Scalable Database Replication through Dynamic Multiversioning

Kaloian Manassiev*, Cristiana Amza†

**Department of Computer Science, University of Toronto, Canada*
†*Department of Electrical and Computer Engineering, University of Toronto, Canada*
kaloianm@cs.toronto.edu, amza@eecg.toronto.edu

Abstract

We scale the database back-end in dynamic content cluster servers by distributing read-only transactions on a set of lightweight database replicas while maintaining 1-copy-serializability. This is contrary to conventional wisdom in replicated databases which says that one could have either 1-copy serializability or scalability, but not both.

The key to scaling is a novel integrated fine-grained concurrency control and data replication algorithm called Dynamic Multiversioning that provides fine-grained distributed concurrency control at the level of a memory page across a database cluster. We exploit the different distributed data versions that naturally come about as a result of asynchronous data replication in order to increase concurrency by running conflicting transactions in parallel on different replicas.

At the same time, the serialization order is determined using fine-grained concurrency control at a master database and enforced through a version-aware scheduling technique. Our technique does not put any crucial data in the scheduler, which permits easy reconfiguration, without loss of data, in the case of single-node failures of any node in the system.

Our measurements show near-linear scaling up to 8 databases for the browsing, shopping and even for the write-heavy ordering workload of the industry-standard e-commerce TPC-W benchmark.

1 Introduction

This paper studies scaling the database tier in dynamic content web sites by using dynamic in-memory data replication on clusters, while maintaining strong consistency (i.e., 1-copy serializability).

Dynamic content sites commonly use a three-tier architecture, consisting of a front-end web server, an application server implementing the business logic of the site, and a back-end database (see Figure 1). The (dynamic) content of the site is stored in the database. A number of scripts provide access to that content. The client sends an HTTP request to the web server containing the URL of the script and some parameters. The Web server executes the script, which issues SQL queries, one at a time, to the database and formats the results as an HTML page. This page is then returned to the client as an HTTP response.

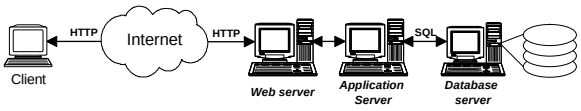

Figure 1: Common Architecture for Dynamic Content Sites

In many current applications (e.g., e-commerce, auction-sites), the application scripts are quite simple to interpret in comparison to most of the database queries that they generate, resulting in the database becoming a bottleneck [5]. In this paper we use a set of lightweight in-memory database engines as "bricks" for building a low-cost, easily scalable and highly available replicated database tier.

Our study draws on recently proposed content-aware scheduling techniques in replicated database clusters [20] and in particular on our own previous work [7, 8] on asynchronous replication with conflict-aware scheduling. We have previously shown that intelligent query scheduling on a replicated database cluster can bring substantial scaling benefits [7, 8]. On the other hand, our previous approaches have treated the databases as black boxes, hence have been limited to using coarse-grained per-table concurrency control at the scheduler for maintaining replica consistency.

In this paper, we introduce Dynamic Multiversioning, a novel scheduler-based scaling scheme for database server clusters integrating fine-grained concurrency control and data replication.

We leverage the availability of several data copies in our in-memory replicated database cluster to implement a multiversioning concurrency control exploiting

the presence of the distributed versions. The replication scheme within our intermediate database tier is a hybrid between eager and lazy replication offering their combined benefits of both scaling and strong consistency at the same time. We categorize incoming queries into conflict classes [16] based on the tables they write, and assign each conflict class on a separate replica, which we call a *master* for this class. Modifications always occur on a master and are broadcast to the remaining replicas as a pre-commit action as in an eager scheme. On the other hand, each slave replica delays the application of modifications thus not delaying the committing master databases in order to provide scaling. As opposed to a classic lazy scheme where the user may see inconsistencies as a result, in our scheme, the scheduler enforces 1-copy-serializability by assigning the most recent version produced by the master to each read-only transaction. The appropriate version for each individual item is then created dynamically at a particular replica when needed by a read-only transaction in-progress at that replica.

The system automatically detects data races created by different read-only transactions attempting to read conflicting versions of the same item. In the common case, the scheduler sends any two such transactions on different replicas, where each creates the item versions it needs and the two transactions can execute in parallel. In addition, update queries belonging to different conflict classes may also proceed in parallel on their respective master nodes.

In summary the overall system has three desirable properties: i) it scales by distributing reads and writes to multiple replicas without restricting concurrency at each replica in the common case, ii) it provides consistency semantics identical to a 1-copy database, iii) it provides data availability through simple reconfiguration in case of failures by keeping very little information outside of the databases.

Our data replication scheme is currently implemented inside an in-memory database tier for maximum flexibility and efficiency. Persistency is, how- ever, ensured by flushing the master database updates to disk synchronously upon commit. A novel aspect of our replicated database tier implementation is its construction from two existing libraries: the Vista library that provides very fast single-machine transactions [15], and the MySQL in-memory ”heap table” code that provides a very simple and efficient SQL database engine without transactional properties. We use these codes as building blocks for our fully transactional in-memory database tier because they are reported to have reasonably good performance and are widely available and used. Following this ”software components” philosophy has significantly reduced the coding effort involved. On the other hand, our replication algorithms are general enough and we believe that our scaling results extend to any database code including replication for traditional on-disk databases.

In our evaluation we use the three workload mixes of the industry standard TPC-W e-commerce benchmark [4]. The workload mixes have an increasing fraction of writes: browsing (5%), shopping (20%) and ordering (50configurations of the in-memory tier (with and without indexes) to study different trade-offs between individual database speed versus scalability over several database replicas. Finally, we compare the throughput obtained through Dynamic multiversioning on a cluster of one master and eight slave replicas with the throughput obtained with a fine-tuned stand-alone database with multiversioning concurrency control (MySQL with InnoDB tables).

We have implemented the TPC-W web site using three popular open source software packages: the Apache Web server [9], the PHP Web-scripting/application development language [18], and the MySQL database server [2]. Our results are as follows:

1. Dynamic Multiversioning provides close to linear scaling for the browsing and shopping workloads in all configurations and for the ordering mix in one of the configurations.

2. The scaling limitation is due to the inherent limit of the conflict class master databases saturating with writes, which is the ultimate theoretical scaling limit of any replication scheme.

3. Using our in-memory transaction tier, we are able to scale the InnoDB on-disk database back-end by a factor of 9.7, 11.5 and 4.18 for the browsing, shopping and ordering mixes respectively.

The rest of this paper is organized as follows. Section 2 introduces our Dynamic Multiversioning scaling solution. Section 3 through 4 describe our prototype implementation and the fault tolerance and high availability aspects of our solution. Sections 5 and 6 describe our results. Section 7 discusses related work. Section 8 provides our conclusions.

2 Dynamic Multiversioning

2.1 Overview

The goal of Dynamic Multiversioning is to scale the database tier through a novel distributed concurrency control mechanism that integrates per-page fine-grained concurrency control, consistent replication and version-aware scheduling.

The idea of isolating the execution of conflicting update and read-only transactions through multiversioning concurrency control is not new [10]. Existing standalone databases supporting multiversioning (e.g., Oracle) pay the price of maintaining multiple physical data copies for each database item and garbage collecting old copies.

Instead, we take advantage of the availability of distributed replicas in a database cluster to run each read-only transaction on a consistent snapshot created dynamically at a particular replica for the pages in its read set. In addition, we utilize the presence of transactions with disjoint write sets in typical e-commerce applications, in order to enable non-conflicting update transactions to run in parallel, thus exploiting the available hardware most optimally.

We augment a simple in-memory database with a replication module implementing a scheme that is i) eager by propagating modifications from a master database that determines the serialization order to a set of slave databases before the commit point. ii) lazy by delaying the application of modifications on slave replicas and creating item versions on-demand as needed for each read-only transaction.

In more detail, our fine-grained distributed multiversioning scheme works as follows:

A scheduler distributes requests on the in-memory database cluster as shown in Figure 2. We require that each incoming request is preceded by its access type, e.g. read-only or update. The scheduler is pre-configured with the types of update transactions used by the application. It uses that information to categorize the incoming requests into conflict classes [16], based on the set of tables that they ask for. Upon detection of a new conflict class, the scheduler assigns it to a separate master database (provided that one is available) and starts sending all queries belonging to the conflict class to its respective node. Read-only transactions are distributed in a load-balancing fashion among the available database replicas as shown in Figure 2.

The master databases decide the order of execution of write transactions based on their internal two-phase-locking per-page concurrency control. Due to the fact that the different conflict classes are disjoint, there is no need for inter-master syncronization, which permits a fully parallel execution of updates.

Each update transaction committing on a master node produces a new consistent state of the database. Each database state is represented by a version vector with a single integer entry for each table of the application working set. Upon transaction commit, each master database synchronously flushes its modifications to disk and to the remaining replicas. The the master communicates the most recent version vector produced locally to

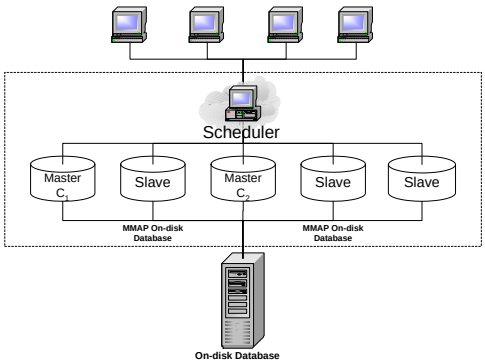

Figure 2: System design.

the scheduler when confirming the commit of each update transaction. The scheduler merges incoming version vectors, tags each read query with the version vector that it is supposed to read (i.e., the most recent version produced by all of the masters) and sends it to a replica. Each read-only transaction applies all fine-grained modifications received from a conflict-class master, for each of the items it is accessing, thus dynamically creating a consistent snapshot of the database version it is supposed to read.

Versions for each item are thus created dynamically when needed by a read-only transaction in progress at a particular replica. Specifically, the replica applies all local fine-grained updates received from a master on the necessary items up to and including the version vector that the read-only transaction has been tagged with. Different read only transactions with disjoint read sets can thus run concurrently at the same replica even if they require snapshots of their items belonging to different database versions. Conversely, if two read-only transactions need two different versions of the same item(s), respectively they can only execute in parallel if sent to different database replicas.

2.2 Version-Aware Scheduling

Dynamic Versioning guarantees that each read-only transaction executing on a slave database reads the latest data version as if running on a single database system. The scheduler enforces the serialization order by tagging each read-only transaction with the last version vector communicated by the master replicas and sending it to execute on a database replica.

The execution of read-only transactions is isolated from any concurrent updates executing on the master replica, whose write set intersects with the read set of the read-only transactions. This means that a read-only transaction will not apply and will not see modifications on items that were written later than the version it was as-

signed. The first condition is guaranteed by the fact that flushes occur only at commit time and the second one is provided by the internal two-phase locking concurrency control of the database.

Assuming that the read-only transaction has been tagged with version $V(v_1,\ldots,v_n)$ by the scheduler, the slave replica creates the appropriate version on-the-fly for all items read by the transaction. Specifically, the slave replica applies all local fine-grained updates received from a master only on the necessary items up to and including version $V(v_1,\ldots,v_n)$.

We employ a scheduler that is aware of the data versions that exist or are about to be created at each replica and sends a read-only transaction to execute on a replica, where the chance of conflicts or waiting is the smallest. Our current heuristic selects a replica from the set of databases where read-only transactions with the same version number as the one to be scheduled are currenly executing if such databases exist. Otherwise it selects any database. The scheduler load balances across the database candidates thus selected using a shortest execution length estimate as in the SELF algorithm we introduced in our previous work [7].

In the case of imperfect scheduler knowledge about a transaction's working set or insufficient replicas, a read-only transaction may need to wait for other read-only transactions using a previous version of an item, or for update transactions writing the item to finish. In rare cases, a read-only transaction T1 may need to be aborted if another read-only transaction T2 first upgrades a shared item to a version higher than that required by T1.

3 Implementation

3.1 Overview

Figure 2 shows the architecture of our caching tier. Its components are presented within the dotted box. We have the *scheduler* node, several *master* databases to execute update transactions and multiple *read-only* replicas which execute the typically heavier read-only workload.

The *scheduler* serves as a communication point for clients of the replicated database. Its functions are to categorize incoming transactions into either update or read-only ones, to determine conflict classes, to send updates to their respective masters and reads to any of the replicas, and to dispatch responses from the databases back to the client. The client applications open connections to the scheduler and start submitting SQL transactions. Any multi-statement transaction should start with a `BEGIN TRANSACTION` and should end with a `COMMIT` statement. Transactions consisting of a single statement are implicitly committed. In addition, in order to make full use of the distributed versioning algorithm, the transaction should be preceded with an information stating whether it is read-only or update and the tables it accesses.

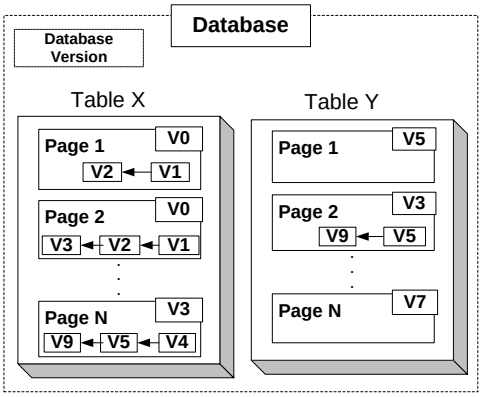

Figure 3: Per-page versioning.

3.2 Implementation Details

For simplicity of implementation, our database replication technique is implemented inside a separate in-memory database tier and not within the on-disk database back-end tier itself. Based on the standard MySQL `HEAP` table we have added a separate table type called `REPLICATED_HEAP` to MySQL. Replication is implemented at the level of physical memory modifications performed by the MySQL storage manager. Since MySQL heap tables are not transactional and do not maintain either an undo or redo log, we capture the modifications transparently using traditional virtual memory page protection violation, twinning and diffing [6] instead. The unit of transactional concurrency control is the memory page as well.

3.3 Integrated Fine-grained Concurrency and Replication

During the execution of a transaction, the database engine keeps track of the pages that it modifies on behalf of that transaction. We use per-page two-phase locking concurrency control between conflicting update transactions. At the end of the update transaction, write-sets created by a master are encapsulated in *diffs*, which are word level run-length encodings of the modifications performed by its data engine on a per-page basis. The master nodes broadcast the fine-grained modifications produced locally to all other replicas, as a pre-commit action for each update transaction. Once the modifications are received at a particular replica, it sends an acknowledgement immediately to the master. Upon receiving acknowledge-

```
 0: MasterPreCommit(PageSet[] PS, TableSet[] TS):
 1:     WriteSet[] WS = CreateWriteSet(PS)
 2:     For Each Table T in TS Do:
 3:         DBVersion[T.ID] ++
 4:     NewVersion = CopyOf(DBVersion)
 5:     For Each Replica R Do:
 6:         SendUpdate(R, WS, NewDBVersion)
 7:         WaitForAcknowledgement(R)
 8: End.
 9:
10: MasterCommit(NewDBVersion):
11:     Scheduler.Send(CommitACK, NewDBVersion)
12: End.
13:
14: MasterPostCommit(PageSet[] PS):
15:     For Each Page P in PS Do:
16:         P.ReleaseLock()
17: End.
```

Figure 4: Master node pre, post and commit actions.

ments from all remote replicas, the master commits its current transaction. At each remote replica, the application of the modifications is however delayed until those modifications are actually needed by a subsequent transaction.

In order to provide consistency and globally serializable transaction execution order, we augment the database engine with a *version vector* field. Each entry in this version vector corresponds to a single table from the database's schema. As a pre-commit action of each committing transaction, master nodes increment those fields of the version vector, which correspond to the set of tables which the transaction modified. Thus, each entry in the version vector specifies the number of successfully committed update transactions that have made changes to the particular table. The first part of Figure 4 shows the pseudo-code for pre-committing a transaction on the master node.

The parameter `PS` (from Page Set) is a data structure maintaining all the pages that the transaction modified during its execution. `TS` has analogous purpose.

At pre-commit, the master generates the *write-set* message with the diffs for each modified page. It then increments the database version vector fields corresponding to each table accessed and sends the write-set and the version that it would turn the database into to all other replicas in a *diff flush* message.

The increment of `DBVersion` in lines 2-3 is implemented as an atomic operation. After the pre-commit step completes, the master node reports back to the scheduler that the transaction has successfully committed and piggybacks the new database version vector on the reply packet. Finally, all page locks are released and the master commits the transaction locally.

The scheduler merges all piggybacked version vectors and uses them to tag all subsequent read-only transactions. The merging of version vectors is simple and involves taking the maximal value of the vector entry belonging to each table. For example, consider two update transactions U1 and U2 belonging to different conflict classes C1 and C2. U1 updates tables with indexes 0 and 1, and U2 updates table with index 2. Suppose that initially, the database version vector is $V_0(x_0, x_1, x_2)$. After U1 commits, the master for C1 will send to the scheduler a version vector $V_{U1}(x_0+1, x_1+1, x_2)$. Similarly, when committing U2, the master for C2 will send a version vector $V_{U2}(x_0, x_1, x_2+1)$ to the scheduler. Suppose that the scheduler first processes U1. This will turn the global database version into $V_1 = V_{U1}$. When the scheduler processes U2, it will turn the global database version vector to $V_2(x_0+1, x_1+1, x_2+1)$.

3.4 Page Organization

This section discusses the handling of transaction flushes as they get delivered at the recipient replicas.

In order to prevent flushes from interfering with read-only transactions which might currently be running on replicas, each page is augmented with three fields: `TableID`, `VersionID` and `DiffQueue`. This is depicted in Figure 3.

The `VersionID` designates the database version that the page currently corresponds to and `TableID` is used as an index in the transaction version vector when comparisons are performed. The `DiffQueue` is a linked list with physical modifications to the page data (diffs), which correspond to the evolution of the page, but have not yet been requested and applied. The `DiffQueue` is ordered and is applied in an increasing order of its version ID, because the modifications are incremental.

For example, consider Page 1 of Table X in Figure 3. If diff entry `V1` is applied to the page, the page will correspond to all versions of the database, having `V1` in the entry for Table X of their version vectors. Since `V2` was produced after `V1`, it should only be applied after `V1` is applied, which will turn the page into `V2`.

Upon reception of a write-set packet from a master node, replicas unpack it into per-page diffs, and enqueue the diffs to the corresponding pages' `DiffQueue`s. Nothing is applied to the pages yet. After that step, the replica node reports to the sending master that it has successfully received the diffs. Figure 5 depicts this process. The per-page mutex is required to ensure that no transaction running on the node is currently accessing the diff queue, but the same operation may be implemented using a lock-free technique.

```
0:  OnFlush(FromID, WriteSet[] WS, VVector):
1:      For Each Page Diff S in WS Do:
2:          DiffEntry D
3:          D.Data = S
4.          D.Version = VVector[Pages[S.PageID].TableID]
5:          Lock(Pages[S.PageID].Mutex)
6:          Pages[S.PageID].DiffQueue.Enqueue(D)
7:          Unlock(Pages[S.PageID].Mutex)
8:      SendAcknowledgement(FromID, MyNodeID)
9:  End.
```

Figure 5: Algorithm for handling the reception of transaction flushes.

3.5 Read-Only Transactions

For the sake of consistency, new queries need always obtain the latest version of the database and should not access mixed data corresponding to different database versions. For that reason, when the scheduler receives a request for new transaction, it tags it with the latest database version vector known at that time. For multi-statement transactions, the tagging ensures that all statements carry the same version vector. The tagged transaction is then forwarded to a replica.

The scheduler does the forwarding in a load-balancing fashion, striving to accommodate as many transactions as possible having the same version on the same database, and trying to mix read-only and update transactions with the same set of accessed items on the same replica. We expect that this arrangement will decrease the level of aborts due to version conflicts, will decrease the lock waiting time, and will manage to load the replicas evenly.

Figure 6 shows the algorithm that transactions go through when accessing a page.

When the transaction needs to access a page, it first checks whether the version of the page corresponds to the version that the transaction expects. If that is the case, the `RefCount` field of the page is incremented to prevent others from updating the version before the transaction has committed. In addition, if the transaction is update, it will hold the page locked, so as to provide serializability.

If the expected version is greater than the current version of the page, the transaction has several options, shown in lines 6 to 16 of the algorithm in Figure 6. The condition on line 6 specifies whether the transaction may proceed with upgrading the page version. Clearly, if there are uncommitted read-only transaction that are still accessing the page, this should not be possible. The conditions in lines 7 through 9 are an optimization. Even if the version of the page is obsolete, if this page has not seen any modifications that the newer transaction should see, it may safely proceed reading the page. Otherwise, the newer transaction should wait until the reference count drops to zero, and then retry accessing to the page.

```
 0: AccessPage(Tx, Page, AccessMode):
 1:     Lock(Page.Mutex)
 2:     Version = Tx.VVector[Page.TableID]
 3:     If (Version == Page.Version):
 4:         Page.RefCount++
 5:     Else If (Version > Page.Version):
 6:         If (Page.RefCount > 0 AND
 7:             NOT DiffQueue.Empty() AND
 8:             DiffQueue.Head().Version <=
 9:                             Version):
10:             Wait Until Page.RefCount == 0
11:             GoTo Line 1
12:         Page.RefCount++
13:         For Each Diff in Page.DiffQueue:
14:             If (Diff.Version <= Version):
15:                 Page.ApplyDiff(Diff)
16:                 Page.Version = Diff.Version
17:     Else: // A version conflict occurred
18:         Unlock(Page.Mutex)
19:         Return ERROR_ABORT
20:     if (AccessMode == READ_ONLY):
21:         Unlock(Page.Mutex)
22:     Return SUCCESS
23: End.
```

Figure 6: Transaction page access.

The last condition in the algorithm is for the case when a transaction already spent some work reading pages with older versions, and then it hits a page, which has been upgraded to a newer version by another transaction. Since versions of the pages are not kept around after being applied, this causes an irresolvable conflict, and the transaction needs to abort.

We make one optimization in this case, which is not shown in the pseudo-code. If it happens that this is the first page access in the transaction's execution, the consistency of the read data won't be violated if we upgrade the version of the transaction to that of the page. In addition, since update transactions run on master nodes for the conflict class, they will always request the latest version of the page. Thus they will never abort due to version conflicts, albeit they might abort due to a deadlock.

Transactions request an access and go through the algorithm once per page. The `RefCount` field guarantees that no other transaction will increment the page version after a non-committed transaction has read it. Keeping the page latch for update transactions, in turn, guarantees that the two-phase locking protocol will be enforced.

Finally, when the transaction commits, it decrements

the reference count of all pages that it accessed. If for any of these pages, the reference count reaches zero, the corresponding waiting transactions are notified to proceed and retry. Also, for update transactions, the versions of all updated pages are set to be equal to the new version entry of the table they belong to.

New versions are dynamically created, older versions overwritten and diffs discarded at each replica when necessary for the purposes of an on-going transaction. Hence, no garbage collection of old versions is required on the replicas in our system unless a data item is written by a master but never read by the particular replica. These situations could easily be taken care of using a periodic examiner thread.

4 Fault Tolerance

In this section we describe how our novel replication protocol *Dynamic Multiversioning* provides transparent failover besides scalability and consistency. The scheduler node is minimal in functionality, which permits extremely fast reconfiguration in the case of single node fail-stop failure scenarios. We assume a fail-stop failure model where failures of any individual node is detected through missed heartbeat messages or broken connections.

4.1 Scheduler Failure

If the scheduler fails, a new scheduler elected from the remaining nodes takes over. The only necessary action upon starting the new scheduler is to collect the highest version vector present on all the conflict-class master nodes. The new scheduler sends a message to the master databases asking them to abort all uncommitted transactions that were active at the time of failure. After a master executes the request, it replies back with the highest database version number it produced. The new scheduler then merges the version vectors and broadcasts the new global database version vector along with an information of the new topology to all the other nodes in the system.

4.2 Master Failure

In this section we discuss the sequence of events that occurs when a master node of the cluster crashes.

Upon detecting a master failure, the scheduler sends a message to all other replicas requesting them to discard any diffs with version numbers higher than the last version number it has seen. The diffs having that property are guaranteed not to have been applied at that time, because the scheduler has not yet sent a read-only query to request them.

This takes care of cleaning up transactions whose precommit diff flush message may have fully or partially reached a subset of the replicas but the master has not acknowledged the commit of the transaction before failure.

For all other failure cases, reconfiguration is trivial. The replication scheme guarantees that the effects of committed transactions will be available on all the slaves in the system. Hence, reconfiguration simply entails electing a new master from the slave replicas and the system can continue to service requests seamlessly. In the case of master failure during a transaction's execution, the transaction's effects are simply discarded since all transaction modifications are internal to the master node up to the commit point (Figure 4). An error is returned by the scheduler to all clients that had active transactions at the time of failure.

4.3 Read-only Node Failure

The final case that needs to be considered is when a read-only node fails.

The failure of a particular slave node is detected by the scheduler. In this case, the scheduler examines its log of outstanding queries, and for those sent to the failed slave delivers an error message back to the clients, which are free to resubmit. The failed slave is then simply removed from the scheduler tables and a new view of the participating nodes is generated and broadcasted.

5 Evaluation

5.1 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council (TPC) [21] is a transactional web benchmark for e-commerce systems. The benchmark simulates a bookstore.

The database contains eight tables: customer, address, orders, order_line, credit_info, item, author, and country. The most frequently used are order_line, orders and credit_info, which contain information about the orders placed, and item and author, which contain information about the books. The database size is determined by the number of items in the inventory and the size of the customer population.

We use the standard size with 288000 customers and 10000 books. The inventory images, totalling 180 MB reside on the web server.

We implemented the fourteen different interactions specified in the TPC-W benchmark specification. Six of the interactions are read-only, while eight 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, and two interactions involving searches. Update transactions include user registration, updates of the shopping cart, two order-placement transactions, and two for administrative tasks. The frequency of execution of each interaction is specified by the TPC-W benchmark. The most complex read-only interactions are BestSellers, NewProducts and Search by Subject which contain multiple-table joins.

The benchmark has three workload mixes characterized by different ratios of reads to writes. The browsing mix has 95% read-only queries and 5% updates. The shopping workload, which is the one that most closely resembles a real-world scenario, consists of 80% read-only queries and 20% updates. The most update intensive workload is the ordering one, which has 50% updates and 50% read-only queries.

5.2 Experimental Setup

We run our experiments on a cluster of 19 dual AMD Athlon MP 2100+ computers with 512MB of RAM and 1.9GHz CPU. All the machines use the RedHat Fedora Linux operating system. We run the scheduler and each of nine database replicas on a separate machine. We used 10 machines to operate the Apache 1.3.31 web-server, which ran a PHP implementation of the business logic of the TPC-W benchmark.

In order to generate the test workload, we used a client emulator, which emulates client interactions as specified in the TPC-W document.

We first run preliminary experiments on the in-memory database tables to determine baseline factors that influence scaling such as the ratio of read versus write query costs on our experimental system. All further experiments focus on overall system scalability. To determine the peak throughput for each cluster configuration we run a step-function workload, whereby we gradually increased the number of clients from 100 to 1000. We then report the peak throughput in web interactions per second, the standard TPC-W metric, for each configuration. At the beginning of each experiment, the master and the slave databases mmap an on-disk TPC-W database. We run each experiment for a sufficient time such that the data becomes memory resident and we exclude the cache warmup time from the measurements.

6 Experimental Results

6.1 Preliminary Experiments

We run a workload session with only one client on an unmodified single MySQL database with heap tables. The goal of this experiment is to determine the cost of each individual query from the TPC-W benchmark executing

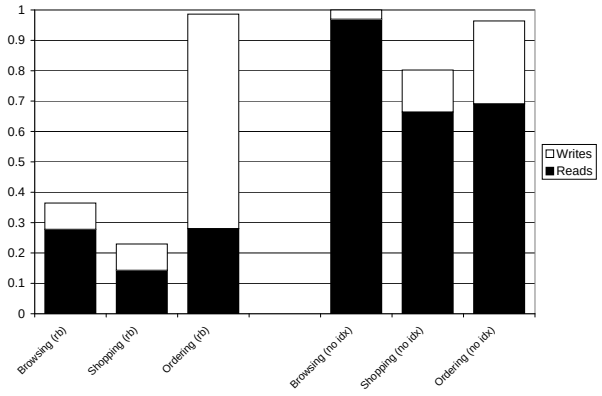

Figure 7: Relative query weights for MySQL heap tables in the two configurations (with and without indexes)

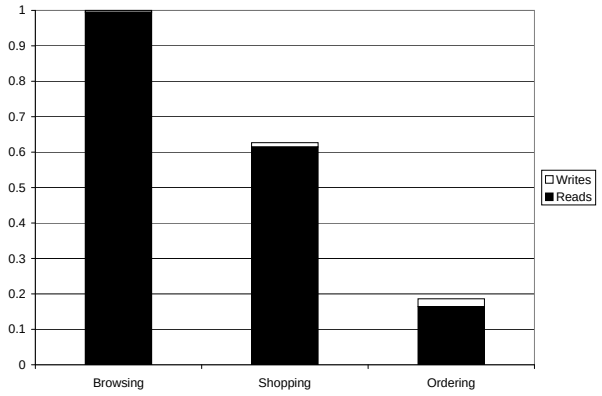

Figure 8: Relative query weights for MySQL InnoDB tables

on our system and to compute the ratio of read versus update query complexity. The first three bars of Figure 7 show the results of this experiment.

These experiments indicate that, with our base in-memory database system, the cost of write transactions is significant. This problem is clearly shown in the ordering mix, where the cost of updates dominates the total cost of read-only queries although the workload mix contains 50% reads including complex multiple table joins such as the BestSeller query. This discrepancy is caused by the high cost of index update operations in MySQL with heap tables.

Since our system builds upon the original MySQL HEAP table engine, we reused its *RB-Tree* [13] index structure. The RB-Tree is a balanced binary search tree, which supports lookup operations with a constant $O(logN)$ cost. Insert and delete operations on the RB-Tree typically cause height imbalances, which then trigger tree rotations. The RB-Tree performs 2 rotations per update on average and the cost of these rotations is significant in the in-memory database system. Thus, update transactions become heavier compared to the aver-

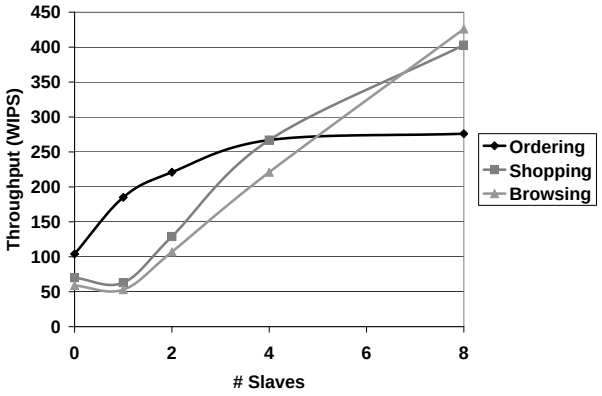

Figure 9: Throughput scaling in the database configuration with indexes for the browsing, shopping and ordering TPC-W mixes.

age read-only query. In addition, it has been shown [12] that the RB-Tree, used as an in-memory database index structure, severely limits concurrency.

Since the read to update ratio affects scaling in replicated databases, we evaluate our protocol with two index configurations. The first index configuration is the same index set as those in the on-disk InnoDB database we used in our prior work [7]. This index configuration is meant to optimize the duration of complex join queries. The second configuration uses no indexes, except for the primary keys. This configuration still conforms to the TPC-W standard specification, since no particular index configurations are required or recommended by the specification.

The second part of Figure 7 shows the read to update ratios for the configuration without the indexes. It can be seen that in this case the cost of read queries dominates.

Figure 8 shows the fraction of read query to write query complexity for our on-disk InnoDB database using a standard *B+ tree* index structure [13]. We can see that the ratios of reads to writes in this case are even higher than the ones we obtain in our configuration without indexes.

6.2 Scalability for the Configuration with Indexes

Figure 9 shows the throughput scaling obtained as we increase the number of slave replicas. We performed measurements with 1, 2, 3, 4 and 8 slaves respectively.

The system exhibits close to linear scaling for the browsing and shopping mixes. The poor scaling of the ordering mix is caused by the fast saturation of the master database with updates due to the fact that writes are more heavyweight than reads on average. This is the ultimate scaling limitation of any replication scheme, since

# of slaves	Browsing	Shopping	Ordering
1	1.54%	1.3%	2.57%
2	1.49%	1.73%	0.5%
4	2.29%	2.63%	0.06%
8	1.6%	0.36%	0.00%

Table 1: Level of aborts due to version inconsistency

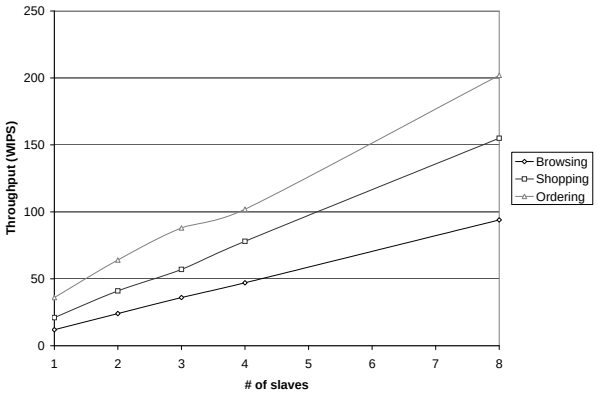

Figure 10: Throughput scaling in the database configuration with no indexes for the browsing, shopping and ordering TPC-W mixes.

the number of writes that the master executes increases with larger database clusters to sustain their higher overall throughput.

Table 1 shows the average number of read-only queries that needed to be restarted during the experiment due to version inconsistency. The numbers are presented as a percentage of the total number of queries that executed during the experiment session.

We see that the level of aborts is very low overall and decreases with the number of slave replicas. The chance that queries seeking different database versions of the same page hit the same replica decreases until there are no aborts at 8 databases.

6.3 Scalability for the Configuration without Indexes

In order to validate the scalability of the dynamic versioning algorithm with a different read to write cost distribution and different application access patterns, we ran the same experiments using the database without indexes. Figure 10 shows the throughput in this configuration.

These results show almost linear scalability for all the workload mixes.

Table 2 lists the abort rates due to version inconsistency in this configuration. Similarly to the indexed database, the level of aborts decreases with the number

# slaves	Browsing	Shopping	Ordering
1	1.86%	3.60%	1.12%
2	1.02%	2.81%	1.34%
3	1.62%	1.78%	1.10%
4	0.62%	0.99%	0.001%
8	0.39%	0.86%	0.10%

Table 2: Level of aborts due to version inconsistency (configuration with no indexes)

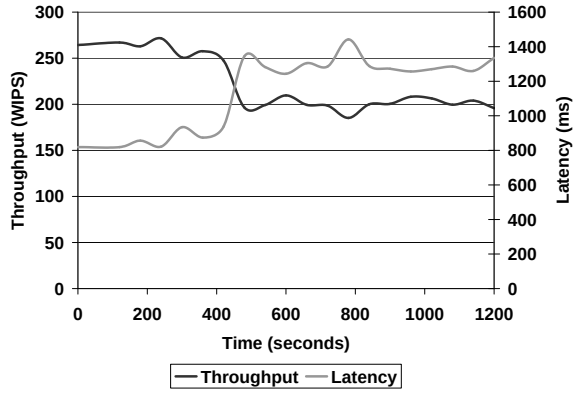

Figure 11: Master failure effect.

of replicas. In the index-less case, however, the level of aborts is very slightly higher. This is because the full table scan for all read queries is the worst case scenario for our algorithm. Since all read-only transactions, and in particular complex queries such as BestSellers and NewProducts, have similar reading sets and access the maximum number of pages, the chance of version conflicts between read-only transactions is high as well. Despite this fact, the abort rates are negligible and are expected to be an upper bound for abort rates with any database size and workload mix.

6.4 Failure Reconfiguration

Figure 11 shows the effect that a failure of a master node in a 5 node configuration has on the throughput. In this scenario, we started a run with the shopping mix and killed the master node at approximately the 360^{th} second. From the chart it can be seen that the drop in throughput is about 21%, which is to be expected in the 5 node configuration. The increase in latency, however, is more pronounced due to the thread scheduling and queuing effects caused by more transactions being sent to nodes.

7 Related Work

A number of solutions exist for replication of relational databases that aim to provide both scaling and strong consistency. They range from industry-established ones, such as the Oracle RAC [3] and IBM DB2 HADR suite [1], to research and open-source prototypes, such as Distributed Versioning [8], C-JDBC [11], Postgres-R [14] and Ganymed [19].

The industry solutions provide both high availability and good scalability, but they are costly and require exotic hardware such as Shared Network Disk. The research prototypes use commodity software and hardware, but they either have limited scaling for moderately heavy write workloads such as the ordering mix of TPC-W [7, 8, 11] due to their use of coarse-grained concurrency control implemented in the scheduler, or sacrifice on fault tolerance and failure transparency by using single points of failure [19].

Reliable broadcast-based systems, such as Postgres-R [14] and Pronto [17] provide the fault-tolerance guarantees of the group communication library that they use. However, the scalability of these solutions under e-commerce workloads is unclear.

8 Conclusions

In this paper we introduce a novel scheduler-based versioning algorithm, *Dynamic Multiversioning*, which preserves strong consistency and at the same time offers high concurrency by exploiting the naturally arising versions across database replicas. In addition, *Dynamic Multiversioning* keeps minimal state at the scheduler thus providing transparent failover and fast reconfiguration in case of failures.

We avoided duplication of database functionality in the scheduler for consistency maintence by integrating the replication process with the database concurrency control.

With the utilization of a per-page diff queue, we avoided copy-on-write overhead associated with systems that use stand-alone database multiversioning offering snapshot isolation. In order to do this, we applied an optimistic approach in the version management, which assumed that the probability of transactions bearing higher database versions causing aborts of lower-version transactions is low. To further decrease that probability and the waiting time, we use a scheduler algorithm, which strives to distribute transactions requesting different version numbers across different nodes.

Finally, we described the fault-tolerance and automatic reconfiguration aspects of our protocol.

While we have used our in-memory replication

scheme to scale an e-commerce database back-end, our system has potential applicability in other application domains such as telecommunication applications that use in-memory databases as storage medium. In the future, we plan to obtain information about the performance and workload requirements of such applications and evaluate the applicability of our system to their support.

References

[1] IBM DB2 High Availability and Disaster Recovery. `http://www.ibm.com/db2/`.

[2] Mysql Database Server. `http://www.mysql.com/`.

[3] Oracle Real Application Clusters 10g. `http://www.oracle.com/technology/products/database/clustering/`.

[4] The Transaction Processing Performance Council. `http://www.tpc.org/`.

[5] 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.

[6] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[7] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, pages 71–84, Mar. 2003.

[8] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *ACM/IFIP/Usenix International Middleware Conference*, June 2003.

[9] The Apache Software Foundation. http://www.apache.org/.

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

[11] E. Cecchet, J. Marguerite, and W. Zwaenepoel. RAIDb: Redundant array of inexpensive databases. In *IEEE/ACM International Symposium on Parallel and Distributed Applications (ISPA'04)*, December 2004.

[12] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 181–190. Morgan Kaufmann Publishers Inc., 2001.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[14] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *The VLDB Journal*, pages 134–143, 2000.

[15] D. Lowell and P. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[16] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *DISC '00: Proceedings of the 14th International Conference on Distributed Computing*, pages 315–329. Springer-Verlag, 2000.

[17] F. Pedone and S. Frolund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 176. IEEE Computer Society, 2000.

[18] PHP Hypertext Preprocessor. http://www.php.net.

[19] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada*, October 18-22 2004.

[20] U. Rhom, K. Bhom, H.-J. Schek, and H. Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *Proceedings of the 28th International Conference on Very Large Databases*, pages 134–143, Aug. 2002.

[21] Transaction Processing Council. http://www.tpc.org/.