary sites.

Log merging for transaction T_i ends once T_i's commit record is placed into the unified stream. The difficult question is to determine when log merging for T_i starts. Given that a transaction's log record may arrive before the transaction has committed globally, we cannot simply measure it from the arrival of first record of T_i at the log merging site as this would overlap with actual execution time at the primaries. On the other hand, we also cannot measure it from the arrival of last commit record of T_i as it would not include the delay from waiting to collect all records from all participating sites. Experimentally, we found that measuring the time taken from the arrival of the first commit record of global update transactions at the log merging site to the time that their commit records were placed into the unified stream fell in the middle of these two delays and was representative of both of these cases occurring. Thus, we ran experiments to measure this log merging delay using TPC-C and TPC-W queries. For TPC-W queries, this log merging delay was about 25% of the total latency. For TPC-C queries, this delay was about 15% of the total latency. In general, since response times for TPC-C update transactions are lower than TPC-W update transactions, our expectations that the log merging delay will also be lower (as the timespan of the TPC-W transactions is longer) is confirmed. The replay time, which is the time taken to transactionally apply the log record using the unmodified PostgreSQL hot standby feature constituted about 70% of the total latency for TPC-W queries while it is about 80% for TPC-C. Given the higher number of clients submitting TPC-C queries, this is expected as there would be more contention at the secondaries between these queries and the update records being installed there. The remainder of the time is network latency and transaction execution at the secondaries that together constituted about 5% of the total latency. This shows that the log merger is not the bottleneck for maintaining database replicas at the secondaries.

7. RELATED WORK

Schenkel et al. [30] propose algorithms to achieve global SI on federated databases. They outline pessimistic and optimistic algorithms to ensure global SI, but these depend on a centralized coordinator to issue *begin* and *commit* operations to the database. Their approach requires correct timing and serialization of all begin and commit operations at each participating site by the centralized coordinator, with possibly prolonged delays due to synchronization of all transaction starts and commits.

Bornea et al. [5] show how local SI concurrency controls can be used to provide global serializability on fully replicated databases using a central certifier. They do not consider partitioned databases.

Session-based SI guarantees have been proposed [12, 14, 19] but none of these consider scaling up through partitioning. Parallel SI [31] supports multiple primary sites but each primary site can install updates in different orders and does not provide global SI. Clock-SI [13] uses physical clocks at each primary sites to produce consistent snapshots of key-value store data that satisfy SI, but it relies on physical clock synchronization. Further, it requires the underlying database system to support obtaining a snapshot older than the latest installed one.

The problem of partitioning an SI database has also gained interest from the open source community, as is evident with the release of Postgres-XC (http://postgres-xc.sourceforge.net/), which is a transparent synchronous solution for partitioned SI databases using Postgres as the underlying DBMS. Postgres-XC uses a centralized global transaction manager to assign transaction identifiers and snapshots [28]. C-JDBC [8] is an open-source system that allows a cluster of database instances to be viewed as a single database. However, it only allows one update, commit, or abort executing at any point in time on a virtual database, and uses a single scheduler to handle concurrency control and isolation level.

In [26], the authors proposed replication over a multi-owner system by ensuring consistent commit of update transactions that relies on physical clock synchronization. In contrast, our solution provides replication over a partitioned database and we depend only on logical timestamps instead of physical clock synchronization.

Calvin [33] proposes to use a "deterministic" approach to improve the performance of distributed transactions. It relies on acquiring advance knowledge about the workload before transaction execution can start, examples of which include workload analysis

and development of storage system API to support Calvin calls for multi-operation transactions for every workload. ConfluxDB does not require such advance knowledge and it does not belong to the domain of deterministic systems. Moreover, our work is not related to comparing non-deterministic systems against systems from the deterministic domain. Hyder [4] on the other hand adopts a shared storage design for scale out that works with only solid-state storage devices such as flash. It uses log-structured store that is mapped to shared flash storage but does not hold a database system like ConfluxDB. Since the transactional and indexed storage layers are not implemented with a flash-based log implementation, a combination of system and simulation techniques are used to predict performance.

Log records are commonly used by database systems to provide durability and fault tolerance. DB2 uses a form of log merging to assist recovery of its partitioned database [15]. Using log records in database replication has been implemented in PostgreSQL, but it only supports full replication of one database instance. [11, 23] explored the feasibility of merging multiple log streams to infer a global serialization order. In contrast, our work focuses on partitioned SI databases.

8. CONCLUSION

In this paper, we show how to provide global SI for a partitioned database and to lazily maintain its replicas. Our techniques remove the single-site bottleneck previously faced by updating transactions without relying on a centralized component to order them. Distributed update transactions are free to update multiple data items since no restrictions are placed on how the primary database is partitioned. We propose a scheme that merges multiple update streams from the partitioned database sites into a single stream that is consistent with global SI ordering to lazily maintain replicas of the database at secondary sites. We demonstrate the effectiveness of our approach through ConfluxDB, a PostgreSQL-based implementation of our techniques. Our experimental results show that ConfluxDB significantly improves the performance of update transactions while maintaining global SI for read-only transactions in the distributed system.

9. REFERENCES

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware '03*.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.

[3] P. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann Publishers Inc., 1997.

[4] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, 2011.

[5] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *ICDE '11*.

[6] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *SIGMOD '99*.

[7] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *PODS '97*.

[8] E. Cecchet. C-jdbc: A middleware framework for database clustering. *IEEE Data Eng. Bull.*, 27(2):19–26, 2004.

[9] P. Chairunnanda. Multi-Master Replication for Snapshot Isolation Databases. Master's thesis, University of Waterloo, 2013.

[10] K. Daudjee and K. Salem. A Pure Lazy Technique for Scalable Transaction Processing in Replicated Databases. In *ICPADS '05*.

[11] K. Daudjee and K. Salem. Inferring a serialization order for distributed transactions. In *ICDE '06*.

[12] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 715–726, 2006.

[13] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *SRDS '13*.

[14] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. *SRDS '05*.

[15] IBM. *Log stream merging and log file management in a DB2 pureScale environment*.

[16] IBM. *DB2 Universal Database Replication Guide and Reference*, 2000. version 7.

[17] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. Improving the scalability of fault-tolerant database clusters. In *ICDCS '02*.

[18] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 2000.

[19] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly consistent replication for a bargain. In *ICDE '10*.

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

[21] P.-A. Larson, J. Goldstein, and J. Zhou. MTCache: Mid-Tier Database Caching in SQL Server. In *ICDE '04*.

[22] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*, pages 419–430, 2005.

[23] C. Liu, B. G. Lindsay, S. Bourbonnais, E. Hamel, T. C. Truong, and J. Stankiewitz. Capturing global transactions from multiple recovery log files in a partitioned database system. In *VLDB '03*.

[24] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *ICDE '06*.

[25] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *SIGMOD '02*.

[26] E. Pacitti, M. T. Özsu, and F. Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *Euro-Par 2003*.

[27] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware '04*.

[28] Postgres-XC Development Group. *GTM and Global Transaction Management*.

[29] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.1: Partitioning*, 2012.

[30] R. Schenkel, G. Weikum, N. Weissenberg, and X. Wu. Federated transaction management with snapshot isolation. In *TDD '99*.

[31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP '11*.

[32] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *TODS*, 1979.

[33] 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.

[34] P. S. Yu, M. Chen, H. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE TSE*, 1992.

[35] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *DAPD*, 1998.