A Pure Lazy Technique for Scalable Transaction Processing in Replicated Databases

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

Abstract

Recently, there have been proposals for scaling-up a database system using lazy replication. In these proposals, system scale-up is achieved through the addition of secondary sites which hold replicas of the database at a primary site. The addition of more secondary sites improves system performance by allowing read-only transactions to be serviced at the secondary sites. However, the scalability of the distributed system is limited by the single primary site which services the update workload. As the workload scales-up, an increasing update load is placed on the primary site, which suffers from performance degradation. We address this problem by proposing a pure lazy solution for scaling-up the database system. Our techniques provide scalability and avoid transaction inversions, which can occur when transactions access stale replicas.

1. Introduction

Replication is a technique that is gaining increasing recognition for improving the performance and availability of a database system. Gray et al [10] classified synchronization of replicas into two categories: eager and lazy. In eager synchronization, all copies of a data item are updated by a single transaction. While this makes it relatively easy to guarantee the standard correctness criterion of one-copy serializability (1SR) [3] for replicated data, eager protocols do not work well if all replicas are not available. Moreover, the updating transaction can suffer poor performance in a scaled-up system with a large number of replicas to update [10]. In lazy replicated systems, an update transaction updates one copy of a data item while replicas are updated using separate transactions, thereby avoiding the performance problems of eager protocols.

Consider a database system with two sites: a primary site that holds the database and a secondary site that holds a

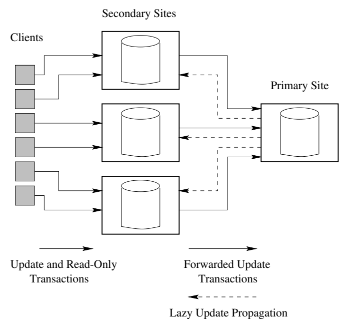
Secondary Sites
Clients
Primary Site
Update and Read-Only Transactions
Forwarded Update Transactions
Lazy Update Propagation

Figure 1. Lazy Master System Architecture

replica of the database at the primary site. Update transactions execute at the primary site and updates are propagated lazily to the secondary site and installed in an order that is consistent with their serialization order at the primary site. If read-only transactions are serviced at the secondary site, 1SR can be ensured in the distributed system. However, as the workload scales-up, the system will not scale unless both the primary and secondary sites are scaled-up.

A lazy master architecture that allows more secondary sites to be added to the distributed system as the workload scales-up has been recently proposed [17, 16, 9, 21]. In this architecture, illustrated in Figure 1, read-only transactions are serviced at secondary sites that have replicas or cached copies of the database at a primary site. However, an increasing update load is placed on the single primary site since all update transactions are processed there. This limits system scalability since the single primary site suffers from performance degradation [9]. The first goal of the techniques presented in this paper is to alleviate this performance bottleneck by scaling-up the single primary site of the lazy master architecture.

A drawback of ensuring only 1SR in lazy replicated databases is that transactions may be serialized in undesirable orders. For example, if a client submits a read-only transaction for execution after an update transaction, the read-only transaction may not see the update made by the client's update transaction. Since update propagation is lazy, the read-only transaction may run against a stale replica, i.e. one that does not include the effects of the client's update transaction. The result is a *transaction inversion*: the read-only transaction precedes the update transaction in the serialization order despite the fact that it follows the update transaction in the client's request stream. In [9], we showed how strong session 1SR can avoid such inversions in lazy master systems with a single primary site.

With multiple primary database sites, the global database state is composed of database states of individual primary sites. The global database state evolves with the execution of update transactions. Thus, the challenge here is to determine a valid serialization order for update transactions that have executed at different primary sites that are synchronized lazily. This serialization order can then be used to install updates at the secondary sites and to avoid transaction inversions by guaranteeing strong session 1SR in the distributed system.

The goal of the techniques presented in this paper is to not only alleviate the bottleneck of a single primary site but also to maintain global 1SR in the system, and to prevent transaction inversions. Our contribution in this paper are techniques to provide scalability and guarantee strong session 1SR, which prevents transaction inversions. Our contribution, presented in Section 3, is a pure lazy solution for scaling-up the primary database and propagating updates to secondary sites. Performance studies show that our techniques can provide scalability and avoid transaction inversions almost as efficiently as 1SR, which does not prevent transaction inversions.

2. Strong Session 1SR

In strong session 1SR, transactions are grouped into sessions. The strong session 1SR correctness criterion avoids transaction inversion within a session but not across sessions. Transactions in an execution history H are grouped into sessions using a session labeling:

Definition 2.1 Session Labeling: *A session labeling L_H of an execution history H assigns a session label (identifier) to each transaction in H.*

We use $L_H(T)$ to refer to the session label of transaction T. In this paper, we associate a distinct session with each client shown in Figure 1, where the session consists of all transactions submitted by that client. Given an execution history H and a labeling L_H, strong session 1SR [9] is defined as follows:

Definition 2.2 Strong Session 1SR: *A transaction execution history H is strong session 1SR under labeling L_H iff it is 1SR and, 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, there is some serial one-copy history equivalent to H in which T_i precedes T_j.*

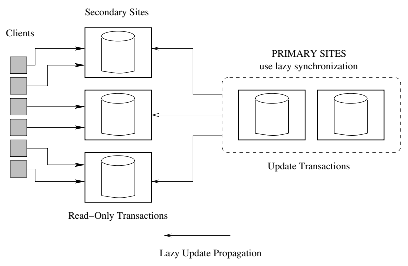

Figure 2. Scaled-up Primary in the Pure Lazy Architecture

In a client's sequence of transactional requests constituting a session, strong session 1SR guarantees that transactions are ordered in the sequence in which they are submitted for execution. Thus, for transactions submitted by different clients, this ordering would not necessarily be preserved. This makes strong session 1SR much less expensive to enforce than *strong serializability* [4], which avoids *all* transaction inversions [9]. Strong session 1SR is guaranteed by the protocol we present in Section 3.

3. A Pure Lazy System

Previous work described techniques for providing strong session 1SR for lazy master systems [9]. However, these techniques are restricted to a single primary site. In this section, we describe a pure lazy master solution for scaling up the primary database site. Our approach is pure lazy in that both synchronization of the primary sites and update propagation to secondaries is performed lazily. Relaxing the single primary site restriction allows the primary database to be partitioned and/or replicated across several sites, resulting in the system shown in Figure 2.

3.1. System Model

The system architecture is illustrated in Figure 3. The system consists of one or more primary sites arranged in a chain, and an arbitrary number of secondary sites. Each site consists of an autonomous database system with a local concurrency controller. Each site guarantees commitment ordering [5], which ensures that transactions are serialized in the order in which they commit.[1]

1 Commitment ordering is enforced by well-known concurrency con-

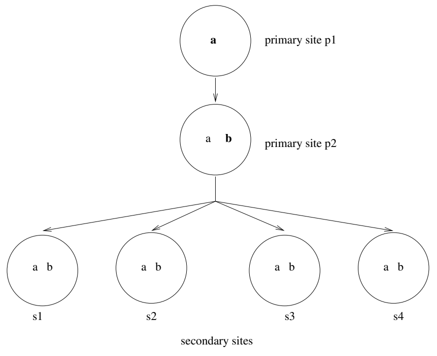

Figure 3. Pure Lazy System Example

As shown in Figure 2, clients connect to the secondary sites and issue transaction requests. Update transactions are forwarded to a primary site and executed there, while read-only transactions execute at the secondaries. Updates are propagated lazily down the chain of primary sites, and from the last primary site to the secondaries.

Each data item has a primary copy, which is located at one of the primary sites. Different data items may have primary copies at different sites. Using Figure 3 as an example, data items a and b (in bold) can have a primary copy at sites p1 and p2, respectively. In addition, a primary site p can hold replicas of data items whose primary copies are located at sites that precede p in the primary site chain. Secondary sites hold replicas of all data items. In the example illustrated in Figure 3, data item a (with primary copy at site p1) is replicated at primary site p2 and, together with b, is also replicated at each secondary site. Lazy master replication configurations to suppport single-site partitioned workloads have been proposed [6, 11, 12].

The primary site where an update transaction commits initiates refresh transactions[2] at its descendant(s) (down the chain) in order to propagate its updates. At each site, refresh transactions are committed in the order in which they are received. Once a refresh transaction has committed at a site, that site then initiates the refresh transaction at its descendant(s), and so on until the updates have been propagated to all descendants of the site at which the original update transaction ran. If a refresh or update transaction T_1 commits before T_2 at a site s_i, then s_i must forward T_1 before T_2 to its children. This is termed first-in-first-out (FIFO) execution of refresh transactions.

The system that we have just described implements a restricted version of the DAG(WT) protocol [6], which is used to guarantee 1SR in lazily updated replicated database systems. However, a key drawback is that the system does not guarantee strong session 1SR, and that transaction inversions can occur. In the following sections, we show how the basic system can be modified to prevent transactions inversions by enforcing strong session 1SR while providing primary site scalability.

trol protocols such as strict two-phase locking in which no locks are released until after commit.

2 A refresh transaction is used to represent the updates, corresponding to a single update transaction, propagated lazily to other sites.

3.2. Representing Global Primary Database State

To provide the strong session 1SR guarantee under the pure lazy approach, a globally unique label is needed to represent the global serialization order. We propose using lexicographically ordered vectors to keep track of the global serialization order of transactions. In Section 3.3, we will use this representation of the global serialization order to guarantee strong session 1SR, which avoids transaction inversions, in the presence of multiple primary sites.

3.2.1. Lexicographically Ordered Vectors (L-Vectors) We use an L-vector at each database site to represent that site's knowledge of the global system state, which changes as a result of updates to database primary items in the distributed system.

Every site maintains an L-vector. An L-vector is a vector of integers of length n, where n is the number of primary sites in the distributed system. We will use t_i to denote the L-vector at site s_i, and $t_i[j]$ to denote its jth component. Component $t_i[j]$ $(j \neq i)$ is used to count the number of update transactions originating at site s_j whose updates have been applied at s_i. Component $t_i[i]$ is the local transaction counter at site s_i. When an update transaction commits at s_i, $t_i[i]$ is incremented. When a refresh transaction originating from an update at site s_j commits at site s_i, $t_i[j]$ is incremented.

We also associate an L-vector with each update transaction and read-only transaction. The L-vector of a transaction T that runs at site s is the L-vector of s after T commits. It should be easy to see that no two update transactions have (are associated with) the same L-vector. However, two read-only transactions may have the same L-vector, and a read-only transaction may have the same L-vector as an update transaction.

Transaction L-vectors are useful because they capture the transaction serialization order. That is, if $T1$ has a smaller L-vector than $T2$, then $T1$ can be serialized before $T2$. L-vector comparison is done lexicographically [8].[3]

Definition 3.1 *Given two L-vectors* $t_1 = (a_0, a_1, ..., a_n)$ *and* $t_2 = (b_0, b_1, ..., b_n)$, $t_1 < t_2$ *if there exists an inte-*

3 Note that L-vectors are different from version vectors in that the comparison of version vectors [20] is not lexicographic, i.e. two version vectors can be compared only if, for all components of a vector, the value of each component of one version vector is less or equal to the value of that component in the other version vector.

ger j, $0 \leq j \leq n$, such that $a_i = b_i$ for all i=0, 1, ..., j-1 and $a_j < b_j$.

Definition 3.2 *Given two L-vectors $t_1 = (a_0, a_1, ..., a_n)$ and $t_2 = (b_0, b_1, ..., b_n)$, $t_1 = t_2$ if $a_i = b_i$ for all i=0, 1, ..., n.*

3.3. Ensuring Strong Session 1SR

In this section we present an adapted form of the BLOCK [9] concurrency control algorithm for enforcing strong session 1SR while providing scalability. If global strong session 1SR in a system with multiple primary database sites is to be preserved, a session needs to know the global serialization order of all update transactions. BLOCK uses the serialization order information stored as L-vectors, described in section 3.2.1, to enforce global strong session 1SR. The session managers, together with the propagation, refresh, and local concurrency control mechanisms described in Section 3.1, are responsible for enforcing strong session 1SR.

In addition to the site L-vectors described in Section 3.2.1, every client session c at a site s_i also maintains a session L-vector denoted by t_c. t_c contains an entry for every site in the system. When a new session starts at site s_i, t_c is initialized to site s_i's L-vector t_i. After an update or read-only transaction from session c commits at site s_j, the value of t_j, $val(t_j)$, is returned to the session c together with the transaction result and t_c is set to $val(t_j)$.

A session's L-vector indicates database state "seen" by the most recently committed transaction in that session. Since the database state at a site, represented by the site's L-vector, is maintained by refresh transactions, the site's L-vector denotes the current database state at the site. The BLOCK algorithm is used with L-vectors and the algorithm is run at each primary site as well as at the secondary sites. This is to ensure that the correct ordering is maintained among all transactions in a session. The algorithm ensures that a transaction from a session is allowed to execute at a site if the condition that the site's t_{site} L-vector is lexicographically at least as large as the session's $t_{session}$ L-vector is satisfied. Otherwise, the transaction blocks until the above condition is satisfied. The $t_{session}$ value is piggybacked onto an update transaction when it is forwarded to a primary site only for the purposes of checking the blocking condition.[4] The BLOCK algorithm, plus the local concurrency control at each site, is sufficient to ensure that each transaction will be serialized after its predecessors in its session.

Example: Consider a client session c at site s3 in the system from Figure 3. Initially, t_{site}(p1) or simply t_{p1}, t_{p2} and $t_{session}(c)$ or simply t_c, are $(0,0)$. If an update transaction T_1 from session c is to update the primary copy of data item a located at site p1, the transaction will be forwarded to p1. After T_1 commits at site p1, t_{p1} becomes $(1,0)$. t_c is now also $(1,0)$. Next, if a read-only transaction T_2 from c is to access a, it will execute at s3 if $t_{s3} \geq t_c$, which will be the case after T_1's update corresponding to the L-vector $(1,0)$ is installed at s3. Ordering guarantees can also be provided to transactions that execute at primary sites. Consider an update transaction T_3 from session c that is to execute at site p2. To ensure that T_3 is serialized after T_1 and T_2, T_3 will block at p2 if $t_{p2} < t_c$. Otherwise, it will be allowed to execute there. Upon T_3's commit, t_{p2} will become $(1,1)$ and t_c will be updated to $(1,1)$.

The pure lazy system guarantees 1SR [6]. The BLOCK algorithm and L-vectors give us strong session 1SR in pure lazy systems.[5]

4. Simulation Model

A simulation model of the pure lazy replicated system has been developed. The model is implemented in C++ using the CSIM simulation package [18].

The execution of transactions are simulated by client processes accessing resources, which are servers or sites. All transactions of a client process, or simply client, are submitted to the secondary site to which the client is connected. The distribution of clients over the secondary sites is uniform.

Each client starts a series of sessions over which a sequential stream of transactions is submitted. Session lengths follow an exponential distribution with a mean of *session_time*. Clients think between transactions, where the think times are exponentially distributed with a mean of of *think_time*. *num_clients* is the total number of sessions in the system at any given time. A new client session is started when the previous session ends. The mean values of *session_time* and *think_time* are as specified in the TPC-W benchmark.

Each update transaction executes at a primary site while each read-only transaction executes at a secondary site. The distribution of update transactions over the primary sites is uniform with equi-probability. Each transaction goes through its execution site's local concurrency control. Each new transaction can conflict with each waiting or running transaction with a probability of *conflict_prob* at its execution site. In addition, each concurrency control is modelled so that a transaction that conflicts with any waiting or executing transactions waits for the conflicting transaction(s) to finish. Otherwise, the transaction is allowed to execute.

4 Since the piggybacked value of $t_{session}$ is used only for the duration of this transaction, synchronization of the variable is not required.

5 Proofs omitted due to space constraints.

The probability that a transaction is an update transaction is *update_tran_prob* and that it is a read-only transaction is *(1 - update_tran_prob)*. The default mix of read-only/update transactions that we use for our workload is 80%/20% but we have also run 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.[6]

Each transaction has a mean *tran_size* number of operations, randomly chosen between 5 to 15. The probability that an operation in an update transaction is an update operation is *update_op_prob*. Otherwise, it is a read operation. Transactions execute at a site by accessing that site's shared server resource, which has a round-robin discipline time-sliced at 0.001 seconds.

Propagation processes running at each site are used to simulate the propagation mechanism. There are two propagation processes per site: a sender process and a receiver process. At each site, the sender propagation process goes through a series of propagation cycles, with a delay of *propagation_delay* time units between the end of one cycle and the beginning of the next. During each propagation cycle, the process propagates all transactions that have committed at the site since the last propagation cycle. To do this, the process generates a message describing the updates of all of these transactions and broadcasts the message to the site's children, if any. The propagation process consumes *op_service_time* during each propagation cycle. It does not use the local concurrency control, since we assume that the propagator is implemented as a log sniffer. After a refresh transaction at a site is committed, the refresher propagates the transaction's updates as a single message to the site's children, if any. The simulation does not include an explicit resource to represent the network. 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.

At each secondary site, there is a receiver propagation process that receives the broadcast propagation messages and installs the transaction update information into a single update queue in the local database. The propagation process consumes *op_service_time* for each propagation message it handles. It does not use the local concurrency control when it inserts update information into the update queue. The queue operations conflict only with the refresh processes, which read from the other end of the queue. Thus, we assume that any contention would be insignificant.

6 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.

Parameter	Default
num_clients	20 per secondary
think_time	7s
session_time	15 min.
update_tran_prob	20%
conflict_prob	5%
tran_size	10
op_service_time	0.02s
update_op_prob	20%
propagation_delay	10s

Table 1. Simulation Model Parameters

A refresh process runs at each secondary site. Updates enqueued into an update queue are read by the refresh process. Each update message (or update record) inserted into the update queue contains the updates of a single update transaction. The refresh process submits a refresh transaction corresponding to a single update record to the local concurrency control, where the refresh transaction has probability *conflict_prob* for conflicting with other transactions at the site. Every refresh transaction is made to conflict with every other refresh transaction. This forced conflict enables the refresh transactions to be installed in the order in which they were committed at the primary sites. The refresher takes *op_service_time* to read a transaction's updates from the update queue and *op_service_time* for every update operation executed at a secondary site.

5. Performance Analysis

We used our simulation model to study system scalability and transaction inversion avoidance. We had three objectives in mind: (i) to observe the effect on system performance as the workload was scaled-up (ii) to determine the cost of providing strong session 1SR over 1SR while providing scalability, and (iii) to determine the factors that affected scalability.

We compare the performance of the BLOCK algorithm against algorithm ALG-1SR, which provides only global serializability (1SR) and not strong session 1SR. ALG-1SR provides no session guarantees and simply routes all update transactions to a primary site and all read-only transactions to the secondary site. ALG-1SR never blocks transactions though they may be blocked by the local concurrency control at their execution site. ALG-1SR is an implementation of the DAG(WT) [6] protocol.

5.1. Methodology

For each run, the simulation parameters were set to the default values shown in Table 1, except as indicated in the

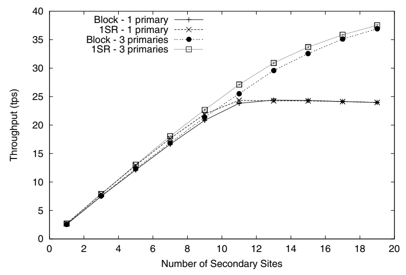

Figure 4. Transaction Throughput, 20 Clients per Secondary, 80/20 mix

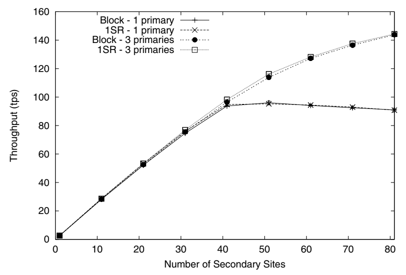

Figure 5. Transaction Throughput, 20 Clients per Secondary, 95/5 mix

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 and other statistics over the remainder of the run. 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.

5.2. Experimental Results

Since the goal of our system is that it should be able to scale with the workload, we ran experiments in which both the workload and system resources were increased in-step. The workload was scaled-up by increasing the number of clients while the resources were scaled-up by increasing the number of secondary sites and primary sites. We measured the performance of the system as extra primary sites were added to the system while the number of clients per secondary site was held constant at 20. The results of these experiments are shown as transaction throughput plots. Each curve shows the behavior of either ALG-1SR or BLOCK.

We present results of experiments to study the scale-up behavior of the BLOCK algorithm, which avoids transaction inversion by providing the strong session 1SR guarantee. Figure 4 shows the throughput results for the pure lazy system under the BLOCK and ALG-1SR algorithms. For the BLOCK algorithm, transaction throughput scales-up as more primary sites are added to the system. Multiple primary sites are able to sustain higher update loads than a single primary site, giving rise to significant increases in throughput as shown in Figure 4. With a single primary site, the pure lazy system suffers from resource contention. Addition of two extra primary sites significantly alleviates resource contention and throughput at high load levels is higher by about 70% over having a single primary site. As the workload scales-up past 20 secondary sites, ultimately data contention dominates resource contention in limiting scalability. Beyond this point, significantly large performance improvement is not observed since the availability of extra system resources does not alleviate data contention.

A key question is whether both scalability and transaction inversion avoidance can be provided efficiently. That is, we need to quantify the cost of providing strong session 1SR over 1SR. As shown by the throughput results in Fig. 4 and the results for the 95/5 read/write transaction mix in Fig. 5, scalability *and* transaction inversion avoidance can be provided efficiently. The BLOCK algorithm performs almost as well as ALG-1SR even though BLOCK prevents transaction inversions while ALG-1SR does not. Scalability is sensitive to the mix of read and write operations in the workload. As Figure 5 shows, significantly greater scalability can be attained with a 95/5 read/write transaction mix.

6. Related Work

There has been recent interest in improving the performance of eager replication protocols or using lazy protocols. Techniques that address the performance drawbacks of eager protocols have been proposed but they rely on the availability and performance of group communication protocols [14]. The viability of lazy master replication has been proved by recent research and commercial approaches that use data that is replicated or cached at secondary sites from the primary site to service the read-only portion of the workload [17, 16, 21, 9]. However, none of the proposed techniques address the problem of how the single primary site can be scaled-up in a pure lazy system while preventing transaction inversions.

Ladin et al [15] proposed using a 2-phase commit protocol to propagate updates to a majority of replicated sites to prevent transaction inversions. However, they do not consider the provision of scalability and session-level guarantees. Bayou [23] is a system that can provide session-based causal ordering constraints on read and write operations. However, since only eventual consistency is guaranteed in

Bayou, transaction inversions are not prevented in the presence of updates.

Several replication protocols have been proposed to guarantee 1SR in lazy master replicated database systems. Breitbart and colleagues have proposed DAG protocols [6], where different database objects may have their primary copies located at different sites, and an acyclic site graph is used to guide the propagation of updates among the sites. Other related work on concurrency control protocols includes the virtual sites protocol of Breitbart and Korth, the quorum consensus protocol of Satyanarayanan and Agrawal, which uses a gossip mechanism to lazily propagate updates to sites that have missed them, and the epidemic update propagation protocol of Agrawal et al [7, 22, 1]. Amza et al [2] address the database scale-up problem using distributed multiversioning of data. Jimenez-Peris et al update replicas using an optimistic group communication protocol [13]. Pacitti et al [19] proposed a lazy propagation protocol in a synchronous environment that allows a global ordering of refresh transactions. None of the above techniques show how to guarantee strong session 1SR in an asynchronous pure lazy system with multiple primary sites.

7. Conclusion

In this paper, we proposed techniques for providing scalability and avoiding transaction inversion in lazy replicated databases. Our technique, a pure lazy approach, employs lexicographically ordered vectors to avoid transaction inversions and allows scalability of the primary database through partitioning and replication.

We studied the performance of our algorithms and found that our techniques cost almost the same as 1SR, which does not prevent transaction inversions. We conclude that our proposed solution is a viable technique for achieving scalability and preventing transaction inversions in lazy replicated database systems.

References

[1] D. Agrawal, A. E. Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proc. PODS*, pages 161–172, 1997.

[2] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. Middleware*, pages 282–304, 2003.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–293, 1992.

[5] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Trans. Soft. Eng.*, 17(9):954–960, 1991.

[6] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proc. SIGMOD*, pages 97–108, 1999.

[7] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proc. PODS*, pages 173–184, 1997.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[9] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Proc. ICDE*, pages 424–435, 2004.

[10] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. SIGMOD*, pages 173–182, 1996.

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

[12] Informix Corp. *Enterprise Replication: A High Performance Solution for Distributing and Sharing Information*, 1998. Whitepaper.

[13] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. Improving the scalability of fault-tolerant database clusters. In *ICDCS*, pages 477–484, 2002.

[14] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, 2000.

[15] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4):360–391, 1992.

[16] P.-A. Larson, J. Goldstein, and J. Zhou. Mtcache: Mid-tier database caching in sql server. In *Proc. ICDE*, pages 177–188, 2004.

[17] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proc. SIGMOD*, pages 600–611, 2002.

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

[19] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2001.

[20] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of Mutual Inconsistency in Distributed Systems. *TSE*, 9(3):240–247, 1983.

[21] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proc. Middleware*, 2004.

[22] O. T. Satyanarayanan and D. Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *TKDE*, 5(5):859–871, 1993.

[23] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Proc. PDIS*, pages 140–149, 1994.