Multi-Master Replication for Snapshot Isolation Databases

by

Prima Chairunnanda

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

Lazy replication with snapshot isolation (SI) has emerged as a popular choice for distributed databases. However, lazy replication requires the 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 have built our techniques into PostgreSQL and demonstrate their effectiveness through experimental evaluation.

Acknowledgements

All praises to Allah Almighty who guided me and gave me strength to complete this thesis.

My deepest gratitude to my supervisor Dr. Khuzaima Daudjee who has given me the opportunity to pursue this study and has advised me throughout my study. This thesis is the result of our countless number of discussions, emails, and Google Hangout sessions. I would also like to thank my thesis readers, Prof. M. Tamer Özsu and Prof. Bernard Wong, for their valuable feedback.

I would like to thank Adam Hartfiel who helped me a lot to resolve cluster-related issues. I further thank Prof. Kevin G. Hare and Prof. Pascal Poupart for their graciousness to let me run my experiments on the cluster.

I could never be where I am today without the support from my family. I am deeply grateful to my parents for always believing in me and for their prayers and support. I would like to specifically thank my dearest wife, Larissa Fedora Hapsari Putri, who supported me and kept me motivated throughout my study and through sleepless nights of writing this thesis.

I would also like to thank my friends and colleagues in Waterloo, back home, and the rest of the world, who directly or indirectly supported me throughout my study. Finally, I would like to thank the University of Waterloo and the faculties and staffs for providing me with an excellent environment to pursue my research interest.

Dedication

I dedicate this work to my beloved parents, and my loving wife.

Table of Contents

List of Tables

List of Figures

Chapter 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. A problem that emerges as a result of this scale-up through distribution is that of providing *global* SI over the distributed database system. This 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.

There have been proposals to provide global SI over partitioned and replicated databases [6, 14, 35, 39]. However, none of these proposals consider how to scale-up a single-site (primary) database system without relying on a centralized site or component to determine a global update order in the distributed system.

Data replication has become a popular choice to improve the performance, availability, and fault tolerance of a database. Broadly, we can categorize data replication protocols into *eager* and *lazy* replication. Eager replication protocols ensure that a data item at all replicas has been updated before the transaction is committed. On the other hand, lazy replication protocols require a data item to be updated only on a subset of replicas before committing the transaction, opening up the possibility that some replicas store out-of-date items. Despite this drawback, lazy replication is an attractive choice to achieve higher performance.

In the lazy master architecture (Figure 1.1) proposed in [29, 25, 35, 11], a client submits transactions to one of the *secondary* sites. Read-only transactions are serviced locally at a secondary, while update transactions are forwarded to the *primary* site for execution. Update transactions are replicated lazily to the secondary sites. A nice property of this architecture is that as the read-mostly workload scales-up, the system can be scaled-up by adding more secondary sites.[1]

[1]Many common workloads in application areas such as web commerce and decision support are read-mostly

However, having a single primary site at which all update transactions execute is a restriction with significant drawbacks. As the read-mostly workload scales-up, an increasing update load is placed on the primary site. This limits system scalability since the single primary site becomes a bottleneck [11]. The fact that the primary becomes a bottleneck is not just an artifact of our study. It is also pointed out in [25] that the scalability of the distributed system is limited by a single saturated primary site. Other researchers that have proposed lazy master replicated systems have also acknowledged the system performance degradation that results from an increasing load on the single primary site [35].

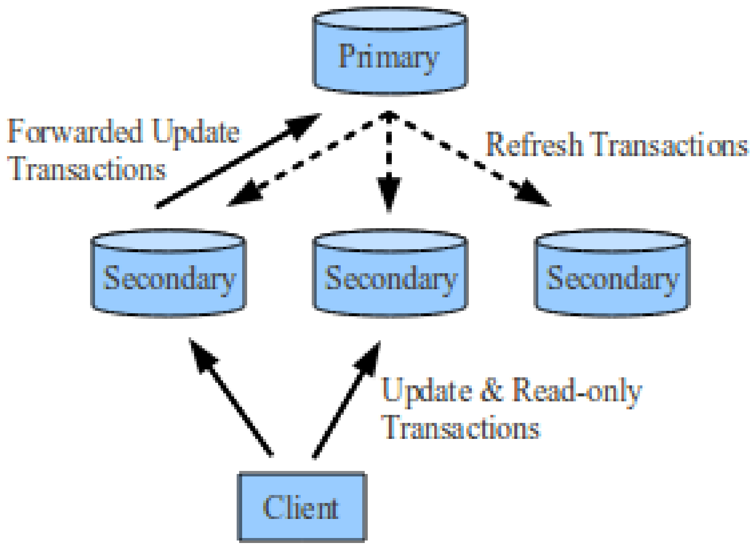
Primary
Forwarded Update Transactions
Refresh Transactions
Secondary
Secondary
Secondary
Update & Read-only Transactions
Client

Figure 1.1: Lazy Master Architecture

In prior related work that considers this problem, the update workload needs to be partitioned a priori or transactions are restricted to updating data at a single site [7, 8, 12]. Other approaches require some type of centralized middleware or sequencer to determine a global order for update transactions [35, 21, 2, 23] or the prediction of conflicts between transactions to partition the primary database a priori into conflict classes [19]. To the best of our knowledge, our approach is the first to not suffer from any of these restrictions in providing global SI over partitioned databases.

The contribution of this thesis is a set of techniques that supports scaling-up a database through partitioning while providing global SI. Distributed update transactions can execute on a

[42, 31].

database partitioned over multiple primary sites. Note that in many workloads such as TPC-W and TPC-C, distributed update transactions cannot be avoided. Our work determines a global order to provide SI for distributed update transactions. We use a novel scheme that merges multiple update streams from the partitioned sites into a single, unified, stream that is consistent with global SI ordering and installs these updates in the same order at sites that hold database replicas. Our scheme also avoids transaction inversions, which happens for example when a client is unable to read its last update despite the update preceding the read in the execution order [14]. 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 [5]. We build our techniques into the PostgreSQL open-source database system to demonstrate their viability and efficiency.

The rest of this thesis is structured as follows. We present our system model in Chapter 2. In Chapter 3, we describe properties that need to be maintained to guarantee global SI. Chapter 4 presents our protocol for transaction management to provide global SI to update transactions executing over the partitioned primary sites. Chapter 5 describes our log merging scheme to derive a single stream of updates from multiple primary sites for installation on replicas at secondary sites. We evaluate the performance of our proposals in Chapter 6 and discuss related work in Chapter 7 before we conclude the thesis.

Chapter 2

Overview

In this chapter, we describe an overview of our system model, followed by the terminology and definitions relevant to the discussion in subsequent chapters.

2.1 System Model

The primary database is partitioned over one or more sites, as shown in Figure 2.1. No restrictions are placed on how the primary database is partitioned. The primary sites comprise the primary cluster. We do not place any restrictions on the contents of each database partition but note that several techniques for distributed database design have been proposed [30, 3, 22]. A complete replica of the primary database is held at each secondary site. Each site consists of an autonomous database system with a local concurrency controller that guarantees SI.

Clients connect to one of the secondary sites and submit transactional requests. Each client's transactions constitute a session. Each transaction has associated with it, either explicitly or implicitly, a session label. The customer sessions may be tracked by the application server or web server using cookies or a similar mechanism. In this case, the upper tiers can create session labels and pass them to the database system to inform it of the session labels. We assume that read-only transactions are distinguished from update transactions in the request streams. Read-only transactions are executed at the secondary site to which they are submitted. Update transactions are forwarded by the secondaries to the primary cluster and executed there.

Transactions execute at the primary sites using a 2PC protocol. The primary site that starts the transaction also acts as the transaction coordinator, which is often the norm. In the 2PC protocol, the coordinator generates a *prepare* message, which is sent to all participant sites involved in

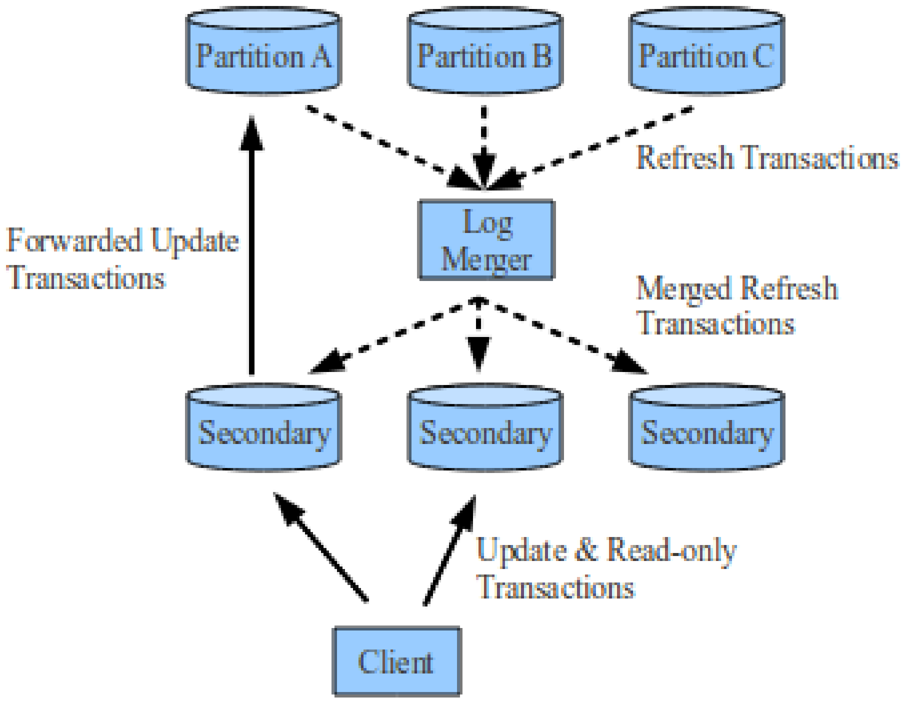

Figure 2.1: System Architecture

the distributed transaction. The participant sites generate and respond with an acknowledgement message, which we shall call *prepare_ack*. After acknowledgement messages are received from all participant sites, the coordinator commits, which generates a *global_commit* message that is sent to all participants. Each participant then either commits, generating a *commit* message, or aborts the transaction. Since the 2PC protocol does not commit a transaction until all read/write operations of the transaction have executed, and each local concurrency control guarantees SI, the system of primary sites guarantees global SI.

Update information can be extracted from the database logs using a standard mechanism such as a log sniffer [18]. The updates from each log are merged into a single stream using the log merging algorithm described in Chapter 5 after which they are propagated lazily to the secondary sites in serialization order.

At each secondary site, propagated updates are placed in a FIFO update queue. An independent refresh process at each secondary site removes propagated updates from the update queue and applies them to the local database copy. To distinguish them from the original update transaction happening at the primary sites, we use the term *refresh transaction* to refer to the application of updates at the secondary sites.

2.2 Terminology

We will now define the terminology relevant to update transactions at the primary cluster presented in Chapters 3 and 4. We will introduce terminology relevant to read-only and refresh transactions later in Chapter 5.

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 thesis, 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 which 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 happening in the database system: the *begin transaction*, *read* (of an item), *write* (item), and *commit transaction*. We assume these events are totally ordered at each site by the *happened-before* relation, where $e \prec e'$ means that event e happened before event e'. Existing mechanisms, such as Lamport clock [24], can be used to derive this relation.

Snapshot Isolation originates from multiversion concurrency control (MVCC) where multiple versions of the same data item may exist at any one time. 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. $r_i[x_j]$). The rule to select which x_j to read is key to the definition of a transaction isolation level, and we will discuss this specifically for SI in the next section.

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 if and only if T_j sees T_i's effects but T_i does not see T_j's effects, or simply $T_i \prec T_j$. To be more precise, at site s, $T_i^s \prec T_j^s$ 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, denoted as $T_i || T_j$.

In considering pairs of transactions, we will use the notion of whether one transaction is dependent on another or not to derive an ordering of these transactions. We formally define the anti-dependency rather than the dependency since it simplifies the presentation.

Definition 2.1 *Let* $RS(T_i^s)$ *and* $WS(T_i^s)$ *be the read-set and write-set of* T_i^s*. The predicate* $\neg dependent(T_i, T_j)$ *is true, if and only if* $(RS(T_i^s) \cap WS(T_j^s) = \emptyset) \wedge (WS(T_i^s) \cap RS(T_j^s) = \emptyset) \wedge (WS(T_i^s) \cap WS(T_j^s) = \emptyset)$.

In the thesis, we will use where needed the positive predicate $dependent(T_i, T_j)$, which is the negation of Definition 2.1, formally defined as:

Definition 2.2 *The predicate* $dependent(T_i, T_j)$ *is true, if and only if* $(RS(T_i^s) \cap WS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap RS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap WS(T_j^s) \neq \emptyset)$.

We will not go into specific details on how to track the read and write sets of a transaction, although it needs to be noted that one may need predicate matching to ensure the read set includes both selected and scanned items.

Chapter 3

Transaction Management for Partitioned SI Database

3.1 Snapshot Isolation

Snapshot Isolation is commonly discussed with reference to the "concurrency phenomena" that are allowed to happen in the database. The original ANSI-92 SQL standard [1] describes three phenomena (Dirty reads, Non-repeatable reads, Phantoms) to define four transaction isolation levels (Read Uncommitted, Read Committed, Repeatable Reads, and Serializable), where the highest level – Serializable – is defined as the absence of the three phenomena. However, it has been pointed out to be insufficient by subsequent studies [4], as there are non-Serializable isolation levels that do not exhibit the three phenomena, with Snapshot Isolation being one of them. Table 3.1 shows a summary of concurrency phenomena that may occur for different transaction isolation levels, with the concurrency phenomena originally described in ANSI-SQL written in italics. Since then, there have been further research in uncovering and discovering more concurrency phenomena (e.g. [16], and a good summary in [33]), but we will only focus on those in Table 3.1.

SI gets around the *Dirty Read*, *Non-repeatable Reads*, and *Phantom* phenomena by reading from a snapshot. Under SI, throughout a transactions's lifetime, a transaction T_i is guaranteed to read data items from the snapshot S_i obtained when T_i starts. In particular, this only includes the 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.

Table 3.1: Concurrency Phenomena on Different Transaction Isolation Levels.

Isolation Level	*Dirty Read*	*Non-repeatable Reads*	*Phantom*	Lost Update	Write-Skew
Read Uncommitted	YES	YES	YES	YES	YES
Read Committed	NO	YES	YES	YES	YES
Repeatable Reads	NO	NO	YES	YES	YES
Snapshot Isolation	NO	NO	NO	NO	YES
Serializable	NO	NO	NO	NO	NO

To avoid the *Lost Update* phenomenon, concurrent transactions are not allowed to write into the same items, and when it happens, only the first commit is successful. This is called the "First-Committer Wins" rule [1].

The *Write-Skew* phenomena happens when two concurrent transactions T_i and T_j read some overlapping items, and then write disjoint sets of items. Under SI, both can commit since their write sets do not overlap. This is not permitted under Serializable isolation level as there is no equivalent serial history [4].

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 $begin(T_i) \prec commit(T_j)$. 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 multiversioning 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.

3.2 Global Snapshot Isolation

When a database is distributed over multiple sites, the challenge is to ensure that SI holds globally for concurrent, distributed, transactions executing over this partitioned database. These transactions will independently issue *begin* and *commit* events at multiple sites, which are autonomous. We make no assumptions about global clock synchronization, i.e. there is no global clock that can accurately assign a total order on all such events in the partitioned, distributed, database

[1]Some SI implementations adopt the First-Updater Rule, which does not prevent our protocol from working equally well. We chose First-Committer Wins for our system implementation as the underlying PostgreSQL adopts it also.

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 as well. Selecting a snapshot S_i for T_i to access at all sites that host the partitioned database is a challenge. In this context, we use the concept of consistent snapshot, formally defined as follows:

Definition 3.1 *A transaction T_i sees a consistent snapshot S_i 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 the 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. There are SI variants that allow transactions to request a snapshot of the database as of a specific time in the past [27]. However, they usually use real time to describe a snapshot, which is challenging to enforce, manage, and synchronize in a distributed system.

Note that we are not interested in solutions that trivially execute a set of transactions serially in the same order at all sites, thereby ensuring a total order over all sites as this would severely limit the performance of the system. Previously, we discussed how the relative order of *begin* and *commit* events affects snapshot visibility for a transaction. We will now focus on specific pairs of *begin* and *commit* operations that matter in ensuring global SI.

The database state seen and committed by update transactions at the primary sites correspond to the transactions' start and commit events. If a transaction T_k is to see a consistent snapshot S_k over the partitioned primary sites, the system needs to ensure that the following conditions hold for each transaction pair T_i and T_j:
C1. 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.
C2. 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.
C3. 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.

C1 and C2 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.

C3 is important to prevent two different snapshots from seeing partial commits that are incompatible with each other. Assume that the value of the data items are originally x_0 and y_0.

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. We avoid the occurrence of such aborts by enforcing C3.

If $\neg dependent(T_i, T_j) \wedge T_i \neq T_j$, 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 where it holds that $dependent(T_i, T_k)$ and $dependent(T_k, T_j)$, but $\neg dependent(T_i, T_k)$, 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. We will go into more details about this detection in our protocol description in Chapter 4.

An important consequence of the above paragraph is that if the majority of transactions only touch a small number of non-intersecting items, it allows their *begin* events to be executed in any relative order with respect to the other transactions' *commit* events, greatly increasing parallelization opportunity. Note that the relative order among *commit* events is still important even if the transactions involved are data-independent to each other for the same reason discussed for condition C3 earlier.

Our work was inspired by Schenkel et al. [39] on achieving global SI under federated setting. In their work, global transactions are sent to a centralized coordinator, which then sends *begin* and *commit* to participating sites at times in accordance with global SI. The role of the coordinator is constrained by the fact that they do not control the participating sites, and thus optimization possibilities are limited.

In contrast, we build a layer, which we call GSI Agent, to guarantee global SI on top of PostgreSQL, which guarantees local SI. We are able to precisely control the transactions while having access to system and transactional state. A nice feature of our work is that all of the techniques we propose are built on top of PostgreSQL, and do not require modifying 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 of our system is that it can be similarly layered in top of other database systems that provide local SI concurrency controls.

The GSI Agent at each individual site coordinates global transactions with the help of other GSI Agents at the other sites. To initiate a global transaction T_i, the client can contact the GSI Agent at any site, after which that GSI Agent instance will become $coord_i$. Preferably, to minimize the communication overhead, $coord_i$ is a site that is involved with processing T_i though this is not required. 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. 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.

We would like to point out two important characteristics of our system. First, any site in the system can be the coordinator of a transaction. In fact, even if T_i never accesses any data at a particular site, that site can still be $coord_i$, although this practice is discouraged for performance reasons. Second, the outcome of a transaction is determined only by the sites that are participating in executing transaction T_i. Our protocol obviates the need for a single, centralized, coordinating site to decide whether a transaction should be committed or aborted. We call our protocol Collaborative Global SI (CGSI), presented in the next chapter.

Chapter 4

Collaborative Global Snapshot Isolation

As has been discussed in the previous chapter, ensuring that a transaction sees a consistent snapshot is a challenge in the distributed setting. Before describing the approach we take to ensure this, we first describe two other approaches used in prior work.

The first approach is to agree on a particular snapshot at the time the transaction begins. On systems where a transaction cannot request which specific database snapshot it wants to see, the ***begin*** operations of subtransactions at all participating sites need to be precisely controlled so that all subtransactions see a consistent snapshot. Consider two sites s and t participating in T_i. Suppose both also participate in T_j that is committing, and that $dependent(T_i, T_j)$ is true. For T_i to see a consistent snapshot, either $begin(T_i^s) \prec commit(T_j^s) \wedge begin(T_i^t) \prec commit(T_j^t)$, or $commit(T_j^s) \prec begin(T_i^s) \wedge commit(T_j^t) \prec begin(T_i^t)$ must hold. This is not possible to enforce unless we know the set of all participating sites from the start. Let us see why this is the case. For example, suppose we did not know site t was also participating and we started the subtransaction only on site s, and as it turned out, $begin(T_i^s) \prec commit(T_j^s)$. Later on, we discovered that site t is also participating but by then, it was too late to perform $begin(T_i^t)$, as $commit(T_j^t)$ had already happened. In [39], this is achieved by centrally controlling and synchronizing the ***begin*** and ***commit*** operations at all sites. This central entity also needs to know the state of all transactions and all sites. The reliance on precise timing exposes this approach to prolonged delays due to synchronization of all transaction starts and commits.

In the second, more optimistic approach in [39], subtransactions are started as needed so the final set of participating sites is known just before commit time. Checks are made every time a subtransaction is started to verify that the snapshot seen by each new subtransaction at every site is consistent with the snapshots seen by all previous subtransactions. We shall refer to this check as *certification*. If an inconsistency is found, the transaction is aborted. This approach also requires a centralized coordinator to make this certification and to control transaction execution accordingly, presenting a bottleneck in the system.

In our approach, all certifications are performed just before commit time. This provides the advantage that starting a subtransaction is fast, and the information required for the certifications can be piggybacked on the 2PC messages. Our approach is more optimistic than the second approach above and we may end up aborting a transaction to ensure that all of its subtransactions see consistent snapshots. However, from our experimental results, the abort rate due to this is very low for workloads where update transactions are not long running, which is usually the case for many real workloads.

4.1 Consistent Snapshot Determination

Recall that there are three conditions (C1–C3) that need to be maintained for a transaction to see a consistent snapshot. We describe how CGSI preserves each of these conditions.

To preserve C1 and C2, 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, the GSI Agent at site s conducts the certification in the following manner. Each of the transactions in $active^s$ and $committed^s$ is categorized into one of two sets: 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 $T_j \in serial(T_i^s)$. S_i is deemed to be inconsistent 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.

The distributed version of the certification that GSI Agent performs is as follows.

1. $coord_i$ asks each participating site s to send the transactions in $concurrent^s(T_i)$.

2. Participating site s sends its $concurrent^s(T_i)$ to $coord_i$.

3. $coord_i$ waits until all participating sites have responded. All responses of $concurrent^s(T_i)$ are merged into a single set $gConcurrent(T_i)$. If $T_j \in gConcurrent(T_i)$, then $T_j^t || T_i^t$ 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 $dependent(T_i, T_j) \wedge (T_j \in gConcurrent(T_i)) \wedge (T_j \in serial^s(T_i))$. 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 result. Otherwise, site s sends a positive certification result.

6. $coord_i$ aggregates the certification results from all participating sites. If any participating site replies negatively, it means some subtransactions of T_i see inconsistent snapshot, and T_i has to be aborted. Otherwise, T_i can proceed to commit.

Theorem 4.1 *For a committing transaction T_i, if there exists a 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 version of the certification algorithm above 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 responses from the sites are positive. Let us now see it from the perspective of site t, where it happens that $T_j \in serial^t(T_i)$. If site t responded positively during the conflict detection, it means site t could not find a data-dependent T_j, such that $(T_j \in gConcurrent(T_i)) \wedge (T_j \in serial^t(T_i))$. It has been established that $T_j \in serial^t(T_i)$ from the initial condition. 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 actually 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. $\square$

Several MVCC-based systems including PostgreSQL are already tracking the list of concurrent transactions to determine tuple visibility in a snapshot. What remains is to track the full set of committed transactions, which is usually not readily available in memory and is only recoverable 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.

In this expiration strategy, we set a limit for the maximum number of transactions kept in the committed transactions set of a site. Once the maximum is reached at site s, it will evict the oldest committed transaction from the set. We maintain and utilize the site's Lamport clock [24] to infer partial ordering between events. Each site s keeps track of the value α^s, which is the highest Lamport clock value as ever evicted from its set. After that, we modify the distributed certification algorithm executed when T_i is committing as follows:

1. At step 2, each site s does the following processing. Let $L[x]$ be the Lamport clock value of event x. Let T_j be a transaction such that $T_j \in concurrent^s(T_i)$ and there is no $T_k \in$

$concurrent^s(T_i) \wedge L[begin(T_k^s)] < L[begin(T_j^s)]$. The value $\theta^s(T_i) = min(L[begin(T_i^s)], L[begin(T_j^s)])$ is then piggybacked on the response sent to $coord_i$.

2. At step 4, $coord_i$ computes the value $\Theta(T_i)$, which is the lowest value among all $\theta^s(T_i)$, and piggybacks this value in the message sent back to the participating sites.

3. At step 5, if any participating site s finds that $\Theta(T_i) \leq \alpha^s$, it immediately sends a negative response.

In using the expiration strategy, if any site s finds that $\Theta(T_i) \leq \alpha^s$, then $committed^s$ no longer contains the necessary information to ascertain whether T_i sees a consistent S_i. Thus, transaction T_i needs to be aborted.

Proposition 4.2 *If T_i sees an inconsistent S_i, and for each participating site s, $\Theta(T_i) > \alpha^s$, then there is at least one participating site t which detects that T_i sees an inconsistent S_i.*

Proof Since T_i sees an inconsistent S_i, then for some sites s and t, there is a T_j where $dependent(T_i, T_j) \wedge (T_j \in concurrent^s(T_i)) \wedge (T_j \in serial^t(T_i))$. As $T_j \in concurrent^s(T_i)$, then $\Theta(T_i) \leq L[begin(T_j^s)]$. With 2PC, we also know that $L[begin(T_j^s)] < L[commit(T_j^t)]$, as all subtransactions must have been started and prepared at all participating sites before it can be committed at any site. Since t is a participating site, $\alpha^t < \Theta(T_i)$, and consequently, we can derive the inequality $\alpha^t < \Theta(T_i) \leq L[begin(T_j^s)] < L[commit(T_j^t)]$. It follows that T_j cannot have been evicted at site t, and site t will be able to detect the conflict. □

Next, we will show how CGSI preserves C3. 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 reduce communication overhead between sites, we piggyback information onto existing messages where possible.

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

1. On coordinator receiving request to prepare a global transaction to commit, the coordinator piggybacks its event clock value to the broadcasted *prepare* messages.

2. On participant receiving *prepare* message from coordinator, the participant updates its event clock with the maximum between its current event clock value and the one carried by the *prepare* message. Then, the participant sends back a *prepare_ack* piggybacking the updated clock value.

3. On coordinator receiving *prepare_ack* message from participants, the coordinator updates its event clock with the maximum between its own event clock value 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 and globally decides to commit the transaction. The incremented event clock value becomes the *commit timestamp* of the transaction.

4. On participant receiving *commit* message from coordinator, the participant updates its event clock with the maximum between its own event clock value 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 having 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, hostname, or an assigned value. This scheme is similar to the timestamp generation methodology used in [41]. Naturally, we require the existence of a total order among the identifiers themselves. Together, the commit timestamp and the coordinator's identifier form the *global commit timestamp* of a transaction.

Definition 4.1 *The* global commit timestamp *of a transaction T_i, denoted as $gct(T_i)$), is an ordered pair $\langle commit\_tstamp(T_i), id(T_i)\rangle$, where $commit\_tstamp(T_i)$ is the commit timestamp of T_i, and $id(T_i)$ is the identifier of $coord_i$.*

Definition 4.2 *For every distinct two transactions T_i and T_j, $T_i \prec T_j$ if and only if:*

1. $commit\_tstamp(T_i) < commit\_tstamp(T_j)$, *or*

2. $commit\_tstamp(T_i) = commit\_tstamp(T_j)$ *and* $id(T_i) < id(T_j)$.

The second case in the definition above is used to break ties in the case of assigned commit timestamps that are the same. The actual commit actions to the database have to be executed strictly in the order of the commit timestamps. Under the assumption of no deadlock, as long as each site locally commits its global transactions in the order defined by $\prec$, 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, a 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 has a priority queue data structure containing prepared global transactions, intuitively called PreparedQueue. Each prepared global transaction in PreparedQueue has a `highest_clock` attribute, which stores the highest event clock value ever received regarding the global transaction. There is also a flag `ready_to_commit`, which indicates whether the global transaction has passed 2PC (i.e. the site has received global-commit decision from the coordinator) and is ready to be written to the database. This flag also serves another important purpose: when it is `true`, the stored `highest_clock` must correspond to the transaction's $commit\_tstamp$, 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 4.3 *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 in which site s is participating with $T_j \prec T_i$.*

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 $commit\_tstamp(T_i) \leq event\_clock^s$. As the coordinator increases its event clock value on commit, it will be the case that $commit\_tstamp(T_j) \geq event\_clock^s + 1$. 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` must be at least $commit\_tstamp(T_i)$. As T_j has been prepared, then T_j must be in the queue, and its `highest_clock` 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 $commit\_tstamp(T_j)$ 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 when T_j has been prepared, $commit\_tstamp(T_j)$ 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 been prepared yet. Then, $commit\_tstamp(T_i)$ must be at most $event\_clock^s$. When the *prepare* message for T_j comes in, site s will piggyback an event clock value of at least $event\_clock^s$. Consequently, the eventual $commit\_tstamp(T_j)$ will be at least $event\_clock^s + 1$, thus $T_i \prec T_j$.

□

Unlike the regular 2PC, transaction T_i is not immediately committed after the global decision to commit has been made. Instead, CGSI records the commit timestamp in `highest_clock`, turns on the `ready_to_commit` flag, 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. Algorithm 4.1 shows the steps taken by this committer thread.

Theorem 4.4 *If each primary site runs CGSI and enforces SI locally, the transactions in the system will run under global SI.*

Proof First, Algorithm 4.1 enforces C3 across all sites by processing commit in a deterministic order. Therefore, it cannot happen that two sites participating in the same transactions T_i and T_j commit them in different order. 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 4.1, T_i^s is allowed to commit only if all T_i^t read from some consistent snapshot that satisfies C1 and C2, where t is the set of participating sites of T_i. Consequently, globally, T_i sees a consistent snapshot S_i that satisfies SI.

Since both of the above statements apply to all committed transactions[1] in the database globally, the database is also running under SI globally. □

[1]Aborted transactions cannot cause any visible changes, and as such, we can safely exclude them from consideration.

Normally, the 2PC participants are notified of the global decision after the transaction has been successfully committed at the coordinator side. If the coordinator fails to commit for any reason, the transaction can be safely aborted. In CGSI, the global decision and the coordinator commit do not necessarily happen at the same time. This might actually lead to a deadlock when two sites want to commit two different global transactions, and each site is the coordinator of one while being the participant of the other. To break up this deadlock, when the global commit decision is made, the coordinator sends a *notify_commit_decision* message carrying the transaction's global commit timestamp. A separate *global_commit* message is sent once the transaction has actually been committed, after which the participants can insert the transaction in their own PreparedQueue. However, in the presence of a separate deadlock detection system, the *notify_commit_decision* message is no longer necessary, and can be safely removed.

A disadvantage of the above approach is that it effectively prolongs the duration of the "prepared" state. While in this state, transactions typically keep holding the locks, and no other transactions can acquire any conflicting locks. To overcome this, we also support an "Optimistic Commit" or **opt-commit** mode, where the coordinator sends *global_commit* message immediately after the global decision has been made without waiting for the actual commit at the coordinator's side. On some preliminary tests, we found that the opt-commit mode increases the throughput significantly across the board, so we always activate this mode for all our experiments. In the event that the coordinator fails to commit, we employ the same error handling mechanism as when a participant fails to commit; thus, a different action is not required. In opt-commit mode, the coordinator must durably log the global commit decision separately from the actual commit operation.

Algorithm 4.1 Commit Thread running on all sites

```
 1: procedure CommitThreadMain($a, b$)
 2:     $pt \leftarrow$ the root of PreparedQueue
 3:     if $pt.ready\_to\_commit$ then
 4:         Commit $pt$ into database
 5:         Update LastCommittedGCT to gct of $pt$
 6:         Remove $pt$ from PreparedQueue
 7:         if this site is the coordinator for $pt$ then
 8:             Inform client that $pt$ has been committed
 9:         end if
10:     end if
11: end procedure
```

4.2 Session Guarantee

In a single-site system, when a client starts a new transaction, the snapshot captures the state of the database up to the last committed transaction. In a decentralized autonomous, multi-site system, each site may not always be fully-synchronized with other sites. The CGSI protocol allows some degree of delay between participating sites. Because a client is notified of the transaction commit right after the coordinator has committed the transaction, it is possible that some of the participating sites have not yet processed the commit, as illustrated in Figure 4.1. First, the client asks $coord_i$, say site s, to commit T_i. $coord_i$ initiates the 2PC, and once it has received positive acknowledgement from all participants, proceeds to issue a global commit decision, commits T_i^s locally, and reports the outcome to the client. The client, satisfied that T_i has been committed, may initiates a new transaction T_j. If the client, which is free to contact another site different from the last one, contacts site t, it is possible that site t might not have committed T_i yet, e.g., due to network latency or delays in thread scheduling. It would then appear to the client that T_i has never been committed, despite the positive commit result received from $coord_i$.

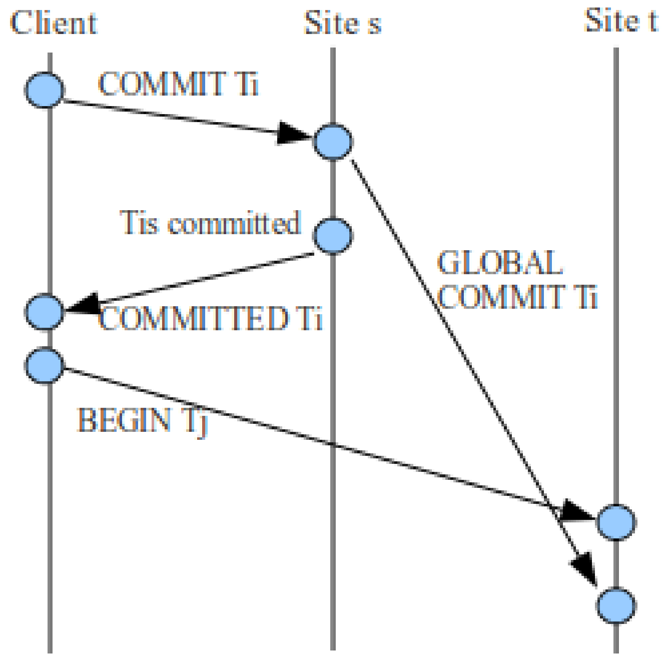

Figure 4.1: Transaction Inversion

This behavior is not unique to CGSI; it has been identified in other lazy systems as a transaction inversion [14]. The opt-commit mode reduces the chance of a transaction inversion but it does not completely eliminate it. To avoid transaction inversions, CGSI offers session guarantees, based on the notion of *minimum visible timestamp* (MVT). Simply, a client transaction T_j can request to see all updates made by all transactions $\{T_i | gct(T_i) \prec MVT\}$.

At the server side, enforcing session guarantees using MVT is straightforward. As shown in Algorithm 4.1, the Committer thread updates `LastCommittedGCT` every time it commits a transaction. The MVT of the oncoming transaction T_j is checked and compared against `LastCommittedGCT`. If `LastCommittedGCT` $\prec MVT$, then T_j is inserted into a priority queue of waiting transactions called WaitingQueue. The WaitingQueue is ordered by the MVT requested by the client, such that the transaction at the root of the queue has the oldest MVT. A separate thread (which can be the Committer thread) will inspect WaitingQueue periodically to see if `LastCommittedGCT` is greater than or equal to the oldest MVT. If so, the transaction at the root of WaitingQueue can be dequeued and processed for execution. The full algorithm to decide whether T_j needs to be blocked is shown in Algorithm 4.2.

Algorithm 4.2 Enforcing Session Guarantee at Site t

```
 1: procedure CHECKSESSIONGUARANTEE($T_j$, $MVT$)
 2:     if $MVT \preceq LastCommittedGCT$ then
 3:         return true
 4:     end if
 5:     if $event\_clock^t \leq MVT$ then
 6:         Update event clock with that in MVT
 7:         if PreparedQueue is empty then
 8:             $LastCommittedGCT \leftarrow$ MVT
 9:             return true
10:         end if
11:     end if
12:     Insert $T_j$ into WaitingQueue tagged with MVT
13:     return false
14: end procedure
```

Note that the event clock is updated by MVT in Algorithm 4.2. A client may contact any site to initiate a transaction. Thus, a global commit timestamp, say, $gct(T_i)$, issued at site s needs to be valid at some other site t. This is needed even when site t was not participating in T_i. When a site in the system communicates its event clock in 2PC messages with another site, they will be synchronized with each other. Depending on the relative frequency of transactions, site s'

$event\_clock^s$ may differ greatly from $event\_clock^t$ at site t. To prevent a client specifying MVT issued by site s from waiting at site t, CGSI performs the following actions:

1. Each site periodically broadcasts their event clock to the other site. This can easily be piggybacked with the heartbeat signal used to detect a lost connection and/or server crash.
2. Treats the MVT as synchronization advice to the other site. When site t receives an MVT with commit timestamp component greater than $event\_clock^t$, site t updates its $event\_clock^t$ to that value.

Proposition 4.5 *For some value β, if site s finds that $event\_clock^s < \beta$, setting $event\_clock^s = \beta$ will ensure that a transaction T_k can have $commit\_tstamp(T_k) \leq \beta$ only if:*

1. *T_k has been committed and is no longer in the PreparedQueue as of then, or*
2. *T_k is still in the PreparedQueue with `ready_to_commit` flag on, or*
3. *T_k is in the "prepared" state as of then and $s \neq coord_k$.*

The updating of $event\_clock^t$ with MVT allows synchronization between sites without direct communication between them. A consequence of Proposition 4.5 is that if site s uses MVT of T_j to update $event\_clock^s$, T_j will never have to wait for any transaction T_k that does not fall into one of those categories since $commit\_tstamp(T_k) > MVT$ assuming T_k eventually does commit. This behaviour is safe because then T_k can only either be an active transaction or a prepared transaction under site s's coordination. In either case, the client could not have requested T_j to see T_k's updates, simply because $commit\_tstamp(T_k)$ has not been assigned yet and MVT of T_j cannot possibly refer to it. Note that if $s \neq coord_k$ and T_k has been prepared, there is a slight chance that $coord_k$ had decided to globally commit T_k and the *global_commit* carrying $commit\_tstamp(T_k)$ is just in transit.

A final note for Algorithm 4.2 is that T_j may also immediately proceed if the PreparedQueue is found to be empty. We can then treat it as if the transaction T_i with $gct(T_i) = MVT$ had been committed at site t because we have allowed T_j which depends on T_i to execute. Hence, we update `LastCommittedGCT` appropriately.

Chapter 5

Replication by Log Merging

In this chapter, we describe how updates of committed transactions at the primary sites that hold the partitioned database are captured in the database logs. We then describe an algorithm for merging the updates from these database logs of primary sites 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 the log information about the transactions are transferred over to the replicas. PostgreSQL employs a Write-Ahead Log (WAL), which we describe next.

5.1 PostgreSQL Write-Ahead Logging

PostgreSQL operates under the Write-Ahead Logging rule and uses a redo log to ensure durability of transactions in the event of failure. Log records can only be appended to the end of the log, which 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. For instance, a tuple deletion operation from T_i might free up space that can be used to store a new tuple created by T_j. Reversing the order of the two would result in the new tuple of T_j being freed by T_i.

While logically the log is a stream of records, they are physically stored in blocks. A block may contain more than one record, and one record is allowed to span multiple blocks. Every 16MB of blocks are organized into one *log segment* which is represented as a file on the filesystem. We utilize the hot standby feature introduced in PostgreSQL 9.0 that allows log records to be transferred as soon as they are generated. Effectively, this enables the replicas to be maintained lazily and with little delay. Prior to PostgreSQL 9.0, one could only transfer a full log segment, meaning that the first log record written into a segment may have to wait for up to 16MB worth of log records to be generated before it gets a chance to be propagated, potentially causing significant delay to processing client transactions at the secondaries.

5.2 Merging Log Streams from Multiple Masters

The straightforward solution to replicate a partitioned database is to designate a set of replicas for each partition, so that each replica hosts a copy of one partition only. Running analytical or multi-join read-only queries involving multiple partitions, however, may incur significant overhead in terms of coordination and intermediate results communication between the replicas. We propose a novel *log stream merging* solution to maintain replicas of multiple partitions under one database instance per secondary site, so that queries involving these partitions can be serviced locally without the need for distributed transactions. Our current solution merges log streams from all masters but it can also be used to merge log streams selectively from only a subset of them.

We design our solution where each partition hosts a distinct set of tables meaning that no table is in more than one partition. This does not mean that our solution is restricted to partitioned tables. PostgreSQL supports a basic partitioning scheme to treat a set of tables as a group [38]. Queries and updates can be executed directly on one specific table or on the whole group. Using this facility, we can still partition a "table" (which is actually a group in this scheme) while enforcing one responsible master for each table rule.

Our solution merges multiple log streams into one single *unified stream*, as shown in Figure 5.1. The master servers, which process update transactions, generate one log stream per server. The log streams are then transferred to the *log merger* component for merging. PostgreSQL already has a messaging protocol in place to ship log records, but we chose to use our own log transfer protocol for simplicity and flexibility reasons.

At the heart of the system is the *log merger* process, which reads log streams from all master servers and merges them into a unified stream. It consists of three main modules with distinct responsibility. The first module, the *log fetcher*, continuously fetches log records with the help of the CGSI Agent present at each master server. Then, the multiple log streams are fed into the *log transformer* to be processed, reordered, and merged into the unified stream. The unified stream is then durably written to the local disk. Finally, the *log sender* module reads the unified stream from the disk, and sends it over to the replica. As the consumer of the unified stream is a PostgreSQL instance, the log sender module implements PostgreSQL's log shipping protocol.

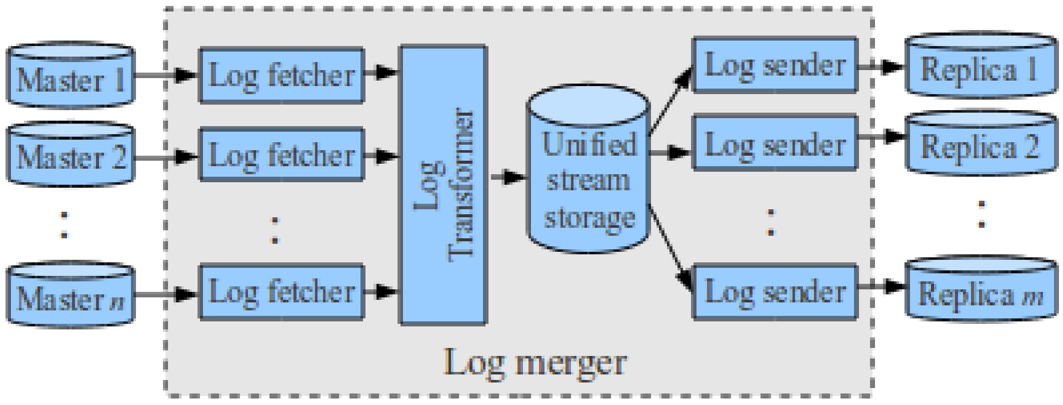

Figure 5.1: Log Merger Modules

Each replica to which the unified stream of updates is to be shipped is a PostgreSQL instance running in hot standby mode. Each replica connects to the log merger component by spawning a *WAL receiver* to grab log records, and a *Startup* process to 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. As we will discuss throughout this section, 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. Each replica also spawns a CGSI agent to process read-only transactions from the client, and to enforce session guarantee, which we discuss later.

We have also considered the option where a log merger exists at each replica, thus each replica can fetch and merge logs independently. While this is possible, each replica will then have to dedicate a portion of its computing resource for log merging, thereby reducing the remaining capacity available to service read-only queries. Furthermore, as each replica needs to fetch complete log records from all master servers, the bandwidth requirement is greater than if only the unified stream is pushed to the replicas (the unified stream contains less records than the sum of the original streams, due to reasons explained in Section 5.2.2). As such, we did not implement this variant as we expect it to be less scalable.

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 snapshot consistency. Finally, we provide session guarantee such that subsequent transactions submitted by the same client see at least the preceeding transaction's effects.

5.2.1 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 with the next stream. If a stream does not have any records available, the stream is temporarily marked *unavailable*. Unavailable streams are skipped for processing for a period of 1 second before they are retried.

Not all log records from the source stream will appear at the unified stream. Log records pertaining to distributed transactions are present in multiple streams, but the unified stream will contain only one combined record. Also, log records related to database checkpoints and commit logs (CLOG) from the source stream cannot be directly placed in the unified stream because the unified stream will be moving at a different pace than any of the source streams.

Each database item will be assigned a unique *object identifier* (OId) when it is created. The OId is heavily used in the log records to identify the item the log record applies to. Now, suppose that we have two master servers, each hosting one table each. It is likely that if we start each master from fresh, both tables will end up with the same OId. Thus, when we process a record specifying that OId, we need to figure out which table it actually refers to. One solution would be to use the source stream to distinguish them. While that works, we use a simpler method to solve this problem. Instead of initializing each master independently, we initialize all of them from a *super image*. The super image contains the union of tables, indexes, and other database items from all master servers, but they are all empty. After a master is initialized from this super image, it will be populated with only the part of data it is responsible for. The replica should also be initialized from the super image. This way, we entirely avoid OId conflicts, and in fact, we can simply leave the OIds as it is since the replicas will have the same understanding of what it refers to.

Transaction identifiers, called XId in short, is another important value which is ubiquitous throughout the log records. Each record has an XId field indicating which transaction performed an operation on it. Additional XId fields are also present in some log record types, for example to specify when an unused disk block can be reused. Whatever the use case, the value of any XId fields must remain valid and correct after the merging. To avoid ambiguity, we will use the term *ReplicaXId* when specifically referring to the XId used in the unified stream. For each master server, we maintain an `XIdMap` containing the mapping between an XId in the source stream to its ReplicaXId. If an XId is found in the map, it will be modified into the mapped ReplicaXId; otherwise, the XId Manager issues a new ReplicaXId and creates the appropriate entry in the master's `XIdMap`. XIds of distributed transactions are treated specially, and we will discuss this separately.

To protect against log corruption, each log record in the stream has a CRC value and a back record pointer. Since the log transformer may modify values in the log record, and it can also reorder records, these two values are likely to be invalidated. Hence, the log transformer needs to recompute the correct values just before placing the record in the unified stream.

5.2.2 Ensuring Snapshot Consistency

As the CGSI and the local concurrency control at each master server have handled the snapshot consistency aspects among update transactions, the log transformer does not have to perform any additional checks, though it still needs to 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. Therefore, the log transformer simply follows the same order for the unified stream.

A distributed transaction may have *prepare*, *abort*, *abort prepared*, and *commit prepared* records in the master log stream. In the unified stream, however, all distributed transactions will be represented as local transactions, i.e. only *abort* and *commit* records will be present. PostgreSQL *commit prepared* and *abort prepared* records are actually supersets of the corresponding *commit* and *abort* records.

PostgreSQL does not identify a transaction as a distributed transaction until *prepare* time. As such, the only identifying information about a transaction before the *prepare* record is its XId. This actually poses a complication as follows. Let us say that log transformer reads a log record mentioning an XId χ. Upon consultation with the stream's `XIdMap`, the system finds it has never seen χ before so then the XId Manager issues a ReplicaXId χ', and maps $\chi \Rightarrow \chi'$ in `XIdMap`. Now, imagine that the transaction referred by χ is a distributed transaction involving another master server, and the XId assigned by the other server is ω. When the log transformer first sees ω, it does not know its correlation with χ; to the log transformer it just appears to be yet another transaction. If χ and ω are two different XIds issued by different master servers, but they refer to the same distributed transaction, they must be mapped to the same ReplicaXId.

To overcome this problem, we introduce a new *AssignGXid* record type which will be inserted whenever PostgreSQL assigns a new XId for a distributed transaction. As XId assignment is mandatory before any log record related to a transaction T_j can be written, *AssignGXid* is guaranteed to be the first record seen that mentions T_j. The *AssignGXid* carries the global transaction Id, enabling correlation between different XIds of the same distributed transaction from different log streams.

Algorithm 5.3 depicts how the log transformer handles transactional log records[1]. First, the record type is examined. If the type is not one we are interested in, the record will simply be

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

Algorithm 5.3 Log Transformer Algorithm to Handle Transactional Log Records

```
 1: procedure TRANSFORMTXNLOGRECORD(master, record)
 2:     if $record.type$ is ASSIGN_GXID then
 3:         if we haven‘t seen this transaction before then
 4:             Initialize data structure $gXact$ for the transaction
 5:             Get a new $replicaXid$ from XIdManager
 6:             Store $gXact$ in the global mapping GXIdMap
 7:         end if
 8:         Store mapping between original record.xid to replicaXid
 9:     else if $record.type$ is COMMIT_PREPARED or COMMIT then
10:         Get transaction data gXact from GXIdMap
11:         Combine commit record with previously seen commit records of gXact
12:         if this is the last commit record for gXact then
13:             Update gct(gXact) from the record
14:             Insert gXact into CommittedQueue
15:         end if
16:     else if $record.type$ is ABORT_PREPARED or ABORT then
17:         Get transaction data gXact from GXIdMap
18:         if gXact is NULL then
19:             Place abort record in the unified stream
20:         else if this is the last abort record for gXact then
21:             Place abort record in the unified stream
22:             Remove gXact from the GXIdMap
23:         end if
24:     else                                                    ▷ Other record types
25:         Place record in the unified stream
26:     end if
27: end procedure
```

Algorithm 5.4 Log Transformer Algorithm to Flush Committed Queue

```
 1: procedure FLUSHCOMMITTEDQUEUE
 2:     $oldestMaster \leftarrow$ the master with the oldest state
 3:     while CommittedQueue not empty do
 4:         $gXact \leftarrow$ the root of CommittedQueue
 5:         if $oldestMaster.state \prec$ gct of $gXact$ then
 6:             break
 7:         end if
 8:         Place the combined commit record of $gXact$ in unified stream
 9:         Remove $gXact$ from CommittedQueue and GXIdMap
10:     end while
11: end procedure
```

placed in the stream (line 24-25). If it is the *AssignGXid* record, we will take note of the global transaction Id and its XId equivalent (line 8). If it is the first time we see that global transaction Id, we also initialize its data structure (line 4-6).

We process *commit* and *commit prepared* records in the same way. After looking up the global transaction Id in the map (line 10), we combine the data from new commit records with any previously-seen commit records for this transaction (line 11). 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` (line 13-14).

A distributed 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 changes. It is also possible that an *abort* record is present without *AssignGXid*, for instance if it is a local transaction. In such a case, we can immediately place the record in the stream (line 19). Otherwise, we delay placing the record until it is the last abort record of the transaction (line 21-22). Note that in all cases, placing the record in the unified stream also includes replacing all XId references with the appropriate ReplicaXIds, recomputing CRC, and fixing the back record pointer.

The transactions queued into `CommittedQueue` are placed in the unified stream using Algorithm 5.4. The log transformer can place the *commit* record of a transaction only when no master server can possibly coordinate a transaction with a lower global commit timestamp (line 5-6 stops the algorithm if this condition is detected). To aid the log transformer in determining this, we add `cgsiState` field into *commit*, *abort*, *commit prepared*, and *abort prepared* log records.

This field captures CGSI state information, such as the event clock and lowest global commit timestamp in the commitQueue at the time the log record was written. We deduce whether a lower global commit timestamp is possible by applying Proposition 4.5 to the CGSI states of all master servers.

Theorem 5.1 *Given that the secondary enforces SI locally, a read-only transaction T_i running at the secondary will execute under global SI.*

Proof Per Theorem 4.4, the primaries are producing a sequence of operations that are consistent with global SI. Also, the log merger generates commit records via the same deterministic total order used to enforce C3 at the primaries. Because updates of a transaction are visible only after the transaction commits, the snapshot S_i seen by a read-only transaction T_i at the secondary will always correspond to a particular global snapshot of the system, such that there exists some T_j where the effects of all transactions $T_k \preceq T_j$ are visible to T_i and the effects of all transactions $T_k \succ T_j$ are not visible to T_i. This is equivalent to executing T_i at the primaries right after T_j has been committed, and since T_i is a read-only transaction, it does not have any *write-write* conflict. Thus, in the absence of deadlock, T_i would have always been successfully committed had it run at the primaries, and T_i has to be consistent with global SI. $\square$

5.2.3 Scheduling Read-Only Transactions and Updates

While read-only transactions never conflict with update transactions under SI, read-only transactions may *interfere* with update log installation 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, ultimately causing excessive delays incurred by future read-only transactions to satisfy session guarantees. 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 now discuss several approaches we explored to deal with these possible interferences. First, we activate the PostgreSQL "vacuum deferral" and "maximum streaming delay" features.

The former avoid 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, after which the read-only transaction will be aborted.

Second, we consider implementing an admission control policy based on the difference between the last received log record and the last record successfully replayed (the *replay lag*). A growing replay lag is a sign that the secondary is unable to keep up with the update stream and is unsustainable in the long run. Whenever the replay lag exceeds a threshold, the secondary will temporarily cease servicing read-only transactions with the hope that the system will be less loaded and be able to catch up.

Third, we also look into thread scheduling to boost 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 the same 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.

Based on our preliminary experiments, a combination of the PostgreSQL "vacuum deferral" and "maximum streaming delay" features with the thread scheduling produce the most consistent results. We found that setting the appropriate replay lag threshold for admission control is not a straightforward issue, and it might benefit from a feedback-loop based control system. Therefore, for all experiments in this paper, we only use the first and third approaches, and we defer the exploration of an admission control policy to future work.

5.2.4 Providing Session Guarantee

To provide session guarantee at the replica, we use a similar mechanism as the one used at the primaries. The biggest difference is that unlike the primaries, the replica does not have a Committer thread that updates the `LastCommittedGCT` value. Instead, `LastCommittedGCT` is updated by the Startup process every time it replays a *commit* using the global commit timestamp included in the log record. Unlike the master, the replicas need not synchronize event clocks as the only authoritative event clock is the one reflected in the unified stream.

5.2.5 Fault Tolerance

First, distributed transaction failure at the primaries can be handled in accordance with the provisions of the 2PC protocol [5]. When the primary goes down, it is possible that some log records have not been transferred to the log merger. Let us say T_i is the last transaction committed at the failed primary according to the log stream received by the log merger. It is possible there is some committed transaction $T_j \succ T_i$ at the failed primary, but T_j's commit record has not been sent to the log merger. To avoid inconsistencies, the log merger cannot generate commit record for any T_k having $gct(T_k)$ greater than $gct(T_i)$. The secondaries can still continue serving read-only transactions, as long as it does not require MVT that is greater than $gct(T_i)$. When the failed primary recovers, the Startup process at that primary will first bring the primary to a consistent database state by replaying log records from the last successful checkpoint and resolving any unfinished 2PC. Any actions taken by the Startup process during the recovery will result in additional records describing the action being written into the log, in particular the resolution of any unfinished 2PC. Once this completes, the database is back in a consistent state, and the log merger can reconnect back to the primary. Since the log merger keeps track of the last successfully received record from each primary (*primary markers*), the log merger simply requests the just-recovered primary to fetch log records starting from that point onwards, and then the log merging proceeds as normal. The log merger also resumes sending the unified log stream to the secondaries, and the secondaries continues replaying records as well. Finally, the CGSI agent reconnects itself with its peers, and the primaries can resume servicing update transactions. At this point, the system as whole is back to being fully operational. Optionally, to avoid prolonged blocking of the log merger when a primary fails, it might also be desirable to setup a hot-standby for each primary, either using the standard PostgreSQL feature [37] or other approaches (e.g. [32]). We would like to emphasize that this is strictly optional, and our system still maintains its consistency regardless.

Second, the secondaries can also fail. However, as no update transactions can happen at the secondaries, the impact is minimal. In the worst case, the client will simply have to find another secondary and resubmit the read-only transaction. To recover the failed secondary, one can replay the log from the last log merger checkpoint, or to replicate from another secondary. Once the secondary catches up with the latest record in the unified log stream, it can start serving read-only transactions again. If we choose to recover the secondary by replaying the logs straight from the log merger, there is no extra implementation necessary, as that is precisely what PostgreSQL Startup process does during database startup.

Finally, the log merger process can fail as well. There is a potential failure window between reading the log records and propagating them to the secondaries. Again, since the log merger keeps track of the primary markers, the log merger can simply resume the merging staring from the primary markers onwards. Alternatively, log merger failures can also be handled by running multiple log merger processes that each read and process the log records in parallel with primary markers used to maintain records in case one of the log merger processes fails.

5.2.6 Parallelizing Log Merging

It is possible to enhance our system to have multiple log mergers, each one responsible to merge a subset of the tables. This is useful for both scalability and fault tolerance. We will briefly discuss how to modify our system to support this, and the impacts of such modification.

As has been mentioned in Section 5.2.1, we initialize the primaries and secondaries from the same *super image* containing the superset of all tables and indexes. As a result, each of the tables and indexes in the database will have a unique object Id (OId) in the system. Each PostgreSQL log record typically describes an action applied to an object that is identified by its OId. Therefore, we can enhance the log merger so that it only processes log records pertaining to some OId values. Effectively, that log merger will only process updates related to some subset of 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. What this means is, if a log merger processes updates for a table then it must also process updates for all of its indexes; otherwise, the index will not be available at the secondaries.

Next, we designate those secondaries to answer read-only transactions that touch those tables only, freeing up other secondaries to service other transaction types. With this setting, we can potentially have one dedicated log merger and a dedicated set of secondaries to answer a particular transaction type. More popular transaction types can be allocated to more secondaries, and vice versa. Finally, a client no longer submits the request to a secondary, but rather to a router that will choose the appropriate secondary instance to answer the client's transaction type.

5.2.7 Limitations

Some specific features of PostgreSQL are currently not supported with the log streams merging solution. While we strive to close all the gaps, PostgreSQL is a feature-rich, evolving system. Nevertheless, the limitations presented here do not prevent the running of workloads.

First, the log streams merger can only keep a limited number of (XId $\Rightarrow$ ReplicaXId) mapping per primary. In PostgreSQL, there can be over a billion XIds, and keeping the full mapping is prohibitively expensive. Currently, we limit the system to track only the mapping of the last 10,000 XIds. However, this limit can be easily adjusted to fit the characteristics of the update workload. Typically, workloads containing long-running update transactions will necessitate a higher limit. We, however, found that this limit is more thcan sufficient to support the TPC-W workload.

Whenever the log streams merger finds an XId that is older than the oldest in the XIdMap, the XId is replaced by the *FrozenXId*. The FrozenXId is a special value used by PostgreSQL to refer to transactions that happened “a long time ago”, at least a billion XIds old from the last generated XId. Effectively, our approach reduces the window before an XId is frozen, but this is not a problem as long as no older transaction is still running at that time.

Second, we turned off the *full page write* feature of PostgreSQL. This feature protects against data corruption due to partially-written blocks. Ordinarily, only the incremental update to a block is logged. With the feature activated, the whole updated disk block will be logged on the first update to the block after a checkpoint. As of now, we do not support transformation of a whole block, although this is a possible future enhancement of our system.

To support block level transformation, the log merger needs to inspect the content of the whole block, and then converts all XId values to the corresponding ReplicaXId values. In some ways, this may degrade the performance, as the log merger needs to process and output more data, and consequently, the secondaries needs to replay more data as well. In any case, there are already file systems and hardware solutions that can prevent partial disk writes, mitigating possible negative effect.

Chapter 6

Experimental Results

In this section, we present results of experiments conducted to study the effectiveness of the techniques proposed in Chapters 4 and 5. We begin with a description of the workloads and hardware configuration used in our experiments.

6.1 Experiment Setup

6.1.1 TPC-W Workload

The TPC-W workload models an online book store serving customers' browsing, ordering, and administrative functions. We use the TPC-W workload with Shopping (80% read-only and 20% update) and Ordering (50 % read-only and 50% update) transaction mixes. The initial database contains 1,000,000 items and 100,000 customers, resulting in a physical database size of approximately 1GB. We used the default TPC-W values for the client think time and session time.

As TPC-W does not specify a particular partitioning scheme to follow, we partition the database with the goal to balance transactional load on each server. The read-only COUNTRY table is 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 master servers. Thus, each master server will have a chunk of each of those five tables. The ORDER, ORDER_LINE, and CC_XACTS tables are not split; they are hosted under only one partition so that the system can perform order processing locally.

A significant proportion of TPC-W update transactions access multiple tables at once. Left as it is, nearly all update transactions will be distributed. With such an unrealistic high percentage of distributed transactions, the protocol overhead will overshadow any increased concurrency benefit of the additional server. To better control the ratio of distributed transactions, we adapt the workload to favor selecting items from the same partition. Also, we reduce address uniqueness check to within a single partition instead of across all partitions. The resulting workload allows us to vary the ratio of distributed transactions from 15% upwards.

In addition to the standard TPC-W workloads, we also use a version with "lighter" read-only transactions, which we call the "TPC-W*" workload. In standard TPC-W, some read-only transaction types are considerably more work-intensive than others, most notably the Search Result, Best Sellers, and New Products web interactions owing to the table scans and multi-joins involved in them. In our custom "TPC-W*" workload, those transactions are replaced with their "lighter" versions that eliminate the table scans and multi joins so that they are more comparable with the other read-only transaction types. This is also to avoid bias in the performance evaluation if one particular run of the experiment executes many more work-intensive read-only transaction types than the other runs, or vice versa.

6.1.2 TPC-C Workload

The TPC-C workload models store order processing across a number of warehouses. We use TPC-C because the workload is easily partitionable where the majority of the transactions concern only one warehouse, allowing it to scale to higher number of servers. Transactions involving more than one warehouses (e.g. ordering an item or processing payment from a different warehouse) are natural candidates for distributed transactions.

We set up our initial database with 12 warehouses, resulting in an initial database size of approximately 1.6GB. 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 only store the data pertaining to its respective set of warehouses.

In TPC-C, the *remote warehouse probability* value specifies the chance that a transaction involves more than one warehouse. However, since a partition may host more than one warehouse, it means the more partitions you create, the higher the distributed update transactions ratio you will have. This puts a bias against the configuration with more partitions because it will incur more overhead for distributed transactions. To eliminate this bias, we tweak the workload so that if the transactions involve more than one warehouse, at least two of the warehouses must be hosted by two different partitions. Effectively, the remote warehouse probability now refers to the probability that an update transaction is a distributed transaction.

The standard TPC-C workload is update intensive, with 92% of the transactions being update transactions (we will thus name this TPC-C 8/92). We can use this 8/92 mix to demonstrate the scalability of the primaries, but we will not be able to show the scalability of the secondaries because they will not receive enough load. To demonstrate the scalability of the secondaries, we will use a customized version called TPC-C* that has a ratio of 70% read-only to 30% update transactions. There are three types of update transactions, and their mix relative to each other remains unchanged. Among the two read-only transaction types, we increase only the number of the relatively "lighter" OrderStatus transaction because the "heavier" StockLevel transaction is not intended to be executed frequently. In fact, the StockLevel transaction has lower consistency requirement where reads in a transaction do not have to be repeatable, i.e. transaction T_i may at first perform $r_i[x_j]$ and then later $r_i[x_k]$, as long as $C(T_k) \prec r_i[x_k] \wedge T_j \prec T_k$. This means that transaction T_i does not necessarily see the same snapshot S_i throughout its life; it is allowed to see a newer snapshot S_i' on subsequent statements. In PostgreSQL, this is the "read committed" transaction isolation level and we use it specifically for StockLevel transactions only.

While TPC-W picks transaction types based on a state machine, TPC-C picks them randomly. In preliminary experiments, this random picking proves to be problematic because it may considerably change the actual output of the mix. We found that TPC-C update transactions have significant contention with each other, and they are more likely to be aborted than the read-only transactions. Especially when the primaries have saturated, a 70/30 mix may shift closer to an 80/20, and as such the results will be difficult to compare with those that are really 70/30. We will show and discuss this briefly later. To overcome this, we enforce transaction retries for TPC-C and TPC-C* such that a transaction – be it update or read-only – will be retried until it is successful. This ensures that the mix will be maintained regardless of the abort rate.

6.1.3 Hardware Configuration

We ran the experiments on a cluster of 16 servers (machines), each equipped with two dual-core 2.4GHz processors, and 8GB of RAM. Each machine stores its data on its local 160GB hard drive running at 10000 RPM. The machines are connected over a router providing dedicated high-speed networking. Each of the primaries and secondaries is hosted on a different machine. The log merger is also hosted on a dedicated machine. All of the clients are hosted on another machine, separate from the primaries, the secondaries, and the log merger. Each data point in our experiment is averaged over 5 runs, with computed 95% confidence intervals marked by the error bars.

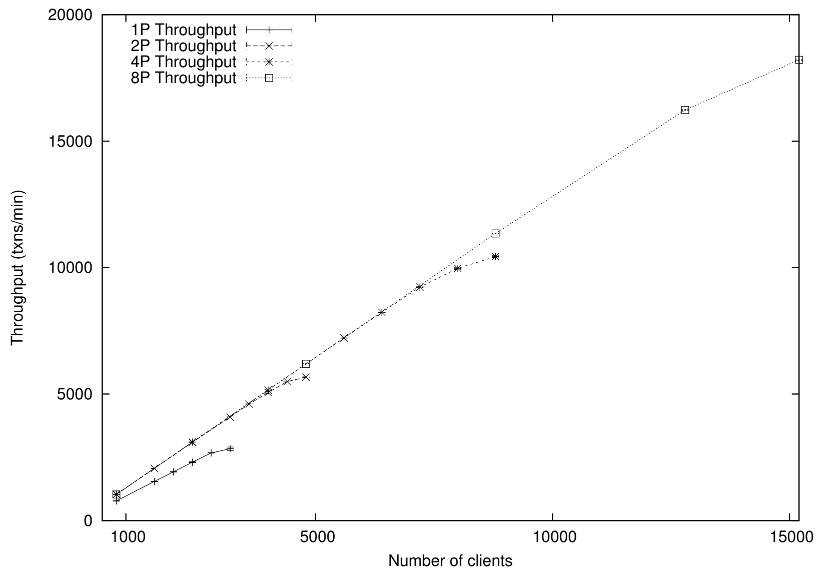

Figure 6.1: 1P/2P/4P/8P Throughput on TPC-W Shopping (80/20)

6.2 Primary Servers Performance

First, we would like to find out the performance of our global SI protocol. We ran the TPC-W Ordering and Shopping workloads, and TPC-C 8/92, while varying the number of clients in the system and the number of partitions (we used 1, 2, 4, and 8 partitions for TPC-W; we used 1, 2, 4, and 12 partitions for TPC-C). We will refer to a system with x partitions as xP (e.g. 4P is a system with 4 partitions). Each primary will host exactly one partition. For the TPC-W Ordering workload, we set the ratio of distributed update transactions to 30%; for the TPC-W Shopping workload, we used a ratio of 20% distributed update transactions; for the TPC-C 8/92, we used the default distributed update transactions ratio. Distributed transactions execute at multiple primary sites based on the data items referenced in each transaction that is part of the TPC-W workload. As we want to focus on the primaries' performance, the clients do not send read-only transactions and simply assume that the read-only transactions are successful and start the think time again.

We can see in Figure 6.1 that using the TPC-W 80/20 workload, the 2P, 4P, and 8P systems can produce nearly two, four, and eight times the throughput of the 1P system respectively. The average response time observed in Figure 6.2 shows that the 2P, 4P, and 8P systems can handle 67%, 200%, and 400% more clients respectively for the same response time as the 1P

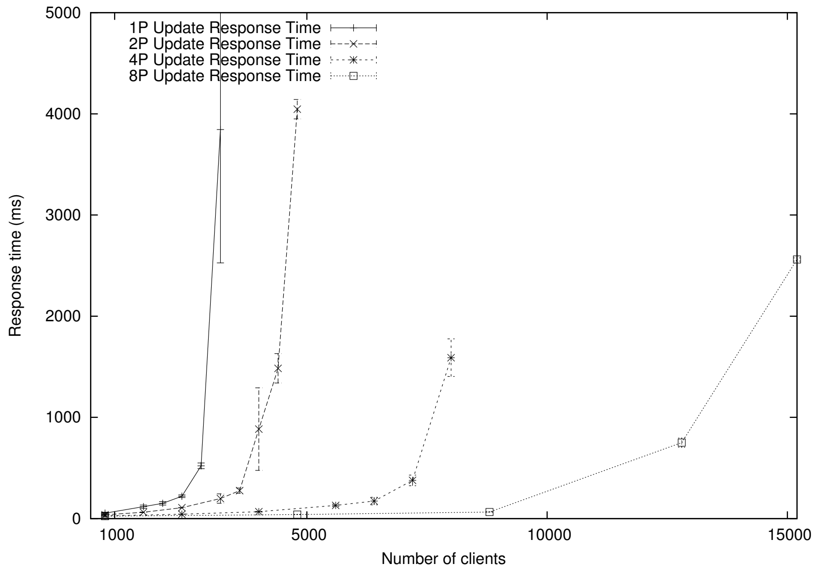

Figure 6.2: 1P/2P/4P/8P Update Response Time on TPC-W Shopping (80/20)

system. The same trend for throughput and response time can also be seen for the TPC-C 8/92 results in Figures 6.5 and 6.6. In particular in Figure 6.5, we can see that CGSI scales really well with partitioned workloads like TPC-C. While we plot all response time data in Figures 6.2 and 6.6, in practice, a response time above 3s may be undesirable, and we include those for completeness only. Doubling the number of primaries does not double the throughput, but this is expected because there is some extra overhead to coordinate transactions among servers. As the number of servers increases, the system administrator can minimize this overhead by designing a partitioning scheme requiring the least number of servers to complete a transaction. Hotspots must also be avoided, although this may be difficult in some workloads. The fact that some of our TPC-W tables are hosted in one partition also reduces performance gain in the 50/50 mix (Figures 6.3 and 6.4), where the proportion of update transactions touching those tables is considerably higher, making them hotspots. Additionally, the ratio of distributed update transactions is also higher in the 50/50 mix.

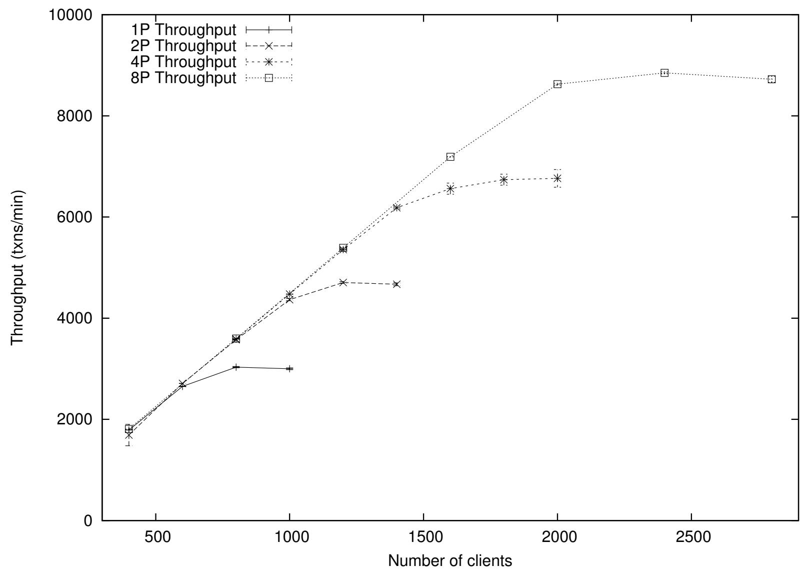

Figure 6.3: 1P/2P/4P/8P Throughput on TPC-W Ordering (50/50)

6.3 Secondary Servers Performance

The secondaries have three roles in the system: serving read-only transactions, forwarding update requests to the primaries, and acting as a backup for the primaries. Of the three, the first one usually takes most of a secondary's resources, especially with the standard TPC-W. The secondaries also need to replay updates from the primaries. Thus, to support a particular number of clients, we expect we would need a higher number of secondaries than primaries. We will discuss our findings on the TPC-W workload first, followed by the TPC-W* and TPC-C* workloads. As has been mentioned, we do not use TPC-C 8/92 to evaluate the secondaries as it does not impose enough load on the secondaries.

6.3.1 TPC-W Workload

We deployed the system varying the number of primaries and secondaries. We use the notation xPyS to refer to a system with x primaries and y secondaries. Due to the noticeably more work-intensive read-only transactions in TPC-W, we run the experiments only with the Ordering (50/50) mix which has a better balanced workload between the read-only transactions and update transactions. It is desirable that the primaries and secondaries are relatively equally loaded to

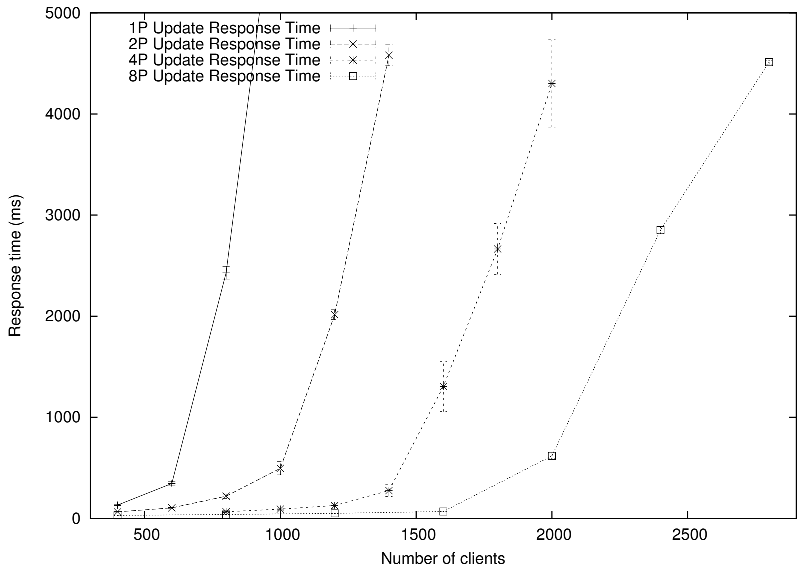

Figure 6.4: 1P/2P/4P/8P Update Response Time on TPC-W Ordering (50/50)

demonstrate scalability. From our preliminary experiments with the Ordering workload, 125 clients exert a moderate load on one secondary so we use this ratio of clients per secondary to scale up the workload while scaling up the number of secondary sites. Figure 6.7 shows the throughput of multiple configurations of an xPyS system. The number of secondaries y is proportional to the number of clients (e.g. we deployed 4 secondaries to serve 500 clients, 6 secondaries to serve 750 clients, etc). At 500 clients, we do not observe any significant difference across all tested configurations. As we increase the load to 750 clients, the update response time for 1PyS starts to shoot up, as can be seen in Figure 6.8. Further increasing the load to 1000 clients results in a significant jump of the update response time, but only a slight increase in the read response time (Figure 6.9). From this observation, the single primary becomes the bottleneck at about 1000 clients, limiting the overall system throughput.

This bottleneck is removed by adding another primary, resulting in a 2PyS system. Unlike the 1PyS system, we do not observe the leveling out of performance after 750 clients although the performance grows at a slower rate between 1000 and 1250 clients. Another important observation is that for the same value of y, the read response time of a 2PyS system is generally higher than that of a 1PyS system. This is expected, because we need to merge log streams in the 2PyS system, and a distributed transaction cannot be replayed until its commit record appears in both streams.

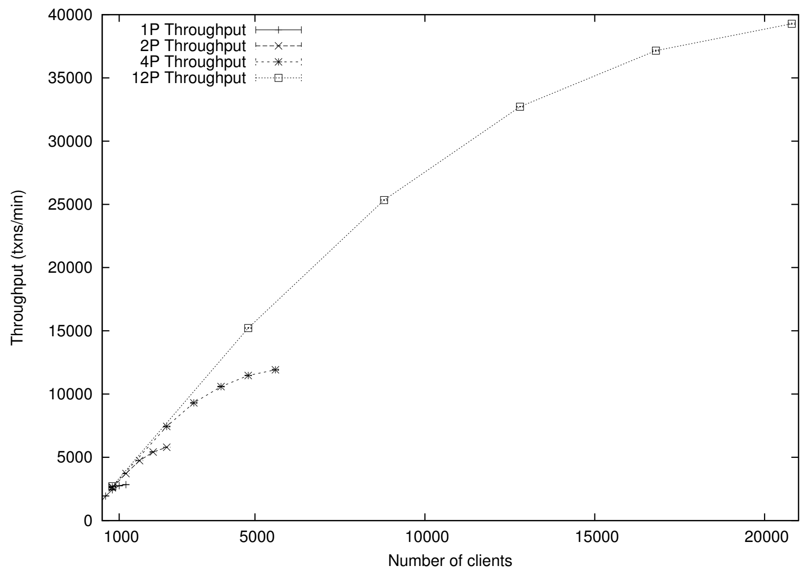

Figure 6.5: 1P/2P/4P/12P Throughput on TPC-C 8/92

For the 4PyS configuration, the performance deviates from the 2PyS configuration starting at around 1000 clients. This is because two primaries have sufficient capacity to service up to 1000 clients, so additional primaries do not help much with the throughput until there are more than 1000 clients. The update response time of the 4PyS system is also lower than the 2PyS system, most notably when the workload scales up to beyond 1000 clients. Additional primaries also help lowering the read response time slightly, as evidenced by the difference in 2PyS and 4PyS curves in Figure 6.9. The less-loaded primaries in 4PyS are able to push log records faster to the log merger, helping the log stream to arrive sooner at the secondaries, thus satisfying the required session guarantee earlier and lowering the response time.

6.3.2 TPC-W* Workload

The relatively heavy read-only transactions in TPC-W cause the secondaries to be saturated much earlier than the primaries. Figures 6.7, 6.8, and 6.9 suggest that it takes approximately 6 secondaries to saturate the single primary with TPC-W workload. To examine the xPyS system performance with higher number of primaries, we switched to use the TPC-W* workload. Our expectation is that with lighter read transactions, a secondary can support more transactions, thus letting us examine the scalability of the system with more clients. For the TPC-W* experiments,

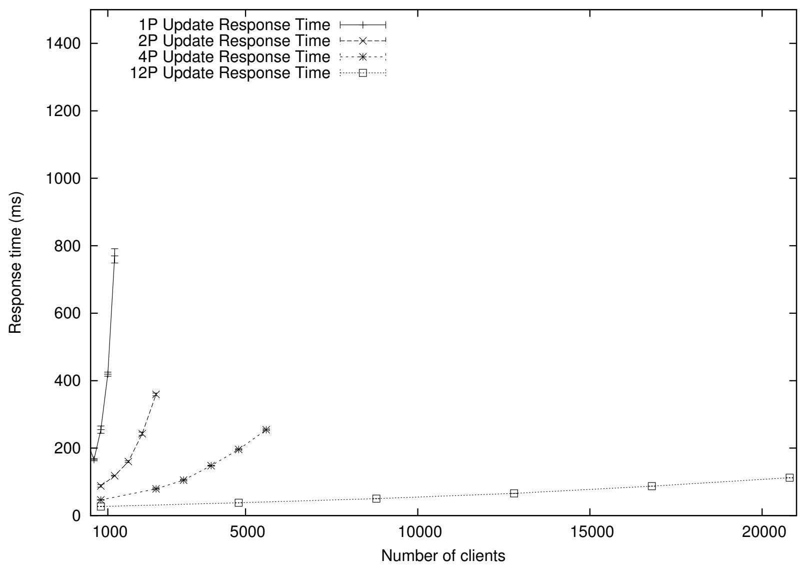

Figure 6.6: 1P/2P/4P/12P Update Response Time on TPC-C 8/92

we assign 150 clients per secondary deployed in the system (e.g. we deployed 4 secondaries to serve 600 clients, 6 secondaries to serve 900 clients, etc).

As shown in Figure 6.10, we can see that the 1PyS system still saturates at approximately 750 clients, and the single primary remains a bottleneck as indicated by the jump in update response time in Figure 6.11. If we double the number of primaries – making it a 2PyS system – the system now saturates at around 1200 clients, at which point the two primaries are becoming the bottleneck. With the TPC-W* workload, the 2PyS system can support approximately 50% more transactions than the 1PyS system.

When we double the number of primaries again to 4PyS, the system can support many more transactions, although we were unable to record the peak throughput capacity of the 4PyS system under TPC-W* workload due to the limited number of machines available in our experimental cluster. It has been demonstrated in Figure 6.10, however, that at the very least the 4PyS system could achieve double the throughput of the 1PyS system. It has to be noted that this is the lower bound, as the 4PyS system is still not saturated at 1500 clients.

It is worthwhile to note that the results in Figure 6.12 further support the earlier observation that in addition to lowering the update response time, adding primaries also lowers the read response time. Thus, for some existing xPyS system, given the possibility to add one extra server, the system administrator should carefully examine which one between a $(x + 1)$PyS and

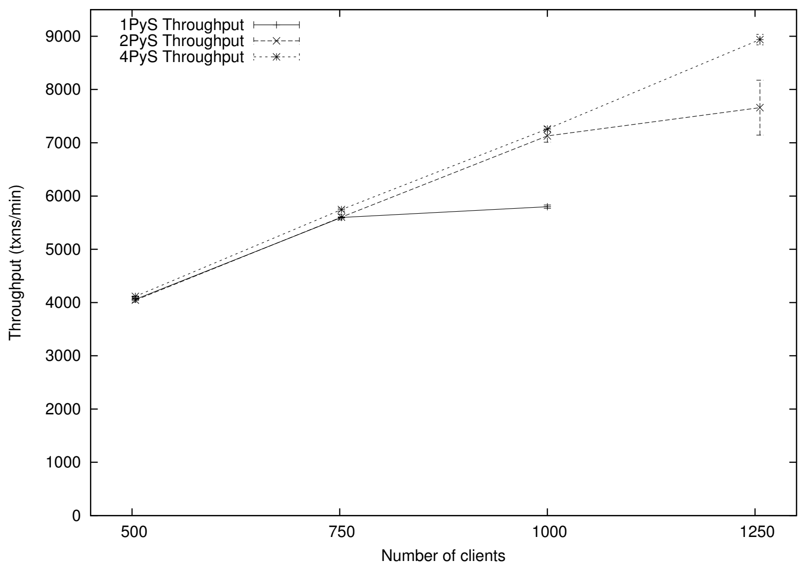

Figure 6.7: xPyS Throughput on TPC-W Ordering (50/50)

a xP$(y + 1)$S system provides greater benefit.

6.3.3 TPC-C* Workload

Similar to the TPC-W and TPC-W* experiments, we assign a set number of clients for each secondary in the TPC-C* experiments. First, we present our preliminary TPC-C* experiment which led us to implement transaction retries. For this experiment, we found that a secondary can serve 800 clients with moderate load. In Figure 6.13, 4PyS shows only 10% improvement in throughput over 2PyS. However, when we separate the throughput into read-only and update transaction throughput, we see a different trend between 2PyS and 4PyS. The 2PyS update throughput peaks around 4000, and adding more clients does not increase it any further. The 4PyS system keeps going on, and at 6400 clients it produces 50% more update throughput than 2PyS. What happened was, at that point the 2PyS system failed almost half of the update transactions resulting in only 20% of all transactions being update transaction. The 2PyS system then produced more read-only transaction throughput than the 4PyS system for two reasons: (1) a read-only transaction executing immediately after a failed update transaction is more likely to satisfy the session guarantee as the secondary has more time to install the update, and (2) each secondary in 4PyS has to work harder to install the extra 50% updates than 2PyS.

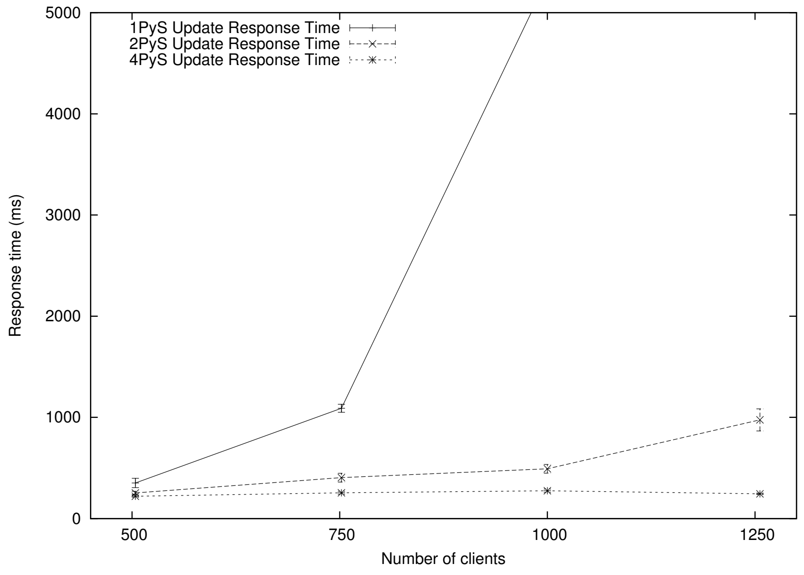

Figure 6.8: xPyS Update Response Time on TPC-W Ordering (50/50)

The remainder of the TPC-C* results were run using transaction retries. Retrying transactions puts more stress on both primaries and secondaries, and we found that a secondary can then serve 500 clients with moderate load. Similar to our observations on TPC-W and TPC-W* workloads, we can see in Figure 6.14 how on TPC-C* workload the single primary of 1PyS quickly becomes the bottleneck, and adding additional primaries and secondaries allows the system to produce more throughput. Past 3000 clients, the 2PyS is reaching saturation point and its throughput is leveling out. The 4PyS still shows linear scalability past this point. Figure 6.15 further shows the saturation point of the 1PyS and 2PyS primaries. We can also observe in Figure 6.16 how the additional primaries of 4PyS lower the read response time by 50% compared to 2PyS when the primaries of 2PyS are saturated at around 4000 clients. This observation confirms the earlier findings from the TPC-W and TPC-W* experiments that adding primaries also helps lowering the read response time.

6.3.4 Cost of Enforcing Session Guarantee at the Secondaries

We would like to examine the cost of enforcing session guarantee at the secondaries. We ran the TPC-W* 50/50 workload again but without session guarantee at the secondaries. Without session guarantee, the read-only transactions are executed immediately after being submitted to

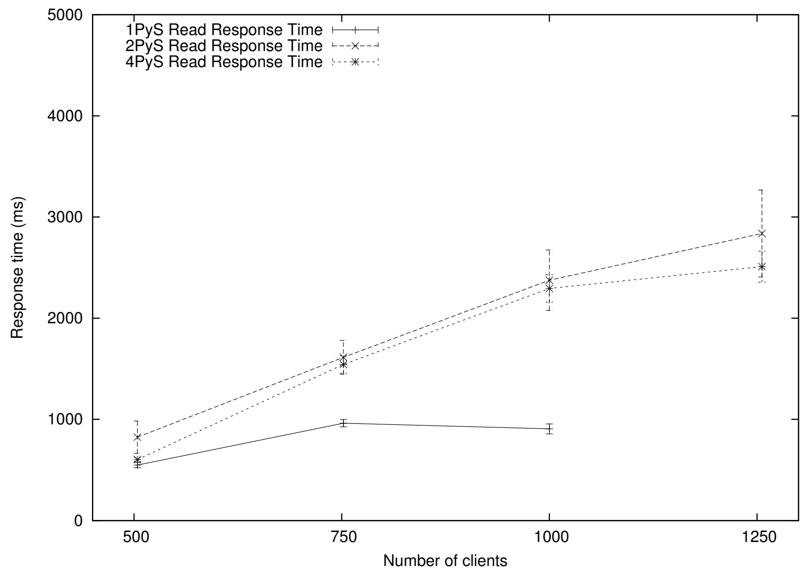

Figure 6.9: xPyS Read Response Time on TPC-W Ordering (50/50)

the system even when the update requested in the MVT has not been installed yet. For the course of this discussion, we use the term “session guarantee” to specifically mean the session guarantee applied to read-only transactions at the secondaries.

As can be seen in Figure 6.17, turning off session guarantee increases the overall peak throughput for the 2PyS and 4PyS configurations by 6% and 8% respectively. We do not plot the 1PyS throughput because there is no difference from when we use session guarantee. The lack of increase in the 1PyS system is because the single primary is the sole contributor to the performance bottleneck observed in Figure 6.10; removing session guarantee will not help this in any way.

For the 2PyS and 4PyS system, turning off session guarantee reduces the read response time to less than 200ms (Figure 6.19) while increasing the update response time slightly (Figure 6.18). The reduction in the read response time compared to Figure 6.12 is due to the read-only transactions not having to wait for the session guarantee to be satisfied. This is also indicative of the propagation delay from producing the log at the primaries to getting the log replayed at the secondaries. Deploying additional primaries helps to reduce this delay, with a reduction of nearly 40% with the 4PyS system over the 2PyS system at 1500 clients. Since the clients are able to finish read-only transactions faster, the clients can also send more update transactions over the same duration. Effectively, this places a higher load at the primaries while eliciting higher throughput

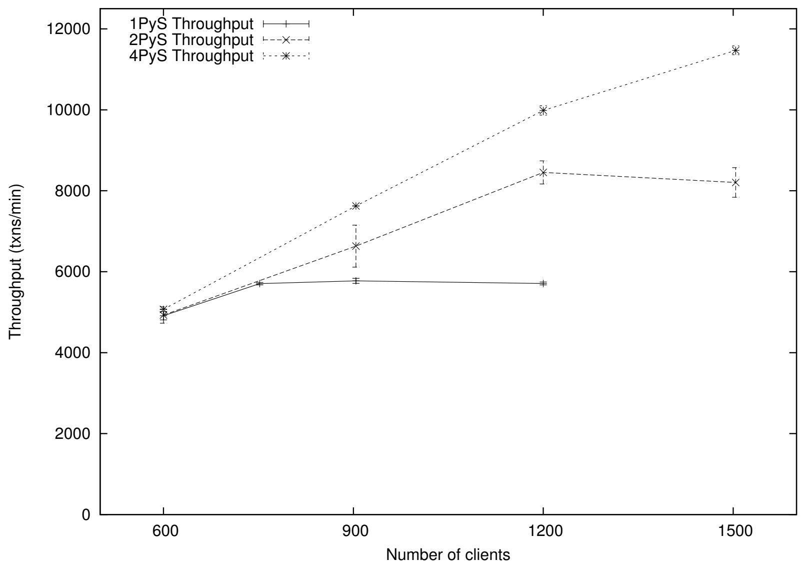

Figure 6.10: xPyS Throughput on TPC-W* Ordering (50/50)

and at the same time increasing the update response time. The cost of enforcing session guarantee is noticeably smaller on the TPC-W* 80/20 workload mix, where the read response time curves of xPyS systems with session guarantee being only slight above those without session guarantee (Figure 6.20), up until the point where the system is reaching saturation.

We also ran the TPC-C* 70/30 workload again without session guarantee. The performance comparison between with and without session guarantee is presented in Figures 6.21, 6.22, and 6.23. For TPC-C*, turning off session guarantee does not produce significant throughput difference until the system is nearing saturation, as experienced by 2PyS at 4000 clients (the 4PyS curves are still indistinguishable at this point). This is different from TPC-W and TPC-W* where the throughput curves start diverging early. The reason is that for TPC-C*, even with session guarantee, read-only transactions do not need to wait that long, as evidenced from only a slight increase to read response time in Figure 6.23. This shows that the cost of session guarantee varies from workload to workload depending on factors such as transaction mix, transaction profile, think time, data distribution, etc.

Across all experiments, we did not encounter the single log merger site to be a bottleneck. With higher number of primaries, there may come a point where the log merger is unable to fetch and transform the log records as fast as the records being created. One feasible solution is to deploy multiple log mergers, as has been described in Section 5.2.6.

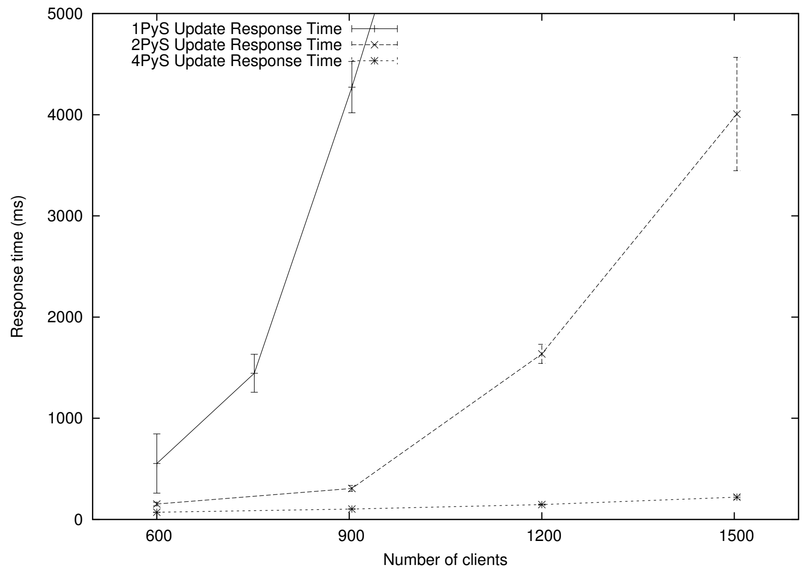

Figure 6.11: xPyS Update Response Time on TPC-W* Ordering (50/50)

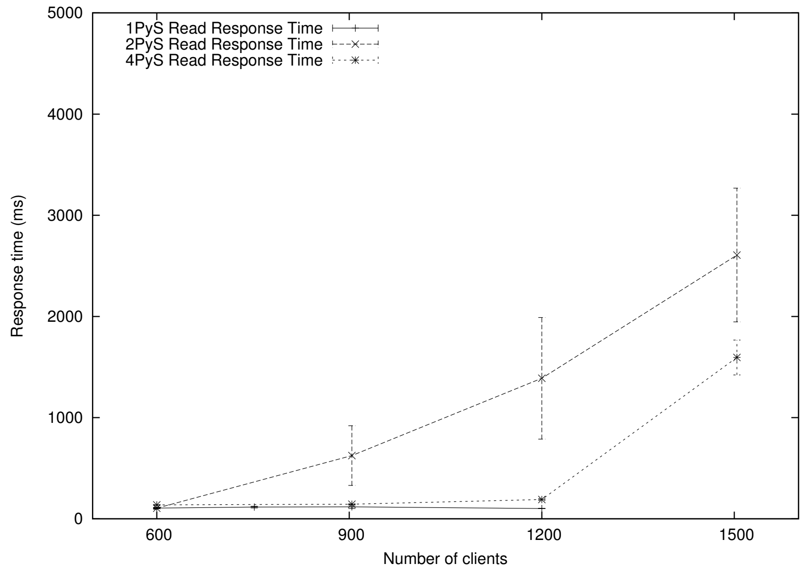

Figure 6.12: xPyS Read Response Time on TPC-W* Ordering (50/50)

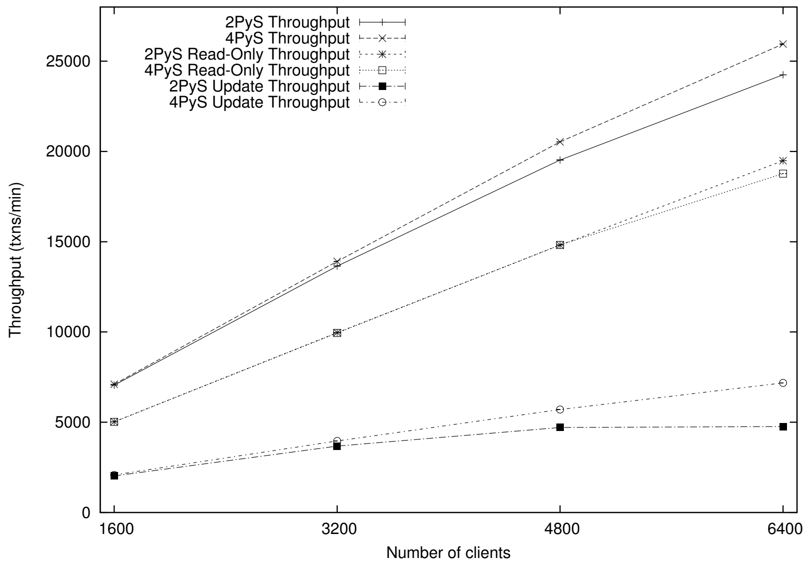

Figure 6.13: xPyS Throughput on TPC-C* 70/30 Without Transaction Retries

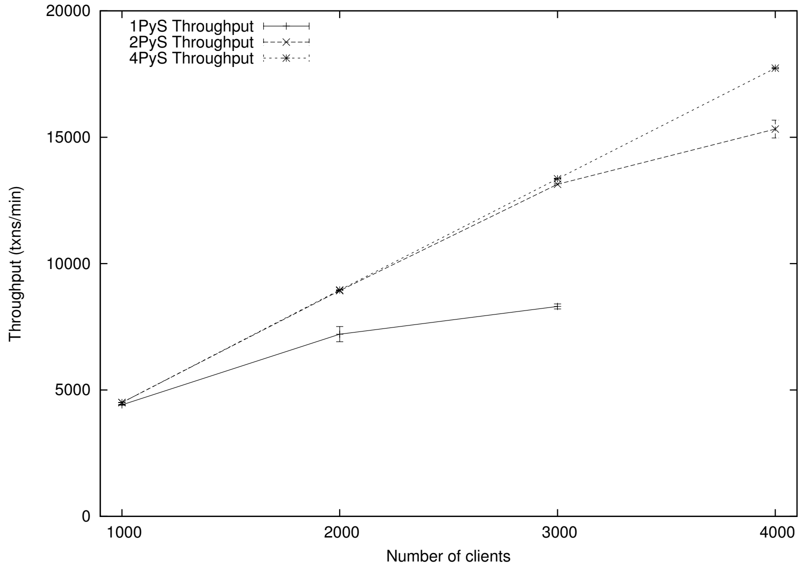

Figure 6.14: xPyS Throughput on TPC-C* 70/30 With Transaction Retries

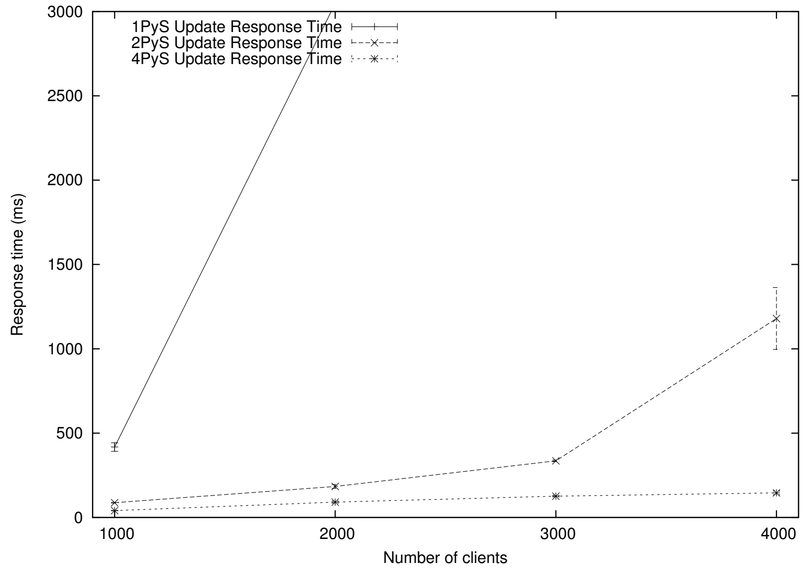

Figure 6.15: xPyS Update Response Time on TPC-C* 70/30 With Transaction Retries

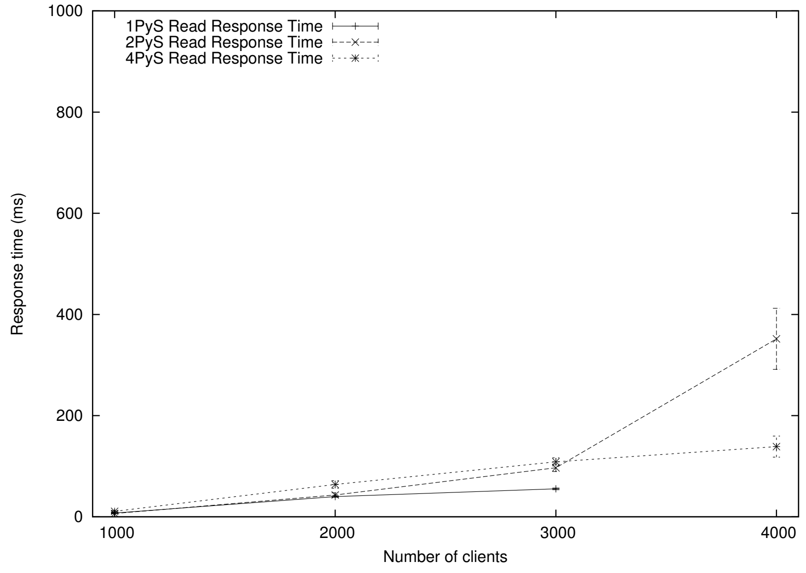

Figure 6.16: xPyS Read Response Time on TPC-C* 70/30 With Transaction Retries

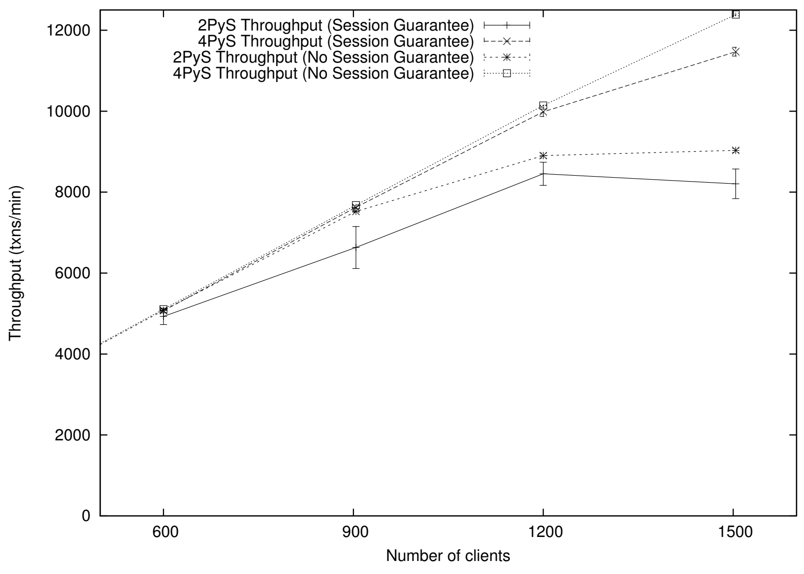

Figure 6.17: xPyS Throughput on TPC-W* Ordering (50/50), With and Without Read Session Guarantee

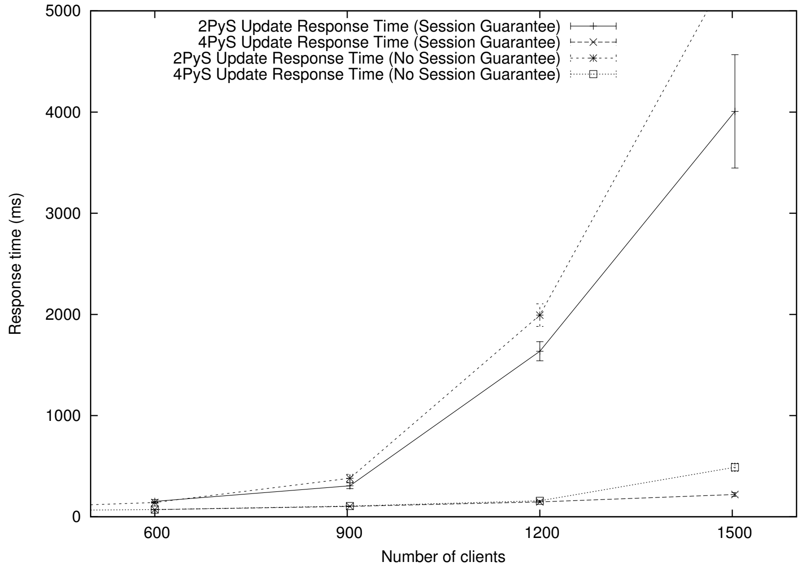

Figure 6.18: xPyS Update Response Time on TPC-W* Ordering (50/50), With and Without Read Session Guarantee

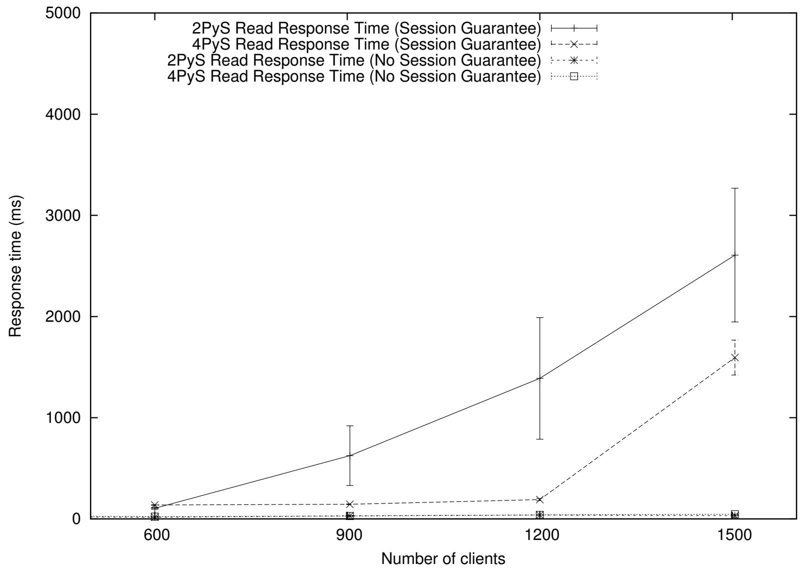

Figure 6.19: xPyS Read Response Time on TPC-W* Ordering (50/50), With and Without Read Session Guarantee

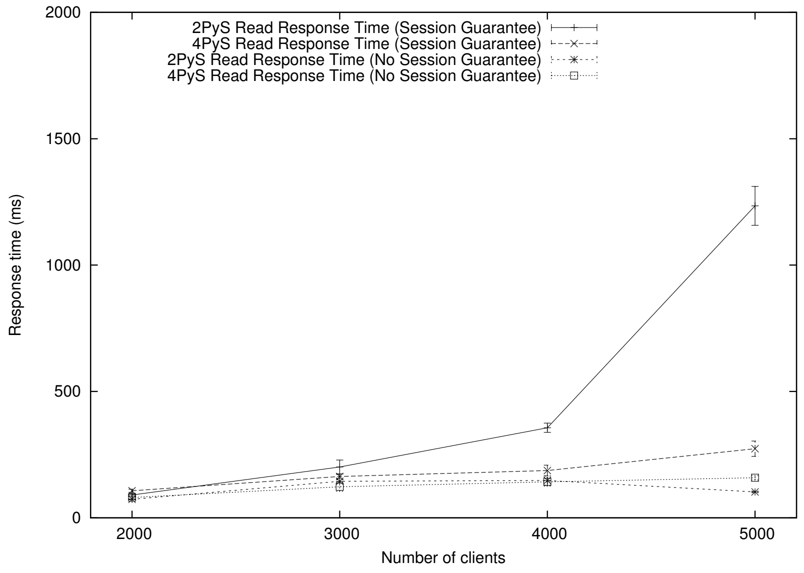

Figure 6.20: xPyS Read Response Time on TPC-W* Shopping (80/20), With and Without Read Session Guarantee

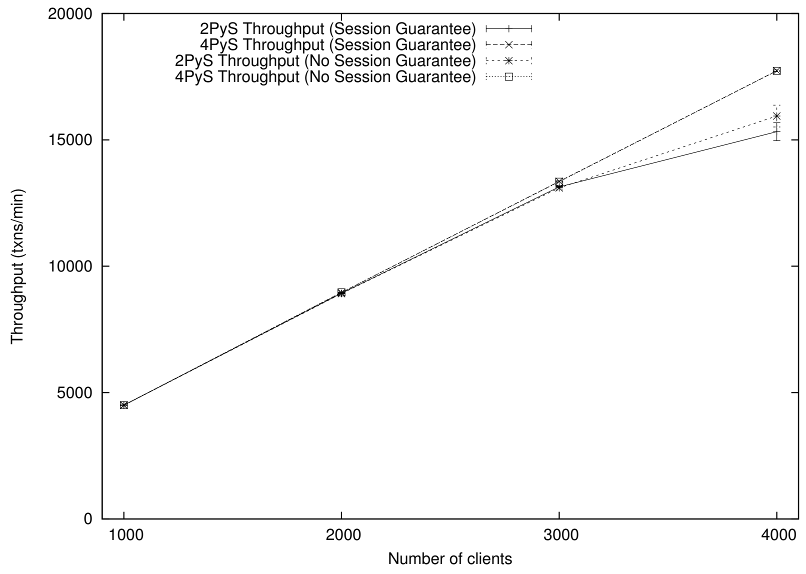

Figure 6.21: xPyS Throughput on TPC-C* 70/30, With and Without Read Session Guarantee

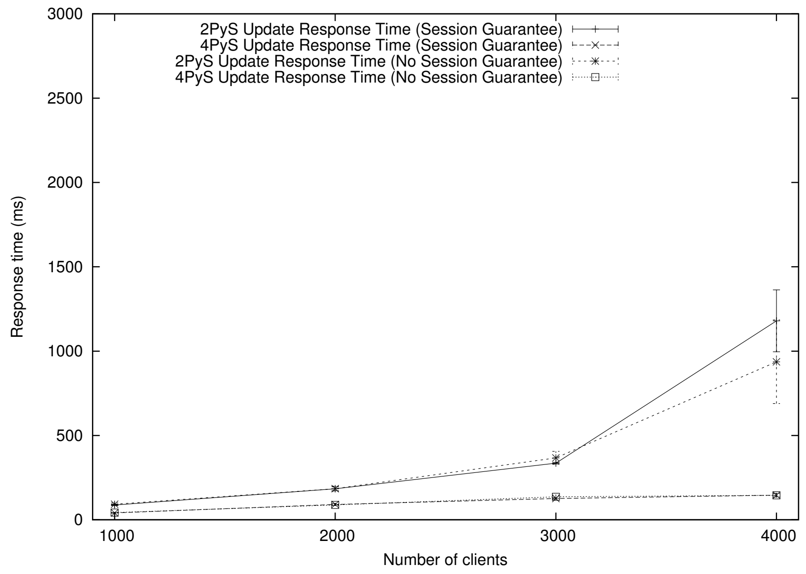

Figure 6.22: xPyS Update Response Time on TPC-C* 70/30, With and Without Read Session Guarantee

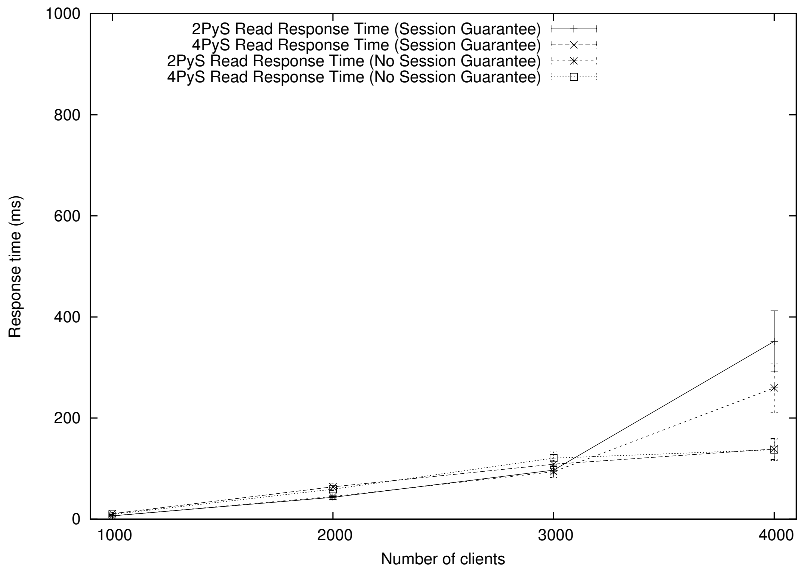

Figure 6.23: xPyS Read Response Time on TPC-C* 70/30, With and Without Read Session Guarantee

Chapter 7

Related Work

In [39], Schenkel et al. proposes algorithms to achieve global SI on federated databases. They outlined the pessimistic and optimistic algorithms to ensure global SI, but both algorithms depend on a centralized coordinator to issue *begin* and *commit* operations on the database. Bornea et al. [6] 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 partitioning the database to scale-up. Moreover, we maintain replicas by deriving an SI-consistent update order using database logs.

A number of cloud-based databases or key-value stores offer the eventual consistency level with limited or even no support for transactions at all. There have been various works to implement multi-row transaction support providing global SI using HBase as the underlying database [47, 34, 28].

Spanner [10] is an SQL-like database that guarantees external consistency for all reads and writes. It has been used in Google as the data store for their advertising backend. For its concurrency control, Spanner depends on highly synchronized clocks across the data centers. It has been noted by the authors that they need to use both GPS and atomic clocks with independent failure models. This demonstrates the difficulty in keeping clock synchronization across a distributed system, which our protocol avoids. Additionally, Spanner was built from the ground up to support reading data with older timestamp. In contrast, our protocol does not need this capability from the underlying database, and is thus more feasible to install on top of an existing database.

Session-based SI guarantees have been proposed in [14, 15, 23] but none of these consider scaling up through partitioning.

A weaker version of SI, the Parallel SI (PSI), is described in [40], specially targeting geographically distributed systems. There can be multiple primary sites in PSI, but each primary site may perform update in different order. In other words, there is no global commit order. As described in Chapter 4, our distributed solution has a strict commit order, ensuring greater transaction consistency among the primaries and secondaries.

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 [1], which is a transparent synchronous solution for partitioned SI databases using Postgres as the underlying DBMS. However, Postgres-XC uses a centralized global transaction manager to assign transaction identifiers and snapshots [36]. C-JDBC [9] is an open-source system which 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. On top of that, the concurrency control and isolation level are still handled by the single scheduler in the system.

Prior to the introduction of replication feature on PostgreSQL, there has been several works on replicating PostgreSQL instances. The Postgres-R, and later Postgres-R(SI) [46] systems are built into PostgreSQL to provide eager replication, update everywhere protocol over a cluster of instances. Another system based on Postgres-R, called the RSSI [20], offers a Serializable Snapshot Isolation consistency level. These works only consider the case where the database is fully replicated. In contrast, our approach allows the primary sites to be partitioned, and each replica can contain data from one or more partitions.

Remus-DB [32] offers a novel approach to database replication on virtualized environment. This is particularly applicable to cloud environment, where database instances run on top of a hypervisor. By carefully hooking to the virtual machine, a database server can be replicated with almost no change to the database engine itself. Again, this approach only provides full replication of a database instance, and does not allow partitioning of primary sites, or merging several primary sites into the same replica.

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 [17]. LogBase [44] is a log-storage system, where updates are stored as log records instead of in disk blocks. Using log records in database replication has been implemented in PostgreSQL, but it only supports full replication of one database instance. [13, 26] explored the feasibility of merging multiple log streams to infer a global serialization order. In contrast, our work focuses on partitioned SI databases.

Slony-I [45] is a replication system which uses triggers to monitor data changes and produces equivalent SQL statements for them. The SQL statements can then be replayed on the replicas

[1]Postgres-XC is available from http://postgres-xc.sourceforge.net/

to generate the same effect. However, this means that the replicas need to parse and plan the SQL statements again and perform the actual modification work again. Also, if two statements modify the same data item, they must be correctly time so as not to raise Write-Write conflict inadvertently. As our log merging solution uses lower-level PostgreSQL log stream, there is no penalty from SQL parsing and planning. PostgreSQL log stream is also free from Write-Write conflicts as it has been taken care of by the concurrency control.

Chapter 8

Conclusions

In this thesis, we described the CGSI algorithm to provide global SI on partitioned SI databases without requiring a centralized coordinator, and without the need of global time synchronization. We also developed the log merging technique to efficiently infer transaction order and to enable the partitioned database to be lazily replicated onto secondary servers. To demonstrate the practicality and feasibility of our protocol, we implemented it on top of PostgreSQL.

Through experiments using the TPC-W and TPC-W* workloads, we showed how the combination of our solution enables the system to support many more of both update and read-only transactions with good performance. On the primary-only TPC-W experiments, our 4P configuration produces twice the throughput of the 1P configuration on 50/50 mix, while it produces nearly four times the throughput of the 1P configuration on 80/20 mix. The easily-partitioned TPC-C workload scales really well on the primary-only experiment, with near linear scalability observed on up to 12 primaries.

We have also shown the scalability of our log merging solution for the secondaries. The 4PyS produces 50% and 100% more throughput than 1PyS on TPC-W and TPC-W* workloads respectively with 50/50 mix, while supporting 70% more clients. On TPC-W* 50/50 mix with higher number of clients, the 4PyS reduces read response time by 37% while also reducing update response time by 90% over 2PyS. With TPC-C* workload, the 2PyS configuration can produce almost double the throughput of the 1PyS configuration. Read-only transactions may have to wait to enforce session guarantee, but the wait time is fairly small: on TPC-C* with 4PyS, the wait time is around 100ms at 4000 clients.

8.1 Future Work

Our implementation relies on PostgreSQL Startup process to replay the updates at the secondaries. In the course of our experiments, we found that configuring PostgreSQL recovery to properly work is not a trivial issue. As has been noted in Section 5.2.3, PostgreSQL only has a coarse mechanism to manage interference between the read-only transactions and refresh transactions at the secondary. Devising our own replay algorithm may let us minimize the interference, thus improving the throughput.

Another interesting venue is to use a feedback-based admission control policy at the secondary, so that the secondary can guarantee progress of the log replay. In our current implementation, we isolate the Startup process to a specific CPU core, thereby avoiding the process from being blocked by other database processes due to unavailability of free processors. Unfortunately, the Startup process could still be blocked by table locks, disk access requests, etc. We feel this can be more effectively handled by deciding whether to admit new read-only transactions based on the current "replay lag", disk activity level, and other metrics.

References

[1] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992*. 1992.

[2] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *ACM/IFIP/USENIX International Middleware Conference*, pages 282–304, 2003.

[3] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Trans. Database Syst.*, 13(3):263–304, 1988.

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. SIGMOD '95, 1995.

[5] Philip Bernstein and Eric Newcomer. *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., 1997.

[6] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-copy serializability with snapshot isolation under the hood. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 625–636, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Update propagation protocols for replicated databases. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 97–108, June 1999.

[8] Yuri Breitbart and Henry F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173–184, May 1997.

[9] Emmanuel Cecchet. C-jdbc: A middleware framework for database clustering. *IEEE Data Engineering Bulletin*, 27(2):19–26, June 2004.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally-distributed database. *OSDI*, 2012.

[11] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE)*, pages 424–435, 2004.

[12] Khuzaima Daudjee and Kenneth Salem. A Pure Lazy Technique for Scalable Transaction Processing in Replicated Databases. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems*, 2005.

[13] Khuzaima Daudjee and Kenneth Salem. Inferring a serialization order for distributed transactions. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 154–, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006.

[15] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. SIGMOD '04, 2004.

[17] IBM. *Log stream merging and log file management in a DB2 pureScale environment*. http://pic.dhe.ibm.com/infocenter/db2luw/v9r8/index.jsp ?topic=%2Fcom.ibm.db2.luw.sd.doc%2Fdoc%2Fc0056149.html.

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

[19] Ricardo Jimenez-Peris, M. Patino-Martinez, Gustavo Alonso, and Bettina Kemme. Improving the scalability of fault-tolerant database clusters. In *International Conference on Distributed Computing Systems*, pages 477–484, 2002.

[20] Hyungsoo Jung, Hyuck Han, Alan Fekete, and Uwe Röhm. Serializable snapshot isolation for replicated databases in high-update scenarios. *PVLDB*, 4(11):783–794, 2011.

[21] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[22] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[23] Konstantinos Krikellas, Sameh Elnikety, Zografoula Vagena, and Orion Hodson. Strongly consistent replication for a bargain. In *ICDE*, pages 52–63, 2010.

[24] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[25] Per-Ake Larson, Jonathan Goldstein, and Jingren Zhou. MTCache: Mid-Tier Database Caching in SQL Server. In *Proceedings of the 20th International Conference on Data Engineering*, pages 177–188, 2004.

[26] Chengfei Liu, Bruce G. Lindsay, Serge Bourbonnais, Elizabeth Hamel, Tuong C. Truong, and Jens Stankiewitz. Capturing global transactions from multiple recovery log files in a partitioned database system. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 987–996, 2003.

[27] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Transaction time support inside a database engine. In *ICDE*, 2006.

[28] Yang Lu. Serializable Snapshot Isolation in Shared Nothing, Distributed Database Management Systems. Technical report, Brown University, 2012. Available at http://www.cs.brown.edu/research/pubs/theses/masters/2012/lu.pdf.

[29] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611, 2002.

[30] Salvatore T. March and Sangjyu Rho. Allocating data and operations to nodes in distributed database design. *IEEE Trans. Knowl. Data Eng.*, 7(2):305–317, 1995.

[31] Daniel A. Menasce, Virgilio Almeida, Rudolf H. Riedi, Flavia Ribeiro, Rodrigo C. Fonseca, and Wagner Meira Jr. In search of invariants for e-business workloads. In *ACM Conference on Electronic Commerce*, pages 56–65, 2000.

[32] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. Remusdb: transparent high availability for database systems. *The VLDB Journal*, 2013.

[33] Ragnar Normann and Lene T. Ostby. A theoretical study of 'snapshot isolation'. ICDT '10, 2010.

[34] Vinit Padhye and Anand Tripathi. Scalable transaction management with snapshot isolation on cloud data management systems. In *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 542–549, 2012.

[35] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware 2004, ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-20, 2004, Proceedings*, volume 3231 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2004.

[36] Postgres-XC Development Group. *GTM and Global Transaction Management*. http://postgres-xc.sourceforge.net/docs/1_0/xc-overview-gtm.html.

[37] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.1: Hot Standby*, 2012. http://www.postgresql.org/docs/9.1/static/hot-standby.html.

[38] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.1: Partitioning*, 2012. http://www.postgresql.org/docs/9.1/static/ddl-partitioning.html.

[39] Ralf Schenkel, Gerhard Weikum, Norbert Weissenberg, and Xuequn Wu. Federated transaction management with snapshot isolation. In *Selected papers from the Eight International Workshop on Foundations of Models and Languages for Data and Objects, Transactions and Database Dynamics*, pages 1–25, London, UK, UK, 2000. Springer-Verlag.

[40] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[41] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[42] Transaction Processing Performance Council. *TPC Benchmark W (Web Commerce)*, Feb 2001. http://www.tpc.org/tpcw/default.asp.

[43] Transaction Processing Performance Council. *TPC Benchmark C*, april 2005. http://www.tpc.org/tpcc/default.asp.

[44] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: a scalable log-structured database system in the cloud. *Proc. VLDB Endow.*, 5(10):1004–1015, June 2012.

[45] Jan Wieck. Slony-i, a replication system for postgresql. http://gborg.postgresql.org/project/slony1.

[46] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433, 2005.

[47] Chen Zhang and Hans De Sterck. Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase. In *GRID*, pages 177–184, 2010.

[48] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.