Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites

Cristiana Amza[1], Alan L. Cox[1] and Willy Zwaenepoel[2]

[1] Department of Computer Science, Rice University, Houston, TX, USA
[2] School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland
{amza, alc}@cs.rice.edu, willy.zwaenepoel@epfl.ch

Abstract. Dynamic content Web sites consist of a front-end Web server, an application server and a back-end database. In this paper we introduce distributed versioning, a new method for scaling the back-end database through replication. Distributed versioning provides both the consistency guarantees of eager replication and the scaling properties of lazy replication. It does so by combining a novel concurrency control method based on explicit versions with conflict-aware query scheduling that reduces the number of lock conflicts.
We evaluate distributed versioning using three dynamic content applications: the TPC-W e-commerce benchmark with its three workload mixes, an auction site benchmark, and a bulletin board benchmark. We demonstrate that distributed versioning scales better than previous methods that provide consistency. Furthermore, we demonstrate that the benefits of relaxing consistency are limited, except for the conflict-heavy TPC-W ordering mix.

1 Introduction

Web sites serving dynamic content usually consist of a Web server, an application server and a back-end database (see Figure 1). A client request for dynamic content causes the Web server to invoke a method in the application server. The application server issues a number of queries to the database and formats the results as an HTML page. The Web server then returns this page in an HTTP response to the client.

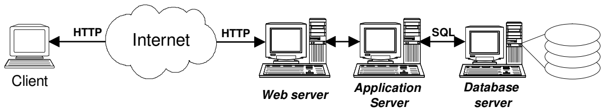

Fig. 1. Common architecture for dynamic content Web sites

Replication [4, 13, 15, 27] of the database back-end allows improved data availability and performance scaling. Providing consistency at the same time as performance scaling has, however, proven to be a difficult challenge. Eager replication schemes, which provide strong consistency (1-copy serializability [7]), severely limit performance, mainly due to conflicts [11]. Lazy replication with delayed propagation of modifications provides better performance, but writes can arrive out-of-order at different sites and reads can access inconsistent data.

Recent work has argued that several distinct consistency models should be supplied, since dynamic content applications have different consistency requirements. Neptune proposes three levels of consistency [22]. An extension of this idea proposes a continuum of consistency models with tunable parameters [28]. The programmer then chooses the appropriate consistency model and the appropriate parameters for the application or adjusts the application to fit one of the available models. Adjusting the application may require non-trivial programmer effort.

In this paper we introduce distributed versioning, a technique that maintains strong consistency (1-copy serializability [7]) but at the same time allows good scaling behavior. Distributed versioning improves on our earlier work on conflict-aware schedulers [3] in two ways:

1. A limitation of the previous scheme is the use of conservative two-phase locking which, while avoiding deadlocks, severely limits concurrency. We introduce a novel deadlock-free concurrency control algorithm based on explicit versions, which allows increased concurrency. Distributed versioning integrates this concurrency control algorithm with a conflict-aware scheduler to improve performance over the methods introduced earlier [3].
2. We investigate the overhead of using 1-copy serializability compared to looser consistency models provided by lazy replication. We study this overhead using a variety of applications with different consistency requirements.

In our evaluation we use the three workload mixes (browsing, shopping and ordering) of the TPC-W benchmark [26], an auction site benchmark [1], and a bulletin board benchmark [1]. We have implemented these Web sites using three popular open source software packages: the Apache Web server [5], the PHP Web-scripting/application development language [18], and the MySQL database server [16]. Our results are as follows:

1. Distributed versioning increases throughput compared to a traditional (eager) protocol with serializability by factors of 2.2, 4.8, 4.3, 5.4, and 1.1 for the browsing, shopping, ordering mixes of TPC-W, the auction site and the bulletin board, respectively, in the largest configuration studied.
2. For the browsing and shopping workloads of TPC-W and for the bulletin board, distributed versioning achieves performance within 5% of the best lazy protocol with loose consistency. The auction site's performance is within 25%. The difference is larger in the TPC-W ordering mix, because of the large number of conflicts, but the best lazy protocol does not respect the application's semantics.
3. There is no penalty for enforcing serializability for applications with loose consistency (e.g., the bulletin board).

The outline of rest of the paper is as follows. Section 2 describes the programming model, the consistency model and the cluster design used for distributed versioning. Section 3 introduces distributed versioning. Section 4 describes our prototype implementation. Sections 5 describes the consistency models and the implementation of the different lazy protocols with loose consistency models explored in the paper. Section 6 presents our benchmarks. Section 7 presents our experimental platform and our evaluation methodology. We investigate how distributed versioning affects scaling, and compare it against the other lazy protocols in Section 8. Section 9 discusses related work. Section 10 concludes the paper.

2 Environment

This section describes the environment in which distributed versioning is meant to work. In particular, we describe the programming model, the desired consistency, and the cluster architecture.

2.1 Programming Model

A single (client) Web interaction may include one or more transactions, and a single transaction may include one or more read or write queries. The application writer specifies where in the application code transactions begin and end. In the absence of transaction delimiters, each single query is considered a transaction and is automatically committed (so called ”auto-commit” mode).

At the beginning of each transaction consisting of more than one query, the programmer inserts a *pre-declaration* of the tables accessed in the transaction and their modes of access (read or write). The tables accessed by single-operation transactions do not need to be pre-declared. Additionally, the programmer inserts a *last-use* annotation after the last use of a particular table in a transaction. These annotations are currently done by hand, but could be automated.

2.2 Consistency Model

The consistency model we use for distributed versioning is 1-copy-serializability [7]. With 1-copy-serializability, conflicting operations of different transactions execute in the same order on all replicas (i.e., the execution of all transactions is equivalent to a serial execution in a total order).

2.3 Cluster Architecture

We consider a cluster architecture for a dynamic content site, in which a scheduler distributes incoming requests on a cluster of database replicas and delivers the responses to the application server (see Figure 2). The scheduler may itself be replicated for performance or for availability. The application server interacts directly only with the schedulers. If there is more than one scheduler in a particular configuration, the application server is assigned a particular scheduler at the beginning of a transaction by round-robin. For each query of this transaction, the application server only interacts with this single scheduler. These interactions are synchronous: for each query, the application server blocks until it receives a response from the scheduler. To the application servers, a scheduler looks like a database engine. At the other end, each database engine interacts with the scheduler as if it were a regular application server. As a result, we can use any off-the-shelf Web server (e.g., Apache) and application server (e.g., PHP), and any off-the-shelf database (e.g., MySQL) without modification. When more than one front-end node is present, an L4 switch is also included. The use of an L4 switch makes the distributed nature of the server transparent to the clients.

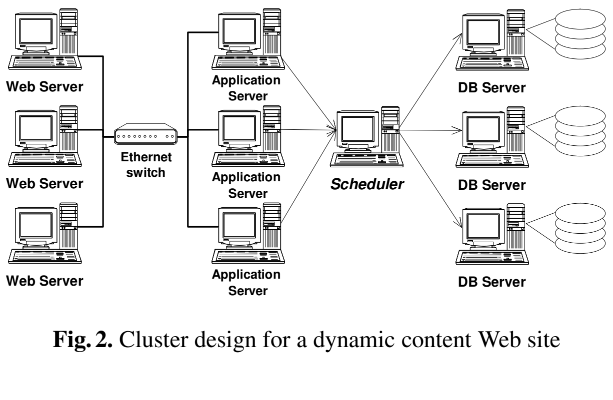

Fig. 2. Cluster design for a dynamic content Web site

3 Distributed Versioning

Distributed versioning achieves 1-copy serializability, absence of deadlock and a high degree of concurrency using a lazy read-one, write-all replication scheme augmented with version numbers, as described next.

3.1 Lazy Read-one, Write-all Replication

When the scheduler receives a write or a commit query from the application server, it sends it to all replicas and returns the response as soon as it receives a response from any of the replicas. Reads are sent only to a single replica, and the response is sent back to the application server as soon as it is received from that replica.

3.2 Assigning and Using Version Numbers

A separate version number is maintained for each table in the database. A transaction is assigned a version number for each table that it accesses (except for single-read transactions, see below). As discussed in Section 2.1, each multi-query transaction declares what tables it reads or writes before it starts execution. The tables accessed by single-query transactions are implicitly declared by the query itself. Based on this information, the scheduler assigns table versions to be accessed by the queries in that transaction. This assignment is done *atomically*, i.e., the scheduler assigns all version numbers for one transaction, before it assigns any version numbers for the next transaction. Version number assignment is done in such a way that, if there is a conflict between the current transaction and an earlier one, the version numbers given to the current transaction for the tables involved in the conflicts are higher than the version numbers received by the earlier conflicting transaction.

All operations on a particular table are executed at all replicas in version number order. In particular, an operation waits until its version is available. New versions become available as a result of a previous transaction committing or as a result of last-use declarations (see Section 3.5).

Transactions consisting of a single read query are treated differently. No version numbers are assigned. Instead, the query is simply forwarded to one of the replicas, where it executes after all conflicting transactions complete. This optimization results in a very substantial performance improvement without violating 1-copy serializability.

3.3 1-Copy Serializability and Absence of Deadlock

If transactions have conflicting operations involving one or more tables, then the version numbers for the conflicting tables assigned to the earlier transaction are strictly lower than those assigned to the same tables for the later transaction. Since all conflicting operations execute in version number order at all replicas, all conflicting operations of all transactions execute in the same total order at all replicas. Hence, 1-copy serializability is established.

A similar argument shows that distributed versioning avoids deadlock. For all tables that cause conflicts between transactions, the version numbers assigned to one transaction must be either all smaller or all larger than those assigned to another transaction. Since transactions only wait for the completion of operations with a lower version number than their own, there can never be a circular wait, and therefore deadlock is avoided.

3.4 Limiting the Number of Conflicts

The scheduler sends write queries to all replicas and relies on their asynchronous execution in order of version numbers. At a given time, a write on a data item may have been sent to all replicas, but it may have completed only at a subset of them. A conflict-aware scheduler [3] maintains the completion status of outstanding write operations, and the current version for each table at all database replicas. Using this information, the scheduler sends a read that immediately follows a particular write in version number order to a replica where it knows the write has already finished (i.e., the corresponding required version has been produced). This avoids waiting due to read-write conflicts.

3.5 Reducing Conflict Duration

In the absence of last-use declarations in a transactions, the versions of various tables produced by the current transaction become available only at commit time. The presence of a last-use declaration allows the version to be produced immediately after the time that declaration appears in the application code. This *early release* of a version reduces the time that later transactions have to wait for that version to become available. Early releases do not compromise 1-copy-serializability. All versions are atomically pre-assigned at the beginning of each transaction, and a version release occurs only after the last use of a particular table. Hence, the total ordering of conflicting transactions at all replicas is the same as the one in the system without early releases.

3.6 Rationale

In our earlier work [3] we use conservative two-phase locking as the concurrency control method for conflict-aware scheduling. We now demonstrate why distributed versioning leads to more concurrency than conservative 2PL.

In both conservative 2PL, and distributed versioning the declaration of which tables are going to be accessed by a transaction is done at the beginning of the transaction. The behavior of the two schemes in terms of waiting for conflicts to be resolved is, however, totally different. In particular, conflict waiting times are potentially much lower for distributed versioning, for two reasons. First, in conservative 2PL, a particular transaction waits at the beginning until all its locks become available. In contrast, in distributed versioning, there is no waiting at the beginning of a transaction. Only version numbers are assigned. Waiting occurs when an operation tries to access a table for which conflicting operations with an earlier version number have not yet completed. The key difference is that at a given operation, distributed versioning only waits for the proper versions of the tables in that particular operation to become available. Second, with conservative 2PL, all locks are held until commit. In contrast, with distributed versioning, a new version of a table is produced as soon as a transaction completes its last access to that table. In summary, the increased concurrency of distributed versioning comes from more selective (per-table) waiting for conflicts to be resolved and from earlier availability of new versions of tables (early version releases).

We illustrate the increase in concurrency with an example. Assume that transactions T_0, and T_1 both execute the code shown in Figure 3, writing three different tables. Assume also that transaction T_0 is serialized before transaction T_1. In conservative 2PL, transaction T_1 waits for the locks on all three tables to be freed by T_0 before it starts executing (see Figure 4). In contrast, with distributed versioning the operations on the different tables are pipelined. This example also clearly demonstrates that, in general, both features of distributed versioning (selective per-table waiting and early availability of versions) are essential. Any single feature in isolation would produce the same behavior as conservative 2PL and thus less concurrency.

```
begin
   write a
   write b
   write c
end
```

Fig. 3. Sequence of updates in a transaction

```
      1  2  3  4  5  6
T0:  a0,b0,c0
T1:           a1,b1,c1
```

```
      1  2  3  4
T0:  a0,b0,c0
T1:     a1,b1,c1
```

Fig. 4. Serial execution in conservative 2PL (left) versus increased concurrency in distributed versioning (right)

One may wonder if similar benefits are not available with alternative 2PL schemes. This is not the case. Selective waiting for locks can be achieved by implicit 2PL, in which locks are acquired immediately before each operation. Implicit 2PL achieves selective waiting, but at the expense of potential deadlocks. Given that the probability of deadlock increases approximately quadratically with the number of replicas [11], any concurrency control algorithm that allows deadlock is undesirable for large clusters. Even if a deadlock-avoidance scheme could be used in conjunction with selective waiting for locks, early releases of locks are limited by the two-phase nature of 2PL, necessary for achieving serializability.

4 Implementation

4.1 Overview

The implementation consists of three types of processes: scheduler processes (one per scheduler machine), a sequencer process (one for the entire cluster), and database proxy processes (one for each database replica).

The sequencer assigns unique version numbers to the tables accessed by each transaction. The database proxy regulates access to its database server by only letting a query proceed if the database has the right versions for the tables named in the query. The schedulers receive the various operations from the application server (begin transaction, read and write queries, and commit or abort transaction), forward them as appropriate to the sequencer and/or one or all of the database proxies, and relay the responses back to the application server. In the following, we describe the actions taken by the scheduler and database proxy when each type of operation is received for processing, and when its response is received from the database.

4.2 Transaction Start

The application server informs the scheduler of all tables that are going to be accessed, and whether a particular table is read or written. The scheduler forwards this message to the sequencer (see Figure 5-a). The sequencer assigns version numbers to each of the tables for this transaction, and returns the result to the scheduler. The scheduler stores this information for the length of the transaction. It then responds to the application server so that it can continue with the transaction. The version numbers are not passed to the application server.

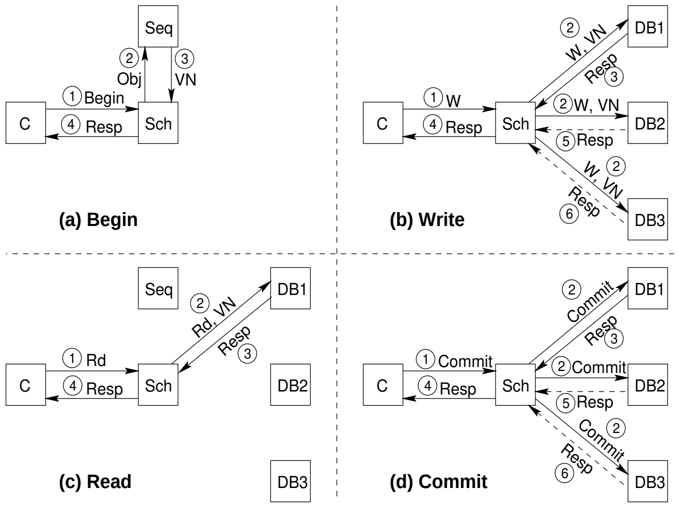

Fig. 5. Protocol steps for each type of query and query response

```
operation              w  w  r  w  r  r  r  w
next_for_read        0  1  2  2  4  4  4  4  7
next_for_write       0  1  2  3  4  5  6  7  7
version assigned       0  1  2  3  4  4  4  7
```

Fig. 6. Sequencer assigned version numbers for a series of operations

For each table, the sequencer remembers two values: the sequence number `next-for-read`, to be assigned if the next request is for a read, and the sequence number `next-for-write`, to be assigned if the next request is for a write. When the sequencer receives a request from the scheduler for a set of version numbers for tables accessed in a particular transaction, the sequencer returns for each table the `next-for-read` or the `next-for-write` sequence number, depending on whether that particular table is to be read or written in that transaction. After a sequence number is assigned for a write, `next-for-write` is incremented and `next-for-read` is set to the new value of `next-for-write`. After a sequence number is assigned for a read, only `next-for-write` is incremented.

The intuition behind this version number assignment is that the version number assigned to a transaction for a particular table increases by one every time the new transaction contains a conflicting operation with the previous transaction to access that table. For example, Figure 6 shows a series of read and write operations on a particular table, each belonging to a different transaction, in the order of arrival of the transaction's version number request at the sequencer. The figure also shows the version numbers assigned by the sequencer for that table to each transaction and the values of `next-for-read` and `next-for-write`. As long as the successive accesses are reads, their transactions are assigned the same version number. Whenever there is a read-write, write-read, or write-write conflict, a higher version number is assigned.

The assignment of version numbers for a particular transaction is *atomic*. In other words, all version numbers for a given transaction are assigned before any version number for a subsequent transaction is assigned. As a result, the version numbers for all tables accessed by a particular transaction are either less than or equal to the version numbers for the same tables for any subsequent transaction. They are only equal if the transactions do not conflict.

4.3 Read and Write

As the application server executes the transaction, it sends read and write queries to the scheduler. In the following, we explain how the scheduler and database proxies enforce the total order for read and write operations necessary for 1-copy-serializability.

Enforcing 1-copy-serializability. Both for read and write queries, the scheduler tags each table with the version number that was assigned to that table for this transaction. It then sends write queries to all replicas, while read queries are sent only to one replica (see Figure 5-b and c).

The following rules govern the execution of a query:

- A write query is executed only when the version numbers for each table at the database match the version numbers in the query.
- A read query is executed only when the version numbers for each table at the database are greater than or equal to the version numbers in the query.

If a write query needs to wait for its assigned versions at a particular replica, it is blocked by the database proxy at that replica. If a read query needs to wait, it is blocked at the scheduler until one of the replicas becomes ready to execute the query.

In more detail, the scheduler keeps track of the current version numbers of all tables at all database replicas. The scheduler blocks read queries until at least one database has, for all tables in the query, version numbers that are greater than or equal to the version numbers assigned to the transaction for these tables. If there are several such replicas, the least loaded replica is chosen.

If there is only a single scheduler, then it automatically becomes aware of version number changes at the database replicas as a result of responses to commits or early version releases. If multiple schedulers are present, extra communication is needed to inform the schedulers of version number changes resulting from transactions handled by other schedulers.

Single-Read Transactions. Since a single-read transaction executes only at one replica, there is no need to assign cluster-wide version numbers to such a transaction. Instead, the scheduler forwards the transaction to the chosen replica, without assigning version numbers. At the chosen database replica, the read query executes after the update transaction with the highest version numbers for the corresponding tables in the proxy's queues releases these table versions.

Because the order of execution for a single-read transaction is ultimately decided by the database proxy, the scheduler does not block such queries. In case of conflict, the read query waits at the database proxy. The scheduler attempts to reduce this wait by selecting a replica that has an up-to-date version of each table needed by the query. In this case, up-to-date version means that the table has a version number greater than or equal to the highest version number assigned to any previous transaction on that table. Such a replica may not necessarily exist.

4.4 Completion of Reads and Writes

On the completion of a read or a write at the database (see Figure 5-b and c), the database proxy receives the response and forwards it back to the scheduler.

The scheduler returns the response to the application server if this is the first response it received for a write query or it is the response to a read query. The scheduler keeps track of the state of outstanding writes and updates its internal data structures when one of the database engines sends back a reply.

4.5 Early Version Releases

The scheduler uses the last-use annotation to send an explicit version_release message that increments the specified table's version at each database.

4.6 Commit/Abort

The scheduler tags the commit/abort with the tables accessed in the transaction, their version numbers and a corresponding version_release flag, and forwards the commit/abort to all replicas (see Figure 5-d). The transaction's commit carries a version_release flag only for the tables where early version releases have not already been performed. Single-update transactions carry an implicit commit (and version_release).

Upon completion at a database, the corresponding database proxy increments the version number of all tables for which a version_release flag was included in the message from the scheduler. It returns the answer to the scheduler, which updates its state to reflect the reply. If this is the first reply, the scheduler forwards the response to the application server.

4.7 1-Copy-Serializability

The algorithm achieves 1-copy-serializability by forcing transactions that have conflicting operations on a particular table to execute in the total order of the version numbers assigned to them.

A transaction containing a write on a table conflicts with all previous transactions that access the same table. Therefore, it needs to execute after all such transactions with lower version numbers for that table. This is achieved by the combination of the assignment of version numbers and the rule that governs execution of write queries at a database replica, as seen by the following argument:

1. `next-for-write` counts all the earlier transactions that access the same table. This value is assigned as the version number for the table for this transaction.
2. The database proxy increments its version number every time a transaction that accesses that table completes.
3. Since the transaction is allowed to execute only when its version number for the table equals the version number for that table at the database proxy, it follows that all previous transactions that have accessed that table have completed.

A transaction containing a read on a table conflicts with all previous transactions containing a write on the same table. It follows that it needs to execute after the transaction containing a write on that table with the highest version number lower than its own. This is again achieved by the combination of the assignment of version numbers and the rule that governs execution of read queries at a database replica, as seen by the following argument:

1. `next-for-read` remembers the highest version number produced by a transaction with a write on this table. This value is assigned to the transaction as the version number for this table.

2. The current transaction is not allowed to execute at a database proxy before the version number for that table at that database proxy reaches (at least) the transaction's version number for this table.
3. The algorithm also allows a read query to execute at a database proxy if the database proxy's version number for the table is higher than that of the transaction. The only way this can happen is as a result of a sequence of transactions with reads on the table, and these can execute in parallel without violating the total order on conflicting operations.

In figure 7, using our earlier example, we now add the version numbers produced by each transaction's commit to those assigned by the sequencer. All three reads assigned version number 4 by the sequencer can also read versions 5 and 6 (i.e., versions produced by other concurrent readers). A write is required to wait until all previous readers are done and the version at the database has been incremented to match its own (e.g., the write assigned version number 7).

```
operation         w w r w r r r w
version assigned  0 1 2 3 4 4 4 7
version produced  1 2 3 4 5 6 7 8
```

Fig. 7. Sequencer-assigned version numbers for a series of transactions and the version number produced at the database proxy after each transaction commits

4.8 Load Balancing

We use the *Shortest Execution Length First (SELF)* load balancing algorithm [2]. We measure off-line the execution time of each query on an idle machine. At run-time, the scheduler estimates the load on a replica as the sum of the (a priori measured) execution times of all queries outstanding on that back-end. The scheduler updates the load estimate for each replica with feedback provided by the database proxy in each reply. SELF tries to take into account the widely varying execution times for different query types. We have shown elsewhere [2] that SELF outperforms round-robin and shortest-queue-first algorithms for dynamic content applications.

4.9 Fault Tolerance and Data Availability

The scheduler and the Web server return the result of an update request to the user as soon as a commit response from any database replica has been received. The schedulers then become responsible for coordinating the completion of the updates on the other database back-ends, in the case of a scheduler, sequencer, or a back-end database failure. To meet this goal, the *completion status*, and all the write queries of any update transaction together with the transaction's version numbers, are maintained in a fault-tolerant and highly-available manner at the schedulers. High data availability is achieved by replicating the state among the schedulers. Additional fault tolerance is provided by also logging this information to stable storage. The details of our availability and fault-tolerance protocol are similar to the ones described in our previous paper [3], in which we also demonstrate that these aspects of our solution do not incur significant overhead in terms of computation, memory, or disk accesses.

5 Loose Consistency Models

In the performance evaluation section of this paper, we compare distributed versioning to a number of replication methods that provide looser consistency than 1-copy serializability. These methods and their implementation are introduced next.

5.1 Definition

We describe the three consistency levels specified in Neptune [22], and the types of dynamic content Web sites for which they are suitable. We further extend these consistency models with an additional model designed to incorporate features from the continuous consistency model spectrum [28].

Level 0. Write-anywhere replication. This is the basic lazy consistency scheme that offers no ordering or consistency guarantees. Writes that arrive out-of-order are not reconciled later. This scheme is only applicable to simple services with append-only, commutative or total-updates such as an e-mail service.

Level 1. Ordered writes. Writes are totally ordered at all replicas, but reads can access inconsistent data without any staleness bounds. This scheme is applicable to services which allow partial updates, and where reads can access stale or inconsistent data such as discussion groups.

Level 2. Ordered writes and staleness control for reads. Writes are totally ordered at all replicas, and reads satisfy the following two criteria:

– Each read is serviced by a replica which is at most x seconds stale, where x is a given staleness bound.
– Each read of a particular client perceives all previous writes performed by the same client in the correct order.

This consistency model is suitable for sites that need stronger consistency requirements such as auction sites. For example, a client needs to perceive his previous bids in their correct order and should be guaranteed to see a sufficiently recent maximum bid.

Special. Per interaction or per object consistency This model is application-specific. For each interaction or for each object a consistency model is defined. This approach can be applied to Web sites which have in general strong consistency needs, but where relaxations can be made on a case by case basis, for specific interactions or objects.

5.2 Implementation of Loose Consistency Methods

For Levels 0, 1 and 2, we remove any transaction delimiters and other annotations from the application code. The scheduler and database proxy are modified as follows.

For Level 0, we remove any checks pertaining to in-order delivery of writes at the database proxy. The database proxy still implements conflict resolution, but all writes are handled in the order of their arrival, which may be different at different replicas. No version numbers are used. The scheduler load balances reads among all database replicas.

To implement Level 1, the scheduler obtains version numbers for each write, and the database proxies deliver the writes in version number order, as in distributed versioning.

No version numbers are assigned to reads. The scheduler load balances reads among all database replicas.

In addition to the functionality implemented for Level 0 and 1, for Level 2 the scheduler augments its data structures with a wall-clock timestamp for each database replica and for each table. The appropriate timestamp is set every time a database replica acknowledges execution of a write on a table. The scheduler load balances reads only among the database machines that satisfy the staleness bound for all tables accessed in the query, and, in addition, have finished all writes pertaining to the same client connection. A 30-second staleness bound is used for all applications. As in the original scheme described in Neptune, the staleness bound is loose in the sense that network time between the scheduler and the database proxy is not taken into account.

The implementation of Special consistency models is application-specific, and its implementation is deferred to Section 6 where we discuss application benchmarks.

6 Benchmarks

We provide the basic characteristics of the benchmarks used in this study. More detail can be found in an earlier paper [1].

6.1 TPC-W

The TPC-W benchmark from the Transaction Processing Council (TPC) [26] is a transactional Web benchmark designed to evaluate e-commerce systems. Several interactions are used to simulate the activity of a bookstore.

The database contains eight tables; the most frequently used are order_line, orders and credit_info, which give information about the orders placed, and item and author, which contain information about the books.

We implement the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 interactions, 6 are read-only, while 8 cause the database to be updated. The read-only interactions include access to the home page, listing of new products and best-sellers, requests for product detail, order display, and two interactions involving searches. Update transactions include user registration, updates of the shopping cart, two order-placement transactions, two involving order display, and two for administrative tasks.

The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers, which results in a database of about 4 GB. The inventory images, totaling 1.8 GB, are resident on the Web server.

TPC-W uses three different workload mixes, differing in the ratio of read-only to read-write interactions. The browsing mix contains 95% read-only interactions, the shopping mix 80%, and the ordering mix 50%.

For TPC-W we implement a Special consistency model. This model follows the specification of TPC-W, which allows for some departures from (1-copy) serializability. In more detail, the specification requires that all update interactions respect serializability. Read-only interactions on the retail inventory (i.e., best-sellers, new products, searches and product detail interactions) are allowed to return data that is at most 30 seconds old. Read-only interactions related to a particular customer (i.e., home and order display interactions) are required to return up-to-date data. Even if allowed to read stale data, all queries need to respect the atomicity of the update transactions that they conflict with. We add a number of ad-hoc rules to the scheduler to implement this Special consistency model.

6.2 Auction Site Benchmark

Our auction site benchmark, modeled after eBay [10], implements the core functionality of an auction site: selling, browsing and bidding.

The database contains seven tables: users, items, bids, buy_now, comments, categories and regions. The users and items tables contain information about the users, and items on sale, respectively. Every bid is stored in the bids table, which includes the seller, the bid, and a max_bid value. Items that are directly bought without any auction are stored in a separate buy_now table. To speed up displaying the bid history when browsing an item, some information about the bids such as the maximum bid and the current number of bids is kept with the relevant item in the items table.

Our auction site defines 26 interactions where the main ones are: browsing items by category or region, bidding, buying or selling items, leaving comments on other users and consulting one's own user page (known as myEbay on eBay). Browsing items includes consulting the bid history and the seller's information.

We size our system according to some observations found on the eBay Web site. We have about 33,000 items for sale, distributed among 40 categories and 62 regions, and an average of 10 bids per item. There is an average of 10 bids per item, or 330,000 entries in the bids table. The total size of the database is 1.4GB.

We use a workload mix that includes 15% read-write interactions. This mix is the most representative of an auction site workload according to an earlier study of eBay workloads mentioned in [22].

Although it has been argued that an auction site can be supported by a Level 2 consistency model, as described in Section 5, program modifications are necessary to ensure correct outcome of the auction site with Level 2 consistency. The problem is that the code in several places relies on atomic sequences, which are no longer available in the absence of transactions. For instance, suppose we do not use a transaction for placing a bid. In the transaction for placing a bid, the maximum bid is first read from the item table and then updated if the input bid is acceptable (higher). If reading and updating the maximum bid for an item are not done in a critical section, then if two clients submit bids concurrently, they can both read the same maximum bid value for that item. Assuming that both bids are higher, both will be accepted, and the maximum bid stored in the items table for that item could be wrong (e.g., the lower one of the new bids). Thus, additional code is necessary to verify the correctness of the maximum bid.

6.3 Bulletin Board

Our bulletin board benchmark is modeled after an online news forum like Slashdot [23]. In particular, as in Slashcode, we support discussion threads. A discussion thread is a logical tree, containing a story at its root, and a number of comments for that story, which may be nested.

The main tables in the database are the users, stories, comments, and submissions tables. Stories and comments are maintained in separate new and old tables. In the new stories table we keep the most recent stories with a cut-off of one month. We keep old stories for a period of three years. The new and old comments tables correspond to the new and old stories respectively. The majority of the browsing requests are expected to access the new stories and comments tables, which are much smaller and therefore much more efficiently accessible. Each story submission is initially placed in the submissions table, unless submitted by a moderator.

We have defined ten Web interactions. The main ones are: generate the stories of the day, browse new stories, older stories, or stories by category, show a particular story with different options on filtering comments, search for keywords in story titles, comments and user names, submit a story, add a comment, and review submitted stories and rate comments. None of the interactions contain transactions. For instance, stories are first inserted into the submission table, later moderated, then inserted in their respective tables but not as a part of a multi-query atomic transaction, although each individual update is durable.

We generate the story and comment bodies with words from a given dictionary and lengths between 1KB and 8KB. Short stories and comments are much more common, so we use a Zipf-like distribution for story length [8]. The database contains 3 years of stories and comments with an average of 15 to 25 stories per day and between 20 and 50 comments per story. We emulate 500 K total users, out of which 10% have moderator access privilege. The database size using these parameters is 560 MB.

We use a workload mix which contains 15% story and comment submissions and moderation interactions. This mix corresponds to the maximum posting activity of an active newsgroup, as observed by browsing the Internet for typical breakdowns of URL requests in bulletin board sites [1].

Among the loose consistency models discussed in Section 5, the normal semantics of bulletin boards can be supported by the Level 1 consistency model.

6.4 Client Emulation

We implement a client-browser emulator. A client session is a sequence of interactions for the same client. For each client session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another. The session time and think time are generated from a random distribution with a specified mean.

7 Experimental Environment and Methodology

We study the performance of distributed versioning and compare them to loose consistency models, using measurement for a small number of database replicas and using simulation for larger degrees of replication. We first describe the hardware and software used for the prototype implementation. Next, we describe the simulation methodology.

7.1 Hardware

We use the same hardware for all machines running the emulated-client, schedulers and Web servers, and database engines (or corresponding simulators). Each one of them has an AMD Athlon 800Mhz processor running FreeBSD 4.1.1, 256MB SDRAM, and a 30GB ATA-66 disk drive. They are all connected through 100MBps Ethernet LAN.

7.2 Software

We use three popular open source software packages: the Apache Web server [5], the PHP Web-scripting/application development language [18], and the MySQL database server [16]. Since PHP is implemented as an Apache module, the Web server and application server co-exist on the same machine(s).

We use Apache v.1.3.22 [5] for our Web server, configured with the PHP v.4.0.1 module [18] providing server-side scripting for generating dynamic content. We use MySQL v.4.0.1 [16] with InnoDB transactional extensions as our database server.

7.3 Simulation Methodology

To study the scaling of our distributed versioning techniques on a large number of replicas, we use two configurable cluster simulators: one for the Web/application server front-ends and the other for the database back-ends. We use these front-end and back-end simulators to drive actual execution of the schedulers and the database proxies.

Each simulator models a powerful server of the given type (Web/application server or database) equivalent to running a much larger number of real servers. The Web/application server simulator takes each HTTP request generated by the client emulator and sends the corresponding queries with dummy arguments to one of the schedulers.

The database simulator maintains a separate queue for each simulated database replica. Whenever the simulator receives a query destined for a particular replica, a record is placed on that replica's queue. The record contains the predicted termination time for that query. The termination time is predicted by adding a cost estimate for the query to the current simulation time. The same method of cost estimation is used as described earlier for load balancing (see Section 4.8). These estimates are relatively accurate, because for the applications we consider, the cost of a query is primarily determined by the type of the query, and largely independent of its arguments. The database simulator polls the queues and sends responses when the simulated time reaches the termination time for each query. The database simulator does not model disk accesses, because profiling of real runs indicates that disk accesses are overlapped with computation. This overlap is partly due to the locality in the application, resulting in few disk accesses for reads, and partly due to the lazy commit style for writes.

8 Results

First, we present a performance comparison of distributed versioning with a conservative 2PL algorithm. Next, we compare the performance of distributed versioning with various loose consistency models.

All results are obtained using a cluster with two schedulers (for data availability). For each experiment, we drive the server with increasing numbers of clients (and a sufficient number of Web/application servers) until performance peaks. We report the throughput at that peak.

The simulators were calibrated using data from measurements on the prototype implementation. The simulated throughput numbers are within 12% of the experimental numbers for all workloads.

8.1 Comparison of Distributed Versioning and Conservative 2PL

Figures 8 through 10 compare distributed versioning to conservative 2PL for the TPC-W shopping mix, the TPC-W ordering mix, and the auction site, respectively. We omit the results for the TPC-W browsing and bulletin board workloads because there is no performance difference between distributed versioning and conservative 2PL. The explanation is that these workloads have low conflict rates. These and all further graphs have in the x axis the number of database engines, and in the y axis the throughput in (client) Web interactions per second. The graphs also show two protocols that only use a subset of the features of distributed versioning. *Dversion - EarlyRel* uses version numbers to selectively wait for the right version of tables, but it does not produce a new version of the tables until commit. Vice versa, *DVersion - LateAcq* waits for the correct versions of all tables at the beginning of the transaction, but produces new versions immediately after the transaction's last use of a particular table. The results clearly confirm the discussion in Section 3.6. With increased conflict rates, distributed versioning produces superior throughput compared to conservative 2PL. Both features

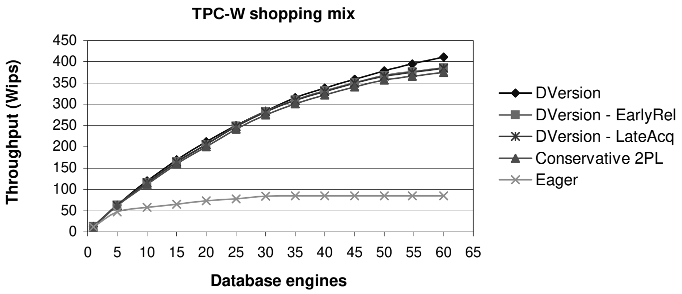

Fig. 8. Comparison of distributed versioning and conservative 2PL for the TPC-W shopping mix

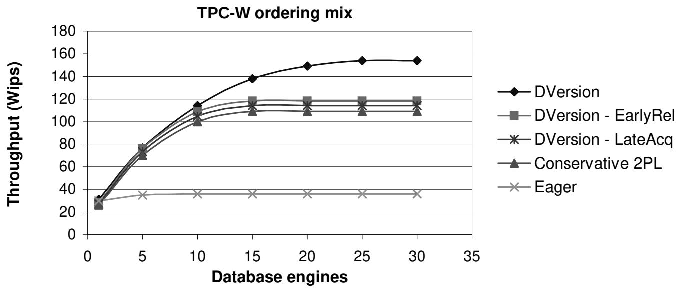

Fig. 9. Comparison of distributed versioning and conservative 2PL for the TPC-W ordering mix

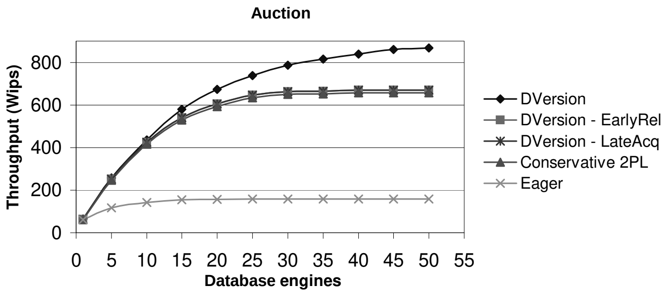

Fig. 10. Comparison of distributed versioning and conservative 2PL for the auction site

of distributed versioning, selective waiting for table versions and early production of new versions, are essential. Without either one of them, improvement over conservative 2PL is minimal.

8.2 Comparison of Distributed Versioning and Loose Consistency Methods

In this section, we investigate the overhead of consistency maintenance for maintaining serializability in distributed versioning. For this purpose, we compare our protocol with all other protocols for all levels of consistency including specialized, and looser than required for each of the three applications. This allows us to detect the overhead of various parts of our solution. For example, we can detect the overhead of in-order delivery for writes or the success of our conflict avoidance and reduction techniques by comparison to the upper bound obtained by assuming that these overheads (for ordering writes or resolving read-write conflicts) do not exist.

Figures 11 through 15 show a comparison of the throughput between distributed versioning protocol (DVersion) and various lazy replication methods. As a baseline, we also include the Eager protocol. These figures allow us to draw the following

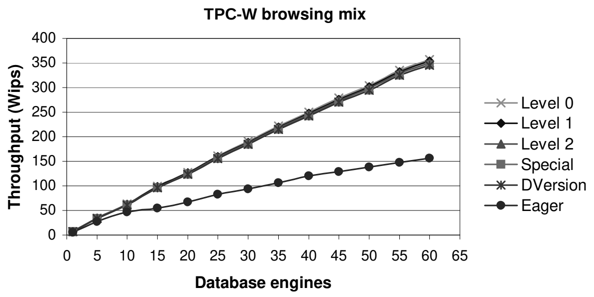

Fig. 11. Comparison of all consistency levels for the TPC-W browsing mix. Special is the specialized consistency level

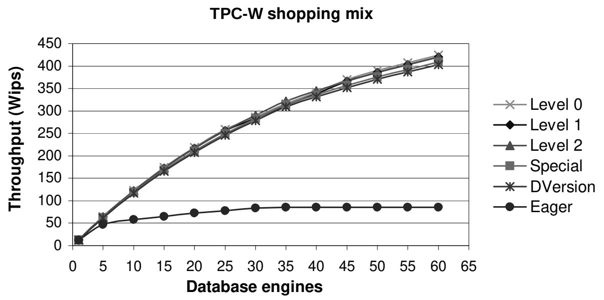

Fig. 12. Comparison of all consistency levels for the TPC-W shopping mix. Special is the specialized consistency level

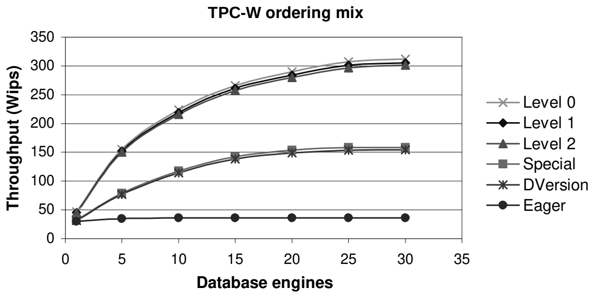

Fig. 13. Comparison of all consistency levels for TPC-W, the ordering mix. Special is the specialized consistency level

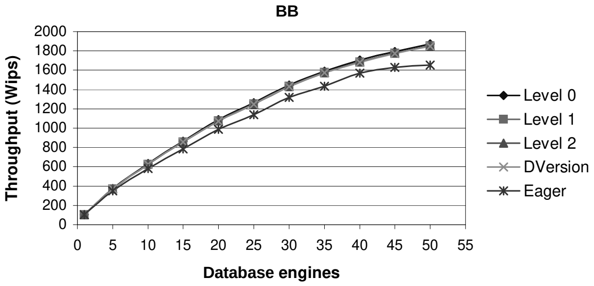

Fig. 14. Comparison of all consistency levels for the bulletin board. Level 1 is the specialized consistency level

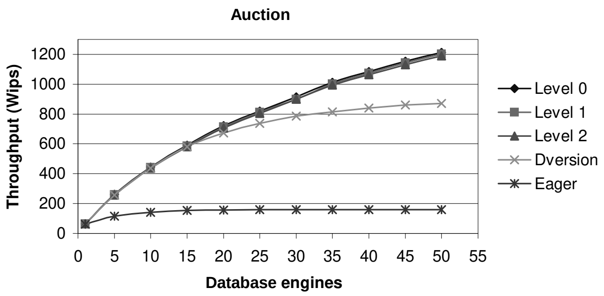

Fig. 15. Comparison of all consistency levels for the auction site. Level 2 is the specialized consistency level

conclusions. First, for all applications, the differences between Levels 0, 1, and 2 are negligible. Second, for the workloads with low conflict rates (i.e., TPC-W browsing and bulletin board), there is no difference between any of the protocols. Third, as the conflict rate increases, there is a growing difference between Levels 0, 1, and 2, on one hand, and DVersion on the other. For the largest simulated configuration, these differences are 5%, 25% and 50%, for the TPC-W shopping mix, the auction site, and the TPC-W ordering mix, respectively. Fourth, the differences between the Special consistency model and DVersion are small for all workloads of TPC-W. Finally, for the bulletin board, which has no transactional requirements, the overhead of enforcing 1-copy-serializability is minimal.

In a cluster, messages usually arrive in the order that they are sent, and the delay in message delivery is low. Furthermore, interactions between the client and the database are such that at least one database replica must have completed the previous request before the next one is issued. A conflict-aware scheduler directs this next request precisely to that replica. These observations explain the small differences in performance between Levels 0, 1 and 2 for all applications, and between DVersion and Special for TPC-W. Loose consistency models show a benefit, instead, when transactional atomicity is removed, and hence the cost of waiting for read-write conflicts is alleviated. As the number of such conflicts increases, the benefit of loose consistency models grows. Among our applications, this is the case for the TPC-W shopping mix, the auction site, and the TPC-W ordering mix. These results should be viewed with the caveat that, as explained in Section 6, these looser consistency models do not, by themselves, provide the right semantics for these applications. Additional programming effort is required to achieve the right semantics, possibly entailing additional execution time overhead.

9 Related Work

Concurrency control protocols based on multiple versions have been discussed [7] and implemented in real systems [19] to increase concurrency while maintaining serializability in stand-alone database systems. More recently, multiversion ordering [6, 12, 20, 21] has been used and optimized for distributed database systems as well. Most of these systems use transaction aborts to resolve serialization inconsistencies. Some systems targeted at advanced database applications such as computer-aided design and collaborative software development environments [6] use pre-declared write-sets to determine if a schedule conflicting at the object level can be serialized, thus avoiding transaction aborts.

Such systems maintain a history of old versions at each distributed location and need a special scheme for reducing version space consumption and version access time [9], or limiting the number of versions stored [12]. Furthermore, if replication is used at all in these distributed systems, the goal is to increase the availability of a particular version [20]. In contrast, in our versioning concurrency control algorithm we do not maintain old copies of items, all modifications are made in place. The goal of our extra-database algorithm is to allow us to choose the correct version among the different versions of a table which occur naturally due to asynchronous replication. On the other hand, multiversion systems have the advantage that the execution of read-only transactions can be made more efficient by completely decoupling their execution from update transactions [12, 20].

Optimistic replication of data [25] has been used in disconnected mobile devices allowing such devices to perform local updates. In this case multiple versions of the same item can arise with the need of serializing potentially conflicting updates by disconnected clients on all replicas.

Eager replication based on group communication has been proposed [24, 27], and optimized [14, 15, 17], to tackle the same problem: providing serializability and scaling at the same time. Their approach is implemented inside the database layer. Each replica functions independently. During the transaction execution, the local database acquires the proper locks for the read and write accesses on its own machine, performs the operations, then sends the other replicas the write-sets through group communication. Conflicts are solved by each replica locally. This implies the need to abort transactions when a write-set coming in conflicts with the local transaction. Approaches that eliminate the need for transaction aborts exist [24, 14], but they either imply relaxing the serializability requirement [14], or introducing an acknowledgment coordination phase between all replicas, at the end of the transaction [24]. These approaches differ from ours in that databases are considered independent (e.g. distributed on a wide area network), where clients have little choice other than executing transactions locally. In our work, the state of the database replicas, the consistency level between them, and the data that the user sees are all controlled by the scheduler(s). We use an extra-database approach where no database modifications are necessary.

Recent work [22, 28] avoids paying the price of serializability for applications that don't need it by providing specialized loose consistency models. Neptune [22] adopts a primary-copy approach to providing consistency in a partitioned service cluster. However, their scalability study is limited to Web applications with loose consistency where scaling is easier to achieve. They do not address e-commerce workloads or other Web applications with relatively strong consistency requirements.

10 Conclusions

The conventional wisdom has been that in replicated databases one could have either 1-copy serializability or scalability, but not both. As a result, looser consistency models have been developed that allow better scalability. In this paper we have demonstrated that for clusters of databases that serve as the back-end of dynamic content Web sites, 1-copy serializability and scalability can go hand-in-hand. This allows for uniform handling of applications and for the use of familiar programming abstractions, such as the use of transactions.

In order to achieve these results, we use a novel technique, called distributed versioning. This technique combines a new concurrency control algorithm based on explicit versions and conflict-aware scheduling to achieve scalability. We have demonstrated that distributed versioning provides much better performance than earlier techniques for providing 1-copy serializability, including eager protocols, and our own earlier work based on conservative two-phase locking. Furthermore, we have compared distributed versioning to various replication methods which only provide loose consistency guarantees. We find that for all our applications, except those with very high conflict rates, the performance of distributed versioning equals or approaches that of looser consistency models.

References

1. C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *5th IEEE Workshop on Workload Characterization*, November 2002.
2. C. Amza, A. Cox, and W. Zwaenepoel. Scaling and availability for dynamic content web sites. Technical Report TR02-395, Rice University, 2002.
3. Cristiana Amza, Alan Cox, and Willy Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, March 2003.
4. Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data: June*, pages 484–495, 1998.

5. The Apache Software Foundation. http://www.apache.org/.
6. N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. In *ACM Computing Surveys*, volume 23, pages 269–317, Sept 1991.
7. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.
8. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE Infocom Conference*, 1999.
9. Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient management of multiversion documents by object referencing. In *The VLDB Journal*, pages 291–300, 2001.
10. On-line auctions at eBay. http://ebay.com.
11. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4–6, 1996*, pages 173–182, 1996.
12. H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Asynchronous Version Advancement in a Distributed Three-Version Database. In *Proceedings of the 14th International Conference on Data Engineering*, 1998.
13. P. Keleher. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, May 1999.
14. Bettina Kemme and Gustavo Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. In *ACM Transactions on Data Base Systems*, volume 25, pages 333–379, September 2000.
15. Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.
16. MySQL. http://www.mysql.com.
17. M. Patiqo-Martinez, R. Jiminez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *International Symposium on Distributed Computing*, pages 315–329, October 2000.
18. PHP Hypertext Preprocessor. http://www.php.net.
19. Postgres. http://www.postgresql.org/docs.
20. O. T. Satyanarayanan and Divyakant Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. In *TKDE*, volume 5, pages 859–871, 1993.
21. D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction Chopping: Algorithms and Performance Studies. In *ACM Transactions on Data Base Systems*, volume 20, pages 325–363, Sept 1995.
22. Kai Shen, Tao Yang, Lingkun Chu, JoAnne L. Holliday, Doug Kuschner, and Huican Zhu. Neptune: Scalable Replica Management and Programming Support for Cluster-based Network Services. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*, pages 207–216, March 2001.
23. Slashdot: News for Nerds. Stuff that Matters. http://slashdot.org.
24. I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, Amsterdam, The Netherlands, May 1998.
25. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings 15th Symposium on Operating Systems Principles*, pages 172–183, December 1995.
26. Transaction Processing Council. http://www.tpc.org/.
27. M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, October 2000.
28. Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.