Lazy Database Replication with Snapshot Isolation

Khuzaima Daudjee and Kenneth Salem
University of Waterloo
{kdaudjee,kmsalem}@db.uwaterloo.ca

ABSTRACT

Snapshot isolation is a popular transactional isolation level in database systems. Several replication techniques based on snapshot isolation have recently been proposed. These proposals, however, do not fully leverage the local concurrency controls that provide snapshot isolation. Furthermore, guaranteeing snapshot isolation in lazy replicated systems may result in transaction inversions, which happen when transactions see stale data. Strong snapshot isolation, which is provided in centralized database servers, avoids transaction inversions but is expensive to provide in a lazy replicated system. In this paper, we show how snapshot isolation can be maintained in lazy replicated systems while taking full advantage of the local concurrency controls. We propose strong session snapshot isolation, a correctness criterion that prevents transaction inversions. We show how strong session snapshot isolation can be implemented efficiently in a lazy replicated database system. Through performance studies, we quantify the cost of implementing our techniques in lazy replicated systems.

1. INTRODUCTION

Database systems that use lazy replication to provide system scalability have garnered the interest of commercial database vendors and academic research groups alike [24, 7, 19, 16, 14, 2]. One-copy serializability (1SR) has been the standard correctness criterion for replicated data. However, there has been significant interest in providing *snapshot isolation* (SI) [3], a transactional guarantee weaker than 1SR. For example, Oracle provides snapshot isolation as its strongest transactional guarantee [22].[1] SI is also available in Microsoft's recent release of SQL Server 2005 [6].

The aim of providing transactional guarantees weaker than 1SR, such as SI, is that the database system can achieve increased concurrency by relaxing the isolation requirements on transactions. Relaxed isolation requirements mean that concurrently executing transactions may see each others' effects indirectly through their effects on the database. In SQL, these effects are also called phenomena [20] or anomalies [3]. In many systems, these anomalies may be acceptable if better system performance is desired. This is the trade-off between transaction isolation and performance. Higher degrees of transaction isolation guarantee fewer anomalies but with larger performance penalties.

Snapshot isolation, proposed by Berenson et al [3], avoids many of the anomalies that are avoided by 1SR. However, SI does not guarantee serializability. SI extends the multiversion mixed method [4] to allow update transactions to read old data. A key benefit in systems that guarantee SI is that reads are never blocked. This decoupling of reads from updates results in increased concurrency and provides better system performance, particularly for read-only transactions.

Under SI, two concurrent update transactions have a write-write conflict if both transactions update at least one common data item. For example, consider a transaction T_1 that reads data items x and y and then updates x, concurrently, with another transaction T_2 that reads data items x and y and then updates y. T_1 and T_2 do not have a write-write conflict since neither transaction updates a common data item.

Now consider the same two transactions in a lazily synchronized replicated database system.[2] Since T_1 and T_2 do not have a write-write conflict under SI, their updates may be committed in the order T_1 followed by T_2 at a site s_1 but in the reverse order at another site s_2 in the replicated system in which each site individually guarantees SI. In this case, a transaction T_r that reads x and y at site s_1 and sees the database state resulting from the commit of T_1 will not see this same database state if it were to be executed on the database replica at site s_2. This replica inconsistency will not occur in a centralized database system that guarantees SI since any transaction that reads the database state resulting from the commit of T_1 will always see the same state.

To preserve the behavior of a centralized SI system in guaranteeing global SI in a distributed, lazily replicated, system, non-conflicting update transactions can be executed sequentially in the same order at all sites. Recall that the objective of the SI concurrency controls is to increase the level of concurrency, and therefore performance, by relaxing the isolation requirements on transactions [3]. Executing these transactions sequentially does not take advantage of the relaxed isolation level of the local SI concurrency controls. Proposals for maintaining global SI in replicated, distributed, systems exist but utilize actions outside of the local concurrency controls for enforcing an order on update transactions [8, 28, 24]. The first

[1] SI is called serializable isolation in Oracle.

[2] In a lazily synchronized system, replicas are updated using separate transactions.

VLDB '06, September 12-15, 2006, Seoul, Korea.

contribution of this paper is to show how global snapshot isolation, or global SI, can be guaranteed in lazily synchronized replicated database systems by taking full advantage of the local SI concurrency controls to order update transactions. Our approach captures the schedule of transaction start and commit operations enforced by the local concurrency control on update transactions at one site and uses this schedule to install these updates at other sites in the replicated system.

A key drawback of a lazily synchronized database system that guarantees global SI is that a client is not guaranteed to see its own updates. As an example, consider a customer (client) who submits two transactions sequentially to an online bookstore. The first transaction, T_{buy}, is an update transaction that purchases some number of books. The second transaction, T_{check}, is a read-only transaction that checks the status of the purchase. When the customer executes T_{check}, they may not see the status of their purchase. That is, T_{check} may run against a copy that does not yet reflect the update made by T_{buy}. This behavior, where the read-only transaction (T_{check}) sees a database state that does not include the effects of the update transaction (T_{buy}) despite the fact that it follows the update transaction in the request stream, results in a *transaction inversion*. Transaction inversions are possible because SI does not guarantee that a transaction will see a particular database state. Potentially any database state that was installed by a transaction that committed before T_{check} started, including one that was installed by a transaction that committed before T_{buy}, may be seen by T_{check} in a system that guarantees SI.

Most centralized systems, in practice, do not allow transaction inversions to occur because these systems actually guarantee that T_{check} will see the latest database state as of the time T_{check} was run. In this paper, we formally define this transactional guarantee, which we call *strong* SI. For example, Oracle and SQL Server guarantee strong SI [22, 6]. Given the popularity of lazy replication to scale-up a database server to support, for example, e-commerce and OLAP-based applications, it is desirable to prevent transaction inversions by guaranteeing strong SI not only in a centralized database system but also in a distributed, replicated, system. However, consider what would be involved to provide this guarantee. Using our previous example, T_{check} should not run until T_{buy}'s updates have been propagated and installed in the replicated system. This requirement effectively nullifies the advantages of lazy replication since it is equivalent to eager replication, where one would need to ensure that all replicas in the system are up-to-date before T_{check} can run. This requirement would have serious performance drawbacks on the lazy replicated system when scaled-up to a large number of replicas.

In this paper, we show that in a lazy replicated system, guaranteeing global strong SI is expensive, and that guaranteeing global SI is not enough to prevent transaction inversions. The second contribution of this paper is a new correctness criterion for replicated data called *strong session SI*. Strong session SI prevents transaction inversions within a client session but not across sessions. Thus, strong session SI provides many of the benefits of strong SI but without the performance penalties associated with guaranteeing strong SI.

We propose techniques that ensure global SI to transactions executing in a lazy replicated database system while preventing transaction inversions. To this end, we proceed in two steps. We first show how global SI can be guaranteed in a replicated system that is synchronized lazily. Our techniques leverage the local concurrency controls in the replicated system that guarantee SI to order update transactions. We then show how strong session SI can be maintained on top of this system. Our performance studies show that strong session SI can be provided at almost the cost of SI and at a much lower cost than strong SI.

2. SNAPSHOT ISOLATION (SI) GUARANTEES

In this section, we first present the operational definition of SI [3]. We then define strong SI and use it to derive our proposed transactional guarantee of strong session SI.

2.1 SI and Strong SI

Snapshot isolation (SI) was proposed by Berenson et al [3]. Snapshot isolation, as defined in [3], does not allow dirty writes (phenomenon P0), dirty reads (P1), fuzzy or non-repeatable reads (P2), phantoms (P3), or lost updates (P4) (definitions of these phenomena appear in the Appendix). However, write skew (P5) is possible under SI, making SI weaker than serializability.

In the definition of SI [3], which informally discusses how SI can be enforced, the system assigns a transaction T a start timestamp, called $start(T)$, when the transaction starts. $start(T)$ determines the database state seen by T. The system can choose $start(T)$ to be any time less than or equal to the actual start time of T. Updates made by any transaction T' that commits after time $start(T)$ will not be visible to T. Updates made by any transaction T' that commits before $start(T)$ will be visible to T. SI also requires that each transaction T be able to see its own updates. Thus, if T updates a database object and then reads that object, it will see the updated version, even though the update occurred after $start(T)$.

When a transaction T is to commit, it is assigned a commit timestamp, $commit(T)$, where $commit(T)$ is more recent than any $start$ or $commit$ timestamp assigned to any transaction. T commits only if no other committed transaction T' with lifespan $[start(T'), commit(T')]$ that overlaps with T's lifespan of $[start(T), commit(T)]$ wrote data that T has also written. Otherwise, T is aborted so as to prevent lost updates. This technique for preventing lost updates is called the *first-committer-wins* (FCW) rule. If T successfully commits, then any transaction T' that is assigned $start(T') > commit(T)$ will see T's update.

SI does not prevent transaction inversions, i.e. SI does not guarantee that for every pair of transactions T_1 and T_2, if T_2 executes after T_1 then T_2 will see T_1's updates. This is because $start(T_2)$ can be less than the actual start time of T_2. In particular, it is possible to choose $start(T_2) < commit(T_1)$ even if T_2 starts after T_1 has finished. In this case, T_2 will see a database state that does not include the effects of T_1. To prevent transaction inversions, we need to guarantee a stronger property, which we call strong SI.

DEFINITION 2.1. **Strong SI:** *A transaction execution history H is strong SI iff it is SI, and if, for every pair of committed transactions T_i and T_j in H such that T_i's commit precedes the first operation of T_j, $start(T_j) > commit(T_i)$.*

In a strong SI schedule, a transaction T_2 that starts after a committed transaction T_1 is guaranteed to see a committed database state that includes the effects of T_1. The operational definition of SI specifies the assignment of commit timestamps to transactions such

that a transaction T's $commit(T)$ has to be larger than any existing $start$ or $commit$ timestamp issued by the system. Additionally, strong SI constrains the assignment of $start$ timestamps by demanding that $start(T_2)$ be larger than $commit(T_1)$ if T_1 commits before T_2 starts. Note that since the commit of each update transaction moves the database state forward, commit timestamps represent database state.

2.2 Terminology

To avoid confusion, we will use the term weak SI in this paper to refer to the notion of SI described in Section 2.1 and originally defined in [3]. Thus, weak SI allows a transaction to see any snapshot earlier than its start timestamp. As in Definition 2.1, we will continue to use strong SI to mean the notion of SI in which a transaction sees only the latest snapshot.

The term weak SI is equivalent to generalized snapshot isolation used in [8] while strong SI is equivalent to the term conventional snapshot isolation from [8] and to SI used elsewhere [32, 17, 6, 24, 28, 27, 9].

2.3 Strong Session SI

Strong SI may be too strong since it enforces transaction ordering and database state constraints between all pairs of transactions. These requirements may be costly to enforce. Ordering constraints may be necessary between some pairs of transactions but not between others. We use *sessions* as a means of specifying which ordering constraints are important and which are not. The transactions in an execution history H are partitioned into sessions. Ordering constraints can be enforced on transactions that belong to the same session but not across sessions.

Since there can be many client sessions in a distributed system, we use a session labeling [7] to assign transactions to sessions. If T is a transaction in history H, we use the notation $L_H(T)$ to refer to T's session label. Given an execution history H and a labeling L_H, we define our session-level correctness criterion as follows:

DEFINITION 2.2. **Strong Session SI:** *A transaction execution history H is strong session SI under labeling L_H iff it is weak SI, and if, for every pair of committed transactions T_i and T_j in H such that $L_H(T_i) = L_H(T_j)$ and T_i's commit precedes the first operation of T_j, $start(T_j) > commit(T_i)$.*

If each transaction is assigned the same session label then strong session SI is equivalent to strong SI since only one ordering matters. If a distinct label is assigned to every transaction, strong session SI is equivalent to weak SI since no ordering guarantees matter.

2.4 Ordering Writes

We now discuss the ordering of update transactions under weak SI. Two update transactions T_i and T_j are sequential if $commit(T_i) < start(T_j)$. In this case, T_i's update will be installed before T_j's update. Two update transactions are concurrent if $start(T_i) < commit(T_j)$ and $start(T_j) < commit(T_i)$. Let the sets of data items that a transaction T_i reads and writes be called rs_i and ws_i, respectively. If T_i and T_j are two concurrent update transactions, both transactions cannot update and commit sucessfully if there is a write conflict, i.e. $ws_i \cap ws_j \neq \emptyset$. A way to resolve write conflicts is to use the first-committer-wins (FCW) rule [12] described in Section 2.1. For concurrent transactions with conflicting writes, the FCW rule ensures that only the updates of the first concurrent transaction to commit are installed. For example, the Oracle commercial database system applies the FCW rule to prevent lost updates by keeping a history of recently committed transactions that have updated each row of data on a per block basis [22].

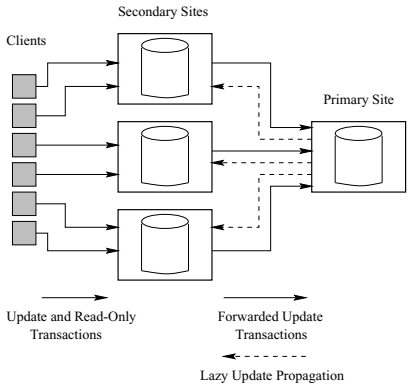

Figure 1: Lazy Master System Architecture

For two concurrent update transactions T_i and T_j, it is possible that their write sets are such that $ws_i \cap ws_j = \emptyset$. In addition, if $rs_i \cap ws_j \neq \emptyset$ and $rs_j \cap ws_i \neq \emptyset$, then P5 (write skew) is possible. In either case, T_i's updates can be ordered before the updates of T_j or vice versa under weak SI. We will consider the case of sequential and concurrent update transactions when we prove that updates in our lazily synchronized system are installed in a consistent fashion.

3. THE WEAK SI SYSTEM

In this section, we first describe a replicated database system with lazy update propagation that provides global weak SI. In the next section, we present an algorithm for enforcing strong session SI in this system.

Figure 1 illustrates the architecture of the weak SI system. A primary site holds the primary copy of the database, and one or more secondary sites hold secondary copies of the database. Each site consists of an autonomous database management system with a local concurrency controller that guarantees strong SI and is deadlock-free, which is an assumption that is true of strong SI concurrency controls in commercial systems.

We assume that the database is fully replicated at one or more secondary sites. Clients connect to one of the secondary sites and submit transactional requests. 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 and executed there.

We assume that when a transaction starts, the local concurrency control assigns it a $start(T)$ timestamp in response to the transaction's $start$ operation. We assume that a logical log containing update records is available. For example, Oracle provides the capability for such a log, which can hold SQL statements that are used to apply updates to a copy of the database [23]. We assume that

each update transaction's *start* timestamp is inserted into the log, followed by the transaction's update records, and then the transaction's commit record tagged with its *commit* timestamp or the abort record. We also assume that the start and commit timestamps are consistent with the actual order of start and commit operations at the site. One approach for generating timestamps is described in [18].

When a transaction T starts at the primary site, T sees the database state there as of some time $start(T)$, which corresponds to a committed database state represented by a unique commit timestamp. This means that when the updates of transaction T are installed using a separate *refresh* transaction R at a secondary site[3], R needs to see the database state seen by T at the primary site. If R does not see the same database state as T, and if R's writes depend on the database state it reads before it writes, R may never install a valid state on commit. In this case, the system would fail to guarantee global weak SI since the system may not be able to assign a start timestamp to a transaction that would allow it to see the same database state seen by T. Thus, to avoid this, if update transaction T' saw the primary database state resulting from T's commit (given by $commit(T)$) then T''s refresh transaction R' also needs to see this same state resulting from the commit of R at a secondary site. This means that the order in which refresh transactions execute, including the database states seen by each of the transactions, has to match that of their corresponding update transactions at the primary.

3.1 Synchronization Overview

We will use the notation $start_p(T)$ or $commit_p(T)$ to mean the start or commit timestamp, respectively, issued by the primary site to transaction T. When referring to the start or commit timestamps of transactions at a secondary, we shall use the subscript s instead of p with this timestamp notation. For example, $commit_s(R)$ means the commit timestamp issued to refresh transaction R's commit at a secondary site s. To ensure that update transactions and their corresponding refresh transactions start and commit in the same order at each site, the following relationships have to hold:

1. $start_p(T_2) > commit_p(T_1) \Rightarrow start_s(R_2) > commit_s(R_1)$
2. $commit_p(T_2) > start_p(T_1) \Rightarrow commit_s(R_2) > start_s(R_1)$
3. $commit_p(T_2) > commit_p(T_1) \Rightarrow commit_s(R_2) > commit_s(R_1)$

The database state seen and committed by update transactions at the primary correspond to the transactions' start and commit timestamps, respectively. The sorted order of these timestamps represents the sequence of database states seen and committed by transactions at the primary site. Thus, the system needs to ensure that the start and commit of all refresh transactions follow the sorted order of start and commit timestamps at the primary. To achieve this, transactions' start and commit records (with updates) are propagated in the order of their timestamps to secondary sites. At each secondary site, we ensure that refresh transactions' start and commit operations follow this propagation order. Next, we describe the propagation and refresh mechanisms.

3.2 Update Propagation

A simple method for propagating updates to secondary sites is to propagate start operations, updates and commit operations in primary timestamp order. These operations can then be executed in propagation order at each secondary site to ensure that relationships 1, 2 and 3 hold. As the workload scales-up, the potential for aborts increases with the number of update transactions. The drawback of the simple approach is that execution and the abort of updates would result in wasted work while consuming significant processing capacity at the secondary sites. Thus, it is important to avoid propagating and applying updates of transactions that would eventually have to be aborted.

Our approach for update propagation is summarized in Algorithm 3.1. Committed updates from the primary site's log are kept in an *update list*. Updates from this list are broadcast to all secondary sites where they are placed into a FIFO *update queue*. Propagation follows the order of start and commit timestamps and we assume that propagated messages are not lost or reordered during propagation. T's commit record, which consists of T's updates and its commit timestamp, is propagated only after T commits. This avoids propagation and execution of updates that may have to be aborted at the secondary sites. It is possible that updates of a committed transaction T are ready to be propagated but that there is a transaction T' with $start_p(T') < commit_p(T)$ that has yet to commit. If T's updates are not propagated until after T' commits, this can affect the progress of update propagation. To ensure propagation liveness, $start$ records are propagated when they are encountered in the primary site's log. Moreover, since updates are propagated in commit order, it is ensured that there is no committed but unpropagated transaction T' with $commit_p(T') < commit_p(T)$.

Algorithm 3.1 Primary Update Propagation

```
1  for each entry at the head of the log
2      do
3          if entry is start_p(T)
4              then propagate start_p(T)
5          if entry is T's update
6              then insert it in update list for T
7          if entry is commit_p(T)
8              then propagate T's update list with commit_p(T)
9          if entry is abort_p(T)
10             then propagate abort_p(T)
```

3.3 Secondary Refresh

We need to ensure that relationships 1, 2 and 3 are maintained when performing secondary refresh. A straightforward method of achieving this at the secondary sites is to apply the start, update and commit operations in primary log sequence. However, it would be advantageous to exploit the potential of the concurrency control to apply updates concurrently while maintaining relationships 1, 2 and 3. Our technique for secondary refresh utilizes this approach.

A refresh process, or refresher, runs at each secondary site. The refresh process dequeues updates from the local update queue and applies them to the database copy using refresh transactions. For each propagated update transaction, the refresher forks an *applicator* thread that applies the transaction's updates as a separate refresh transaction to the local database.

Pseudocode for a single iteration of the refresher is shown in Algorithm 3.2. The refresher communicates with the applicator threads through a FIFO queue called the *pending* queue. Pseudocode for a

[3]Recall from Section 1 that database replicas are updated using separate transactions, which we call refresh transactions.

Algorithm 3.2 Secondary Refresh Algorithm

```
1  if head(updateQueue) = start_p(T)
2     then while (pendingQueue is not empty)
3               do {} /* block */
4          start refresh(T)
           /* start new local transaction to apply T's updates */
5  elseif head(updateQueue) is T's commit record
6     then append(commit_p(T), pendingQueue)
7          fork(RunTransaction(T's updates, refresh(T)))
           /* runs T's updates within refresh(T) */
           /* as described in Algorithm 3.3 */
8  elseif head(updateQueue) = abort_p(T)
9     then abort refresh(T)
10 delete (head(updateQueue)) /* delete processed entry */
```

single iteration of an applicator thread is shown in Algorithm 3.3. When the refresher dequeues the start record for transaction T from the update queue, it blocks until the pending queue is empty before it starts T's local refresh transaction R. The blocking of T's start record ensures that T sees a database state that includes the updates of the transaction that committed before it. Algorithm 3.2 also ensures that all updates of a transaction T are applied within a single refresh transaction R. A single applicator thread is used to apply all updates of T (using Algorithm 3.3), and multiple, concurrent, applicator threads may be active at any one time.

In practice, instead of creating a new thread each time as shown in Algorithm 3.2, a fixed-size pool of applicator threads could be made available at each secondary. Algorithm 3.2 shows a thread creation (fork) for each transaction just to keep the code simple.

Algorithm 3.3 Applicator Thread

```
RunTransaction(updateList, R)
1  for each update in updateList
2       do execute update within refresh transaction R
3  while (head(pendingQueue) is not commit_p(T))
4       do {} /* block */
5  commit(R)
6  delete(commit_p(T), pendingQueue)
```

3.4 Concurrency and Failure

The refresh mechanism described in the previous section enqueues and dequeues records to and from the update queue and the pending queue. Since both enqueue and dequeue include a write operation, a potential problem is that concurrent transactions operating on the queues may have a write-write conflict if they update data items on the same page (or block) of the queue [9]. Under the FCW rule, only one of the concurrent transactions would be allowed to complete successfully while the rest would be aborted. This can undermine the progress of the refresh mechanism and can reduce it to a sequential process. To avoid aborts, propagated records can be stored in a queue outside of the database.

Keeping propagated records in a queue outside a secondary database introduces a potential point of failure. If a secondary site fails, the queued updates and the refresh state would be lost. For two refresh transactions R_1 and R_2 that are concurrent, R_1 may have seen a database state installed before R_2's commit. If R_2 committed at the secondary before the failure, R_1 will not see a database state that precludes R_2's commit if it is restarted. A solution to this problem is to periodically create a copy of the primary database after quiescing it. In the case of failure of a secondary site, this copy can be installed at the secondary and any updates that have committed at the primary between the quiesced state and installation of the database copy can be propagated and installed at the secondary using the refresh mechanism described in the previous section.

3.5 Correctness

We will now establish that relationships 1, 2 and 3 (from Section 3.1) hold for the weak SI system we have described.

LEMMA 3.1. *For any primary transactions T_1 and T_2 and corresponding refresh transactions R_1 and R_2, respectively, $start_p(T_1) < commit_p(T_2) \Rightarrow start_s(R_1) < commit_s(R_2)$.*

Proof: Since $start_p(T_1) < commit_p(T_2)$, the propagator will send T_1's start record before T_2's commit record and they will be received in this order in the local update queue. The refresher will dequeue T_1's start record and start T_1's refresh transaction R_1 before it applies T_2's updates using refresh transaction R_2 and submits $commit_s(T_2)$. This means that $start_s(R_1) < commit_s(R_2)$. □

LEMMA 3.2. *For any primary transactions T_1 and T_2 and corresponding refresh transactions R_1 and R_2, respectively, $commit_p(T_1) < start_p(T_2) \Rightarrow commit_s(R_1) < start_s(R_2)$.*

Proof: Because update propagation is in timestamped order, T_1's commit record will be at the head of the update queue before T_2's start record. This means that the refresh thread will have appended T_1's commit record to the pending queue before it processes T_2's start record. Since the refresher will not process T_2's start record until the pending queue is empty, and since the applicator thread for T_1's refresh transaction R_1 does not remove T_1's commit record from the pending queue until it has commited R_1, the refresher will not start T_2's refresh transaction R_2 until after T_1's refresh transaction R_1 commits. This means that $commit_s(R_1) < start_s(R_2)$. □

LEMMA 3.3. *For any primary transactions T_1 and T_2 and corresponding refresh transactions R_1 and R_2, respectively, $commit_p(T_1) < commit_p(T_2) \Rightarrow commit_s(R_1) < commit_s(R_2)$.*

Proof: Because update propagation is in timestamped order, T_1's commit record will be at the head of the update queue before T_2's commit record. This means that the refresher will insert $commit_p(T_1)$ in the pending queue before inserting $commit_p(T_2)$ in the pending queue. Since $commit_p(T_1)$ will precede $commit_p(T_2)$ in the pending queue, and since the applicator thread will not commit T_2's refresh transaction R_2 until $commit_p(T_2)$ is at the head of the pending queue, the commit of T_1's refresh transaction R_1 will precede R_2's commit and thus, $commit_s(R_1) < commit_s(R_2)$. □

Let S_p^i represent the ith database state at the primary site. This is the database state that is created by the ith update transaction to commit at the primary. Similarly, for any particular secondary site, let S_s^i represent the ith database state at that site. This is the database state created by the ith refresh transaction at the secondary site. The update propagation and secondary refresh mechanisms of the weak SI system ensure that the secondary's database state tracks the state of the primary, although it may lag behind. This property, which was called *completeness* by Zhuge, Garcia-Molina et al [33, 34], is established for the weak SI system by Theorem 3.1.

THEOREM 3.1. *For every secondary site, if $S_p^0 = S_s^0$ then $S_p^i = S_s^i$, for all $i \geq 0$.*

Proof: By induction on i. The base case ($i = 0$) is established by assumption. Suppose that the Theorem is true for all $i < k$. Let T_k represent the kth update transaction to commit at the primary site. T_k produces the state S_p^k at the primary. From Lemma 3.3, refresh transactions commit at each secondary site in the same order as the corresponding update transactions at the primary. Thus, R_k, the refresh transaction corresponding to T_k, will be the kth refresh transaction to commit at the secondary site, producing state S_s^k. Let T_j represent the last update transaction to commit at the primary site prior to $start_p(T_k)$, i.e., $commit_p(T_j) < start_p(T_k) < commit_p(T_{j+1})$. Since the primary site guarantees strong SI locally, T_k sees the database state S_p^j, the state produced by T_j. From Lemmas 3.1 and 3.2, $commit_s(R_j) < start_s(R_k) < commit_s(R_{j+1})$, and from Lemma 3.3, there are no other refresh transactions that commit between R_j and R_{j+1}. Since the secondary site guarantees strong SI locally, refresh transaction R_k sees the state S_s^j produced by R_j. By the inductive hypothesis, $S_s^j = S_p^j$, so R_k sees the same database state that T_k did at the primary site. Since R_k consists of the same updates as T_k and it applies those updates to the same state that T_k did, it will produce the same state at the secondary that T_k did at the primary, provided that it commits. Since the local concurrency control is deadlock-free, R_k will commit, resulting in $S_s^k = S_p^k$. □

THEOREM 3.2. *The weak SI system guarantees global weak SI.*

Proof: All update transactions are executed at the primary site, which guarantees strong SI locally. (Strong SI implies weak SI.) Consider an arbitrary read-only transaction T_r, which is executed at some secondary site. Let R_i represent the refresh transaction that commits immediately prior to $start_s(T_r)$, and let T_i represent the corresponding update transaction at the primary site. Since the local concurrency control at the secondary site guarantees strong SI, T_r sees the database snapshot S_s^i that is produced by R_i. From Theorem 3.1, S_s^i is the same as S_p^i, the primary database state produced by T_i. Thus, running T_r at the secondary is equivalent to running it in the strong SI schedule at the primary site immediately after T_i commits. A similar argument can be made for every read-only transaction running at any secondary site. Thus, the global transaction schedule is weak SI. □

Note that, although each local system guarantees strong SI locally, the global guarantee is weak SI, not strong SI. Theorem 3.2 shows that each read-only transaction sees a transaction-consistent snapshot. However, the snapshot that a read-only transaction sees at a secondary site may be stale with respect to the current state at the primary. As a result, transaction inversions may occur. In particular, a read-only transaction that follows an update transaction may see a database state that does not yet include the update transaction's effects.

4. ENFORCING GLOBAL STRONG SESSION SI

A client's sequence of transactional requests constitute a session. We assume that each transaction has associated with it, either explicitly or implicitly, a session label so that the secondary site can tell which transactions belong to which session. In a web services environment, 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 propose ALG-STRONG-SESSION-SI, which is used to enforce global strong session SI. ALG-STRONG-SESSION-SI uses timestamps that correspond to the commit order of transactions at the primary database. These timestamps are used to control the order in which read-only transactions are executed.

The ALG-STRONG-SESSION-SI algorithm maintains a session sequence number $seq(c)$ for each session c. When an update transaction T from session c commits, $seq(c)$ is set to $commit_p(T)$. A sequence number $seq(DB_{sec})$, maintained by refresh transactions, is used to represent the state of the secondary database in terms of the primary database. When update transaction T's refresh transaction R commits at the secondary site, the applicator thread sets the value of $seq(DB_{sec})$ to $commit_p(T)$. When a read-only transaction T_r starts, T_r will wait if $seq(c) > seq(DB_{sec})$. Otherwise, since each site guarantees strong SI, T_r will see a database state that is at least as recent as the database state seen by the last transaction from the same session. The refresher uses exactly the same code as that shown in Algorithm 3.2. The applicator threads also use the same code as shown in Algorithm 3.3 with the additional step of setting $seq(DB_{sec})$ to $commit_p(T)$ immediately after refresh transaction R commits at the secondary site and before deleting $commit_p(T)$ from the pending queue.

In case of failure at a secondary, a dummy transaction T_d can be executed at the primary site after recovery to obtain the sequence number associated with the primary's latest committed database state. $seq(DB_{sec})$ can then be reinitialized to the sequence number associated with T_d.

THEOREM 4.1. *If each site guarantees strong SI locally, then the* ALG-STRONG-SESSION-SI *algorithm in conjunction with the propagation and refresh mechanisms of the weak SI system guarantees global strong session SI.*

Proof: Suppose that the claim is false, which means that there exists a pair of transactions T_1 and T_2 in the same session c for which T_1 is executed before T_2 but $start(T_2) < commit(T_1)$, i.e. T_2 "sees" a database state that does not include the effects of T_1. There are four cases to consider:

Case 1: Suppose T_1 and T_2 are update transactions. T_1 and T_2 both execute at the primary site. Since the primary site ensures strong SI and since T_2 starts after T_1 finishes, $start_p(T_2) > commit_p(T_1)$, a contradiction.

Case 2: Suppose T_1 is a read-only transaction and T_2 is an update transaction. Since T_1 precedes T_2 and T_2 precedes its refresh

transaction R_2, T_1 precedes R_2 at the secondary site. Since the secondary site ensures strong SI, $start_s(R_2) > commit_s(T_1)$. Since the system provides completness (Theorem 3.1) and global weak SI (Theorem 3.2), $start_s(R_2) > commit_s(T_1)$ is equivalent to $start_p(T_2) > commit_p(T_1)$, a contradiction.

Case 3: Suppose T_1 is an update transaction and T_2 is a read-only transaction. After T_1 commits, $seq(c)$ is set to $commit_p(T_1)$. The blocking condition ensures that no subsequent read-only transaction in session c can run until $seq(DB_{sec})$ is at least as large as $commit_p(T_1)$. Thus, T_2 runs after $seq(DB_{sec})$ is set to $commit_p(T_1)$, which happens after the commit of T_1's refresh R_1. Since the secondary site ensures strong SI, $start_s(T_2) > commit_s(R_1)$. Since the system provides completness (Theorem 3.1) and global weak SI (Theorem 3.2), $start_s(T_2) > commit_s(R_1)$ is equivalent to $start_p(T_2) > commit_p(T_1)$, a contradiction.

Case 4: Suppose both T_1 and T_2 are read-only transactions. Both transactions run at the secondary site. Since strong SI is guaranteed locally there and T_2 starts after T_1 commits, $start_s(T_2) > commit_s(T_1)$, a contradiction.□

5. SIMULATION MODEL

We have developed a simulation model of the weak SI system described in Section 3. The simulator has been used to study the cost of providing the strong session SI guarantee described in Section 4, and for comparing it to the cost of maintaining the strong SI and weak SI guarantees. The simulation model is implemented in C++ using the CSIM simulation package [21].

Each site (or server) is modelled as a CSIM resource. Client processes simulate the execution of transactions in the system. Each client process is associated with a single secondary site, and submits all its transactions to that site. Client processes are distributed uniformly over the secondary sites in the system. Other processes in the system are the (update) propagator and refresher. A summary of the simulation model's parameters appears in Table 1.

Each client process initiates a series of sessions, each of which consists of a sequential stream of transactions. Session lengths are exponentially distributed with a mean session length of *session_time*. Each client process thinks between transactions, where the think times follow an exponential distribution with a mean think time of *think_time*. The total number of sessions in the system at any one time is *num_clients*. When a client session ends, a new one is started immediately. Our *session_time* and *think_time* mean values are taken from the TPC-W benchmark [31].

A transaction is an update transaction with probability *update_tran_prob*, and a read-only transaction with probability *(1 – update_tran_prob)*. The default mix of read-only/update transactions that we use for our workload is 80%/20%. We also ran some experiments with the 95%/5% mix. The 80%/20% mix follows the "shopping" mix in the TPC-W specification while the 95%/5% mix is the "browsing" mix. The TPC-W benchmark specifies web interactions rather than transactions. If each web interaction is a transaction, which the benchmark allows, then the read-only to update mixes are of the same proportions as in our workload mix.

Each transaction is directed to a secondary site or to the primary site for execution. All update transactions execute at the primary site while read-only transactions execute at secondary sites. Each read-only transaction is subjected to its execution site's local concurrency control while each update transaction is subjected to primary site's concurrency control.

Parameter	Description	Default
num_sec	number of secondary sites	varies
num_clients	number of clients	20/secondary
think_time	mean client think time	7s
session_time	mean session duration	15 min.
update_tran_prob	probability of an update transaction	20%
abort_prob	update transaction abort probability	1%
tran_size	mean number of operations per transaction	10
op_service_time	service time per operation	0.02s
update_op_prob	probability of an update operation	30%
propagation_delay	propagator think time	10s

Table 1: Simulation Model Parameters

We use a simple model of a concurrency controller at the primary site that implements local strong SI and the first-committer-wins rule. When an update transaction is ready to commit, it has probability *abort_prob* of aborting. We set the value of *abort_prob* to 1%. On abort, the transaction is restarted so as to maintain the load at the primary. Note that the aim of our simulator is to quantify the cost of providing strong session SI and to measure system scalability. Thus, we are not concerned with simulating a detailed model for transaction aborts. As described in Sections 3.3 and 3.5, start, update and commit operations at the primary are applied at each secondary site while maintaining relationships 1, 2 and 3. Thus, explicit local concurrency controls are not modeled at secondary sites. Read-only transactions are never blocked at the secondary sites since we assume that they access committed snapshots of data and do not contend with refresh transactions.

The number of operations in a transaction is randomly chosen from 5 to 15, with a mean of *tran_size* operations. If the transaction is an update transaction, each of its operations is an update operation with probability *update_op_prob*, otherwise it is a read operation. All transactions running at a site access the server at that site. The server is a shared resource with a round-robin queueing scheme having a time slice of 0.001 seconds.

The propagator at the primary site executes the propagation mechanism described in Section 3.2. During each cycle, the propagation process sends all start and commit records to all secondaries that have accumulated since the last propagation cycle. Since we assume that the propagator is implemented as a log sniffer, it does not use the local concurrency control. A resource to represent the network is not used in the simulator. We assume that the network has sufficient capacity so that network contention is not a significant contributor to the propagation delay. We use a 10s propagation delay to account for delays that may result from network latencies, batching and scheduling at primary sites.

There is one refresher process and multiple applicator threads at each secondary site. The refresher dequeues messages containing start and commit information from the local update queue and processes them according to the algorithm described in Section 3.3. The refresher communicates with the applicator threads by sending messages containing the updates that need to be installed. Com-

mit timestamp messages are inserted into the FIFO pending queue by the refresher. A notification process monitors the state of the pending queue and notifies the refresher when the pending queue is empty. This process also notifies the applicator thread that installs refresh transaction R of update transaction T when $commit_p(T)$ is at the head of the pending queue. This notification allows the applicator thread to commit refresh transaction R of update transaction T. The thread then sets $seq(DB_{sec})$ to $commit_p(T)$ after which it dequeues $commit_p(T)$ from the pending queue.

6. PERFORMANCE ANALYSIS

We conducted experiments using our simulation model to study the efficacy of the ALG-STRONG-SESSION-SI algorithm with respect to transaction throughput and response time. We also wanted to compare the cost of providing strong session SI against that of providing global strong SI and global weak SI. To make this comparison, we implemented two more algorithms against which we compare ALG-STRONG-SESSION-SI:

ALG-WEAK-SI: provides global weak snapshot isolation by simply forwarding all update transactions to the primary for execution while executing read-only transactions at the secondary site. ALG-WEAK-SI, itself, never blocks transactions. Transactions are subjected to the local concurrency control of the site at which they execute. Under ALG-WEAK-SI, the functionality of the system is equivalent to that of the weak SI system described in Section 3.

ALG-STRONG-SI: this algorithm is the same as the ALG-STRONG-SESSION-SI algorithm described in Section 4, except that instead of having one session per client, there is a single session for the system. Only a single session sequence number $seq(c)$ is maintained for the whole system. Since having a single $seq(c)$ implies that only a single global ordering of transactions is enforced, we expect that ALG-STRONG-SI will perform worse than ALG-WEAK-SI and ALG-STRONG-SESSION-SI.

6.1 Methodology

For each run, the simulation parameters were set to the values shown in Table 1, except as indicated in the descriptions of the individual experiments. Each run lasted for 35 simulated minutes. We ignored the first five minutes of each run to allow the system to warm up, and measured transaction throughput, response times and other statistics over the remainder of the run. Since we truncate the simulation at the end of 35 minutes, a cool down phase is not required. Each reported measurement is an average over five independent runs. We computed 95% confidence intervals around these means. These are shown as error bars in the graphs.

6.2 Performance Results

We subjected our configuration of five secondary sites to load from an increasing number of clients. The results of these experiments are shown as throughput, read-only and update transaction response time curves in Figures 2, 3 and 4. To show how many transactions finish within a short time in the system, the throughput curves are response time-related; they show the number of transactions that finish in 3s or less.

The throughput and response time curves (Figures 2 and 3 respectively) show that the ALG-STRONG-SESSION-SI algorithm performs almost as well as ALG-WEAK-SI, and significantly better than ALG-STRONG-SI. The throughput of the ALG-STRONG-SESSION-SI algorithm is almost identical to that of ALG-WEAK-SI at low load. Under moderate to heavy load, the difference in throughput increases and ALG-STRONG-SESSION-SI incurs a small penalty compared to ALG-WEAK-SI. Under ALG-WEAK-SI, since no session constraints are enforced, transactions are free to run as soon as they are submitted for execution. Since session constraints are enforced under ALG-STRONG-SESSION-SI, some transactions within a session are forced to wait before they are serviced, resulting in a small response time penalty (Fig. 3). The performance of ALG-STRONG-SI suffers significantly compared to ALG-STRONG-SESSION-SI since enforcing a total order on transactions is expensive, as shown by Figures 2 and 3. Figure 4 shows small update response times for ALG-STRONG-SI because it forces transactions to wait a long time while enforcing a total order (single session) constraint. This results in a low offered update load, leading to low update response times. Unlike ALG-STRONG-SI, ALG-STRONG-SI and ALG-STRONG-SESSION-SI are not subjected to a total order constraint and are able to offer a higher update load, leading to higher update response times.

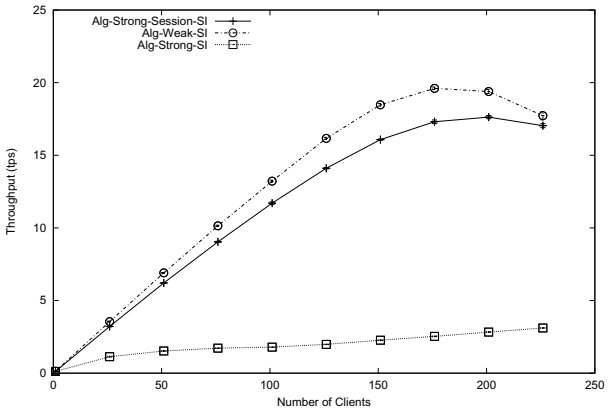

Figure 2: Transaction Throughput vs. Number of Clients, 80/20 workload

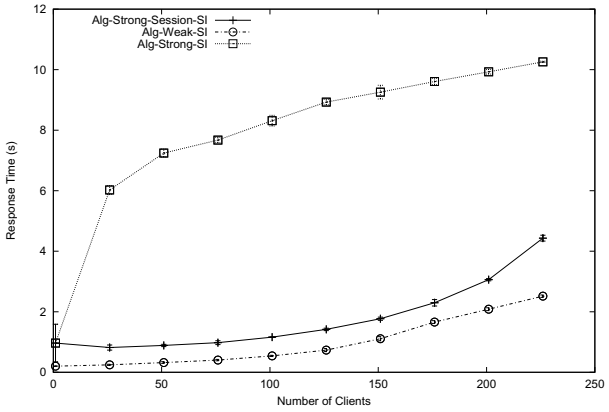

Figure 3: Read-Only Transaction Response Time vs. Number of Clients, 80/20 workload

6.2.1 Scalability

To test system scalability, we ran experiments where the number of secondary sites were scaled with the client load. We measured (response time-related) throughput and response time as both the number of secondary sites and clients were increased gradually. Figures 5, 6 and 7 show the results of this experiment.

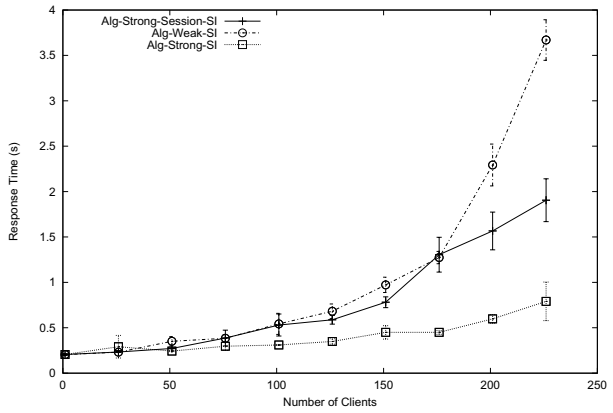

Figure 4: Update Transaction Response Time vs. Number of Clients, 80/20 workload

The ALG-STRONG-SESSION-SI algorithm again showed scale-up behaviour comparable to that of ALG-WEAK-SI. ALG-STRONG-SI, on the other hand, performed poorly compared to the other two algorithms. In the case of ALG-STRONG-SESSION-SI and ALG-WEAK-SI, as the workload scales-up, the primary site becomes saturated with an increasing update load and the update response time then increases rapidly (Fig. 7). Past 11 secondary sites, system throughput peaks as the saturated primary site limits system scalability.

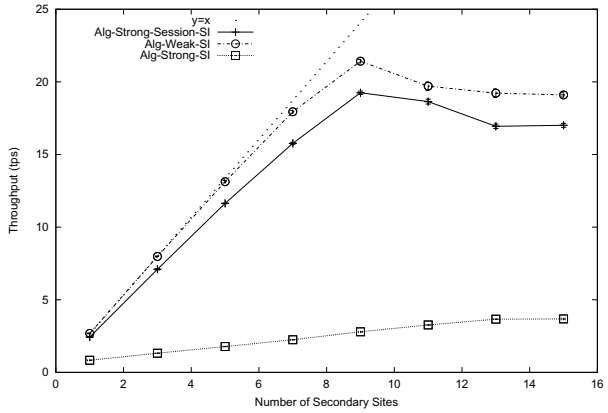

Figure 5: Transaction Throughput, 20 Clients per Secondary, 80/20 workload

The scalability results are sensitive to the mix of read-only and update transactions in the workload. As the results of Figure 8 show, significantly greater scalability is achieved with the 95/5 workload.

7. RELATED WORK

One-copy serializability (1SR) has been the standard correctness criterion for replicated data [4, 14, 2, 1, 5]. Recently, there has been growing interest in providing timetravel functionality (in the context of weak SI) [18, 25], and strong SI [15, 24, 8]. There have been several other proposals for relaxing the one-copy serializability (1SR) requirement for replicated data [11, 26, 10]. However, these proposals do not consider an isolation level such as strong snapshot isolation that is available in commercial database systems.

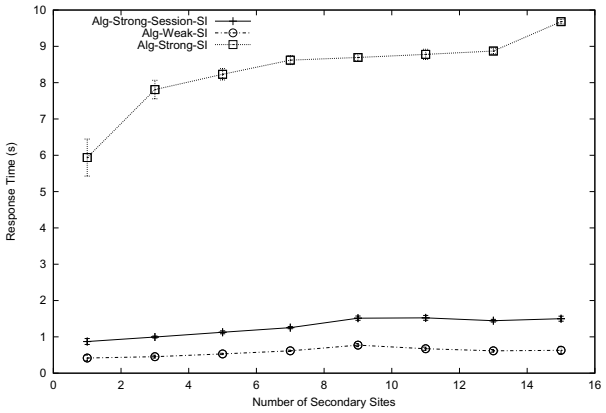

Figure 6: Read-Only Transaction Response Time, 20 Clients per Secondary, 80/20 workload

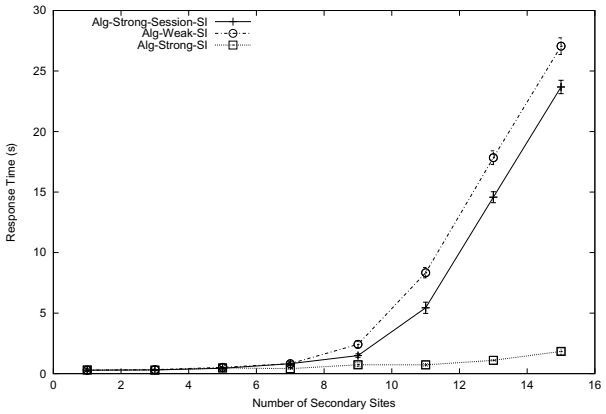

Figure 7: Update Transaction Response Time, 20 Clients per Secondary, 80/20 workload

Kemme and Alonso [15] proposed an eager replication protocol that is equivalent to strong SI. Their approach relies on group communication systems available for local area network clusters to determine a global serialization order for transactions. Wu and Kemme presented an eager replication scheme equivalent to strong SI that is integrated into a database system [32]. The writeset of an update transaction that is submitted for execution locally is multicast to all remote sites using group communication before the transaction commits. To ensure that concurrent update transactions do not execute in different orders at different sites, their approach imposes the restriction that writesets at all sites have to be executed serially in the same order. In contrast to these eager approaches, we focus on using lazy replication to provide global weak SI by exploiting the ordering of update transactions by the local concurrency controls at the primary site and show how session guarantees can be provided to prevent transaction inversions.

Lin et al [17] presented a middleware-based eager replication scheme that is equivalent to the provision of global strong SI. An update transaction executes at a single site, and its writes are applied at other replicas before it commits. This requires that the writeset be extracted from the local database system before commit. Their approach uses group communication to broadcast an update transaction's writeset to all replicated sites. The middleware

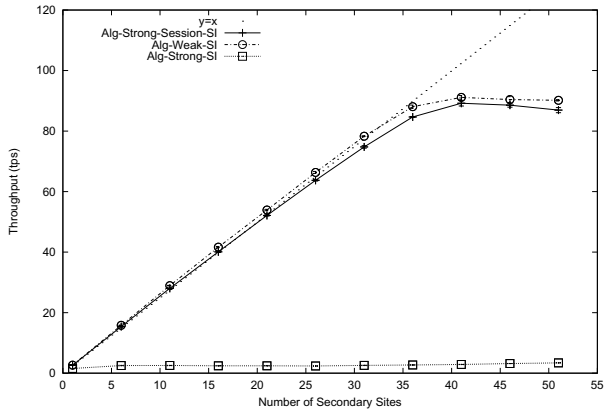

Figure 8: Transaction Throughput, 20 Clients per Secondary, 95/5 workload

determines write-write conflicts between transactions. However, the middleware may not be aware of a write conflict between a remote transaction and a transaction that is local until at commit time. Since this can result in deadlock, they propose to resolve it by allowing non-conflicting transactions whose writesets do not overlap to be installed in an order that is different from that enforced by the middleware. In the terminology of Zhuge et al [33, 34], their system does not provide completeness. In our work, we propose definitions of strong session SI and strong SI. We show that in a lazy system, maintaining global strong SI is expensive and maintaining global weak SI, while being less costly, does not provide any ordering guarantees. To this end, we propose global strong session SI and we show how to maintain it on top of global weak SI in replicated systems. We treat the snapshot replicas at secondary sites as views of the base data at the primary. We maintain the snapshot relations through lazy update propagation. We ensure that refresh transactions at every secondary site start and commit in the same order as their corresponding update transactions at the primary site. This provides completeness [33, 34] in the system.

Plattner and Alonso have recently proposed techniques for scaling-up a database system while guaranteeing equivalence to global strong SI [24]. Their single primary site architecture is similar to that described in this paper. All update transactions execute at the primary and read-only transactions execute at the secondary sites. The middleware allows use of the local concurrency control at the primary site at which update transactions execute but the middleware controls the order in which update transactions commit. Updates committed at the primary are propagated and installed at the secondary sites in commit order. Read-only transactions see the latest database state on execution. There is no notion of session-level transactional guarantees in their work and they do not exploit the relationship between transaction ordering and execution that we have exploited.

Elnikety et al [8] considered the provision of transactional guarantees weaker than 1SR in replicated databases. Their work proposes prefix-consistent SI (PCSI), which allows transactions in a workflow (a concept similar to our proposed notion of a session) to see a database state that includes the effects of previous update transactions within the workflow. Unlike strong session SI, PCSI does not enforce any ordering constraints between read-only transactions within the same workflow. For example, if a workflow contains two read-only transactions T_1 and T_2, where T_1 is followed by T_2, strong session SI requires that T_2's snapshot be at least as recent as T_1's snapshot. However, PCSI has no such requirement according to the PCSI rules in [8]. The techniques proposed in [8] allow update transactions to execute at any site provided that they synchronize with a master site, which orders the update transactions, using a distributed commit protocol. A second model uses a decentralized approach in which update transactions rely on an atomic broadcast protocol to enforce a total order on all updates. In our work, update transactions execute at the primary site. We utilize the primary's strong SI local concurrency controls to order all update transactions. Our approach captures the schedule of transaction start and commit operations enforced by the primary's local concurrency control on update transactions and uses this schedule to install these updates lazily at other sites in the replicated system.

Schenkel et al [28] propose algorithms for guaranteeing global strong SI in federated database systems where each site guarantees strong SI locally. When a transaction accesses data at different sites, to guarantee global strong SI the transaction has to see data at the same timepoint, i.e. the same database state at each site. A global transaction T_2 may execute concurrently with another transaction T_1 at one site but serially at another site. This may violate strong SI, since it is possible that one transaction may read a database state that is different from the other site. To ensure global strong SI, they describe an approach to control the execution order of transactions at each site to enforce a total global order. This approach requires the prediction of data that each transaction needs to access. They also consider an optimistic approach whereby transactions are run but aborted later if they violate global strong SI. This requires the determination of whether transactions execute serially or concurrently and restricts the set of allowable schedules under global strong SI.

Fekete et al [9] develop principles under which non-serializable execution of transactions can be avoided to guarantee serializability when the database system guarantees strong SI. They suggest modifying the workload statically, for example, by introducing conflicts between transactions such that they would effectively be serialized under the first-committer-wins rule. Schenkel et al [27] consider how 1SR can be guaranteed to transactions that execute on a database system that guarantees strong SI. They force update transactions to execute serially, and use tickets to represent their serialization order. The execution order of read-only transactions can then be controlled using these tickets. Optionally, they propose a technique that requires analysis of transaction conflicts and application semantics. These approaches are in contrast to our work, which considers the provision of session-level SI guarantees and not 1SR in lazy replicated systems.

Bayou [30, 29] is a fully replicated system that provides data freshness to user sessions through causal ordering constraints on read and write operations. Writes that are submitted for execution in Bayou must contain information about how to resolve conflicts with other writes. This information is specified in the form of a merge procedure, which is responsible for resolving conflicts detected during the application of a write. Each update is serviced by a single primary site, which assigns the update with a timestamp. Updates and their associated timestamps are propagated lazily between sites. A site is not guaranteed to know the total order of updates at any one time. Thus, Bayou guarantees only eventual consistency, which means that servers converge to the same database state only in the absence of updates. For replicated data, this is the

same weak notion of consistency as convergence [33, 34]. Bayou does not necessarily guarantee that constraints within a session are preserved. It is possible for the system to report that a requested session guarantee, e.g. read-your-writes, cannot be provided. Some recent work allows queries to see data up-to-date to different time-points [13] but this work does not consider data freshness in the context of transactional isolation levels. [7] shows how ordering guarantees under serializability can be provided in lazy replicated systems. However, these techniques are not generally applicable to provide the snapshot isolation transactional guarantees considered in this paper.

8. CONCLUSION

In this paper, we described an architecture and algorithms that take advantage of the local concurrency controls to maintain global weak SI in lazy replicated systems. We defined a new session-oriented correctness criterion called strong session SI and we showed how it can be implemented efficiently to prevent transaction inversions in systems where each local concurrency control guarantees strong snapshot isolation. We showed that our proposed correctness criterion of strong session SI can perform almost as well as algorithms that maintain global weak SI, but without the high cost of providing strong SI. We conclude that our techniques for guaranteeing global weak SI and for avoiding transaction inversions through strong session SI in lazy replicated database systems are promising.

9. REFERENCES

[1] Divyakant Agrawal, Amr El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Symposium on Principles of Database Systems*, pages 161–172, 1997.

[2] F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and Lourens Veen. Fine-grained lazy replication with strict freshness and correctness guarantees. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 565–576, 2005.

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

[6] Microsoft Corp. SQL Server 2005 Beta 2 Snapshot Isolation. www.microsoft.com/technet/prodtechnol/sql/2005/SQL05B.mspx, 2005.

[7] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 424–435, 2004.

[8] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Database replication using generalized snapshot isolation. In *Proc. Symposium on Reliable Distributed Systems*, pages 73–84, 2005.

[9] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[10] Rainer Gallersdörfer and Matthias Nicola. Improving performance in replicated databases through relaxed coherency. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 445–456, 1995.

[11] Hector Garcia-Molina and Gio Wiederhold. Read-Only Transactions in a Distributed Database. *ACM Transactions on Database Systems*, 7(2):209–234, 1982.

[12] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[13] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed currency and consistency: How to say "good enough" in SQL. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 815–826, 2004.

[14] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 156–163, 1998.

[15] Bettina Kemme and Gustavo Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[16] Per-Ake Larson, Jonathan Goldstein, and Jingren Zhou. MTCache: Mid-Tier Database Caching in SQL Server. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 177–188, 2004.

[17] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2005.

[18] David Lomet, Roger Barga, Mohamed Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Transaction Time Support Inside a Database Engine. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, 2006.

[19] 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 *Proc. ACM SIGMOD International Conference on Management of Data*, pages 600–611, 2002.

[20] Jim Melton and Alan Simon. *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[21] Mesquite Software Inc. *CSIM18 Simulation Engine (C++ version) User's Guide*, January 2002.

[22] Oracle Corporation. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*, 1995. Whitepaper.

[23] Oracle Corporation. *Oracle8. Data Guard Concepts and Administration*, 2003.

[24] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proc. Middleware*, pages 155–174, 2004.

[25] Christian Plattner, Andreas Wapf, and Gustavo Alonso. Searching in time. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 754–756, 2006.

[26] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 377–386, 1991.

[27] Ralf Schenkel and Gerhard Weikum. Integrating snapshot isolation into transactional federation. In *Proc. CoopIS*, pages 90–101, 2000.

[28] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, and Xuequn Wu. Federated transaction management with snapshot isolation. In *Eight International Workshop on Foundations of Models and Languages for Data and Objects*, pages 1–25, 1999.

[29] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, and Marvin Theimer. Flexible update propagation for weakly consistent replication. In *Symposium on Operating Systems Principles*, pages 288–301, 1997.

[30] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Conference on Parallel and Distributed Information Systems*, pages 140–149, 1994.

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

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

[33] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 316–327, 1995.

[34] 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.

APPENDIX

A. SQL PHENOMENA

Four phenomena are possible under the ANSI SQL isolation levels [3]:

DEFINITION A.1. *A* **dirty write (P0)** *occurs in a transaction execution history when some transaction T_1 modifies a data item and some other transaction T_2 then modifies the same data item before T_1 commits or aborts.*

The occurrence of P0 can lead to a problem if T_1 or T_2 later abort since it is not clear what the correct value of the data item should be.

DEFINITION A.2. *A* **dirty read (P1)** *occurs in a transaction execution history when some transaction T_1 modifies a data item and then some other transaction T_2 reads that data item before T_1 commits or aborts.*

If T_1 aborts, T_2 has read a data item that was never committed, resulting in an inconsistency.

DEFINITION A.3. *A* **fuzzy or non-repeatable read (P2)** *occurs in a transaction execution history when some transaction T_1 reads a data item, some other transaction T_2 then modifies or deletes that data item and commits, and T_1 then tries to re-read that data item.*

When T_1 rereads the data item, it sees a modified value or finds the value deleted.

DEFINITION A.4. *A* **Phantom (P3)** *occurs in a transaction execution history when some transaction T_1 reads a set of data items satisfying some search condition and some other transaction T_2 then creates or deletes data items that satisfy T_1's search condition and commits.*

If T_1 then rereads the data item with the same search condition, it sees a set of data item values different from its previous read.

Some additional phenomena that are not part of ANSI SQL but are mentioned in [3] are:

DEFINITION A.5. *A* **lost update (P4)** *occurs in a transaction execution history when some transaction T_1 reads a data item and some other transaction T_2 then updates the same data item. T_1 then updates the data item (based on the earlier read) and commits.*

A problem arises in that even if T_2 commits, its update is lost.

DEFINITION A.6. **Write skew (P5)** *occurs in a transaction execution history if some transaction T_1 reads data items x and y and then some other transaction T_2 reads x and y, writes x, and commits. Then, T_1 writes y.*

The problem with P5 is that if there were a constraint between x and y, it could be violated even if each individual transaction's update satisfies the constraint.

Note that serializability requires a history to be conflict-equivalent to a serial history. Thus, none of the phenomena P0 - P5 can occur in a system that guarantees serializability.