Scaling and Continuous Availability in Database Server Clusters through Multiversion Replication

Kaloian Manassiev
Department of Computer Science
University of Toronto
kaloianm@cs.toronto.edu

Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto
amza@eecg.toronto.edu

Abstract

In this paper, we study replication techniques for scaling and continuous operation for a dynamic content server. Our focus is on supporting transparent and fast reconfiguration of its database tier in case of overload or failures. We show that the data persistence aspects can be decoupled from reconfiguration of the database CPU. A lightweight in-memory middleware tier captures the typically heavyweight read-only requests to ensure flexible database CPU scaling and fail-over. At the same time, updates are handled by an on-disk database back-end that is in charge of making them persistent.

Our measurements show instantaneous, seamless reconfiguration in the case of single node failures within the flexible in-memory tier for a web site running the most common, shopping, workload mix of the industry-standard e-commerce TPC-W benchmark. At the same time, a 9-node in-memory tier improves performance during normal operation over a stand-alone InnoDB on-disk database back-end. Throughput scales by factors of 14.6, 17.6 and 6.5 for the browsing, shopping and ordering mixes of the TPC-W benchmark, respectively.

1. Introduction

This paper investigates replication techniques for high-performance, self-configuring database back-end tiers in dynamic content servers. Dynamic content sites currently need to provide very high levels of availability and scalability. On-the-fly reconfiguration may be needed to either adapt to failures or bursts of traffic and should be automatic, fast and transparent. The presence of the database tier in such sites makes fast reconfiguration hard to achieve. Above all, data consistency needs to be ensured during reconfiguration, typically through complex and lengthy recovery procedures. The reconfiguration problem is exacerbated by the need to scale the database tier through asynchronous content replication solutions [5, 10, 20, 13]. Replica asynchrony has been proved absolutely necessary for scaling. On the other hand, because data is not fully consistent on all replicas at all times, asynchronous replication is at odds with fast, transparent reconfiguration. Asynchronous replication techniques thus tend to sacrifice failure transparency and data availability to performance scaling by introducing windows of vulnerability, where effects of committed transactions may be lost. Alternatively, complex failure reconfiguration protocols imply reloading transactional logs from disk and replaying them on a stale replica. Resuming servicing transaction at peak-throughput can take on the order of minutes [12] and possibly more; fail-over times are rarely formally measured and reported.

In this paper, we introduce a solution that combines transparent scaling and split-second reconfiguration. Our key idea is to interpose an *in memory* tier, consisting of lightweight database engines, providing scaling and seamless adaptation to failures on top of a traditional on-disk database back-end. Our middleware tier implements *Dynamic Multiversioning*, a database replication solution allowing both scaling and ease of self-reconfiguration of the overall system. Specifically, our replication solution has the following desirable properties:

1. provides consistency semantics identical to a 1-copy database (i.e., 1-copy serializability), thus making the underlying replication mechanism completely transparent to the user.

2. scales by distributing reads to multiple replicas without restricting concurrency at each replica in the common case.

3. provides data availability through simple and efficient techniques for reconfiguration in case of failures.

In contrast to industry solutions which rely on costly shared network-attached storage configurations [3], our solution

uses only commodity software and hardware. Our focus is on achieving fast reconfiguration for scaling and data availability in the common case of single node failures, while ensuring data persistence in all cases.

An in-memory database tier with asynchronous, but strongly consistent replication offers high speed and scalability during normal operation and inherent agility in reconfiguration during common failure scenarios. An on-disk back-end database with limitted replication offers data reliability for rare failure scenarios, e.g., a power outage. Our research focus on in-memory database tiers is supported by industry trends towards: i) large main-memory sizes for commodity servers, ii) the popularity of database workloads with a high degree of locality [4], such as the most common e-commerce workloads [22], which result in working sets on the order of a few Gigabytes.

Our in-memory replication scheme is asynchronous in order to provide scaling. We have previously shown that the presence of distributed versions of the same page in a transactional memory cluster with asynchronous replication can be exploited to support scaling for generic applications [16]. In this paper, our focus is on supporting both scaling and fast reconfiguration for an in-memory database cluster. In our solution, the fine-grained concurrency control of the database works synergistically with data replication to ensure high performance and ease of reconfiguration. Update transactions always occur on an in-memory master replica, which broadcasts modifications to a set of in-memory slaves. Each master update creates a version number, communicated to a scheduler that distributes requests on the in-memory cluster. The scheduler tags each read-only transaction with the newest version received from the master and sends it to one of the slaves. The appropriate version for each individual data item is then created dynamically and lazily at that slave replica, when needed by an in-progress read-only transaction. The system automatically detects data races created by different read-only transactions attempting to read conflicting versions of the same item. Conflicts and version consistency are detected and enforced at the page level. In the common case, the scheduler sends any two transactions requiring different versions of the same memory page on different replicas, where each creates the page versions it needs and the two transactions can execute in parallel.

We further concentrate on optimizing the fail-over reconfiguration path, defined as integrating a new replica (called a backup) into the active computation to compensate for a fault. The goal is to maintain a constant level of overall performance irrespective of failures. We use two key techniques for fail-over optimization. First, instead of replaying a log on a stale replica, we replicate only the changed pages with newer versions than the backup's page versions from an active slave onto the backup's memory. These pages may have collapsed long chains of modifications to database rows registering high update activity. Thus, selective page replication is expected to be faster on average than modification log replay. Second, we warm up the buffer cache of one or more spare backups during normal operation using one of two alternative schemes: i) we schedule a small fraction of the main read-only workload on a spare backup or ii) we mimic the read access patterns of an active slave on a spare backup to bring the most-heavily accessed data in its buffer cache. With these key techniques, our in-memory tier has the flexibility to incorporate a spare backup after a fault without any noticeable impact on either throughput or latency due to reconfiguration.

Our in-memory replicated database implementation is built 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.

In our evaluation we use the three workload mixes of the industry standard TPC-W e-commerce benchmark [22]. The workload mixes have increasing fraction of update transactions: browsing (5%), shopping (20%) and ordering (50%).

We have implemented the TPC-W web site using three popular open source software packages: the Apache Web server [7], the PHP web-scripting/application development language [19] and the MySQL database server [2]. In our experiments we used MySQL with InnoDB tables as our on-disk database back-ends and a set of up to 9 in-memory databases running our modified version of MySQL heap tables in our lightweight reconfigurable tier.

Our results are as follows:

1. Reconfiguration is instantaneous in case of failures of any in-memory node with no difference in throughput or latency due to fault handling if spare in-memory backups are maintained warm. We found that either servicing less than 1% of the read-only requests in a regular workload at a spare backup or following an active slave's access pattern and touching its most frequently used pages on the backup is sufficient for this purpose. In contrast, with a traditional replication approach with MySQL InnoDB on-disk databases, fail-over time is on the order of minutes.

2. Using our system with up to 9 in-memory replicas as an in-memory transaction processing tier, we are able to improve on the performance of the InnoDB on-disk

database back-end by factors of 14.6, 17.6 and 6.5 for the browsing, shopping and ordering mixes respectively.

The rest of this paper is organized as follows. Section 2 introduces our scaling solution, based on Dynamic Multiversioning, Section 3 describes our prototype implementation and Section 4 presents its fault-tolerance and fast-reconfiguration aspects. Sections 5 and 6 describe our experimental platform, methodology and results. Section 7 discusses related work. Section 8 concludes the paper.

2. Dynamic Multiversioning

2.1. Overview

The goal of Dynamic Multiversioning is to scale the database tier through a 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 [8]. Existing stand-alone 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 update transactions with disjoint working sets in order to enable non-conflicting update transactions to run in parallel, thus exploiting the available hardware optimally.

We augment a simple in-memory database with a replication module implementing a scheme that is i) eager by propagating modifications from a set of master databases 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 1. 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 transactions used by the application and the tables each of them accesses [5]. It uses this information to categorize the incoming requests into conflict classes [18], based on the set of tables that they access. The scheduler assigns a master database to each

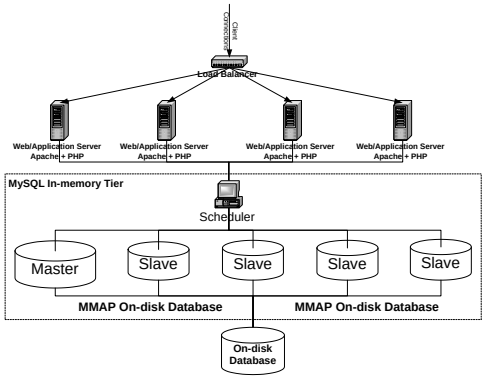


Figure 1. System design.

conflict class. It sends all queries belonging to the update transactions in each conflict class to the respective master node. If no information on conflict classes is available or if conflict classes cannot be statically determined, all update transactions are scheduled on a single node designated as master database. Read-only transactions are distributed among the slave (non-master) database replicas as shown in Figure 1. Read-only transactions can be scheduled on a master replica as well as long as the set of tables they access are not in the master's conflict class. The master database decides the order of execution of write transactions in each conflict class it manages based on its internal two-phase-locking per-page concurrency control. In our scheme, conflict classes are disjoint. Hence, there is no need for inter-master synchronization, 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, called `DBVersion`. Upon transaction commit, each master database flushes its modifications to the remaining replicas. The master communicates the most recent version vector produced locally to the scheduler when confirming the commit of each update transaction. The scheduler merges incoming version vectors and it tags each read-only transaction 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 slave replica. Each read-only transaction applies all fine-grain 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 neces-

sary 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 Multiversioning 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 sent to execute on a database replica with the latest version vector. The latest version vector contains the most recent version communicated by the master replicas on each respective table position.

The execution of read-only transactions is isolated from any concurrent updates executing on the master replicas 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 assigned.

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 each master, only on the necessary items up to and including version $V(v_1, \ldots, v_n)$.

The scheduler selects a replica from the set of databases running read-only transactions with the same version vector as the one to be scheduled if such databases exist. Otherwise, it selects a replica by plain load balancing. In case of insufficient replicas, 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. A read-only transaction T1 may need to be aborted if another read-only transaction T2 upgrades a shared item to a version higher than that required by T1. If T1 has already read a page with its assigned version, then reading a higher version of a different page, would result in version inconsistency. Since we do not keep old versions around, T1 would need to be aborted in this case. However, we expect these situations to be rare.

3. Implementation

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 we add an undo and a redo log. The unit of transactional concurrency control is the memory page. The redo log contains a list of per-page modification encodings. Figure 2 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. At pre-commit, the master generates the *write-set* message with the modifications for each modified page. It then increments the database version and sends the write-set and the version that it would turn the database into to all other replicas. The increment of `DBVersion` vector on line 3 is implemented as an atomic operation to ensure that each committed transaction obtains a unique version vector. After the pre-commit step completes, the master node reports back to the scheduler that the transaction has successfully committed and piggybacks the new *DBVersion* on the reply. Finally, all page locks are released and the master commits the transaction locally. The scheduler records the new version vector and uses it to tag subsequent read-only transactions with the appropriate versions they need to read.

```
1: MasterPreCommit(PageSet[] PS):
2:     WriteSet[] WS = CreateWriteSet(PS)
3:     Increment(DBVerVector, WS)
4:     For Each Replica R Do:
5:         SendUpdate(R, WS, DBVerVector)
6:         WaitForAcknowledgment(R)
7: Return DBVerVector
```

Figure 2. Master node pre-commit actions.

4. Fault Tolerance and Data Availability

In this section, we first describe our reconfiguration techniques in case of master, slave or scheduler node failures. Second, we describe our mechanisms for data persistence and availability of storage in the on-disk back-end database tier. We assume a fail-stop failure model where failures of any individual node are detected through missed heartbeat messages or broken connections.

4.1. Scheduler Failure

The scheduler node is minimal in functionality, which permits extremely fast reconfiguration in the case of single node fail-stop failure scenarios. Since the scheduler's state consists of only the current database version vector, this data can easily be replicated across multiple peer schedulers, which work in parallel. If one scheduler fails and multiple schedulers are already present, one of the peers takes

over. Otherwise, a new scheduler is elected from the remaining 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. This may not be necessary for databases that automatically abort a transaction due to broken connections with their client. After the masters execute the abort request, they reply back with the highest database version number they produced. Then, the new scheduler broadcasts a message to all the other nodes in the system, informing them of the new topology.

4.2. Master Failure

Upon detecting a master failure, one of the schedulers takes charge of recovery. It asks all databases to discard their modification log records, which have version numbers higher than the last version number it has seen from the master. This takes care of cleaning up transactions whose pre-commit modification log flush message may have partially completed at 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 to replaces the failed master replica. Thereafter, the system continues to service requests. In the case of master failure during a transaction's execution, the effects of the transaction are automatically discarded since all transaction modifications are internal to the master node up to the commit point.

4.3. Slave Failure

The failure of any particular slave node is detected by all schedulers. Each scheduler examines its log of outstanding queries, and for those sent to the failed slave, the corresponding transaction is aborted and an error message is returned to the client/application server. The failed slave is then simply removed from the scheduler tables and a new topology is generated.

4.4. Data Migration for Integrating Stale Nodes

In this section, we present the **data migration** algorithm for integrating recovering or other stale replicas. New replicas are always integrated as slave nodes of the system, regardless of their rank prior to failure.

The *reintegrating node* (S_{join}) initially contacts one of the schedulers and obtains the identities of the current masters and an arbitrary slave node. We refer to this slave node as the *support slave*.

In the next step, S_{join} subscribes to the replication list of the masters, obtains the current database version vector $DBVersion$ and starts receiving modification log records. The node stores these new modifications into its local queues, as any other active slave node without applying these modifications to pages. It then requests page updates from its support node indicating the current version it has for each page and the version number that it needs to attain, according to $DBVersion$, as obtained from the master replicas upon joining. The support node then selectively transmits only the pages that changed after the joining node's version to S_{join}.

In order to minimize integration time, all nodes implement a simple fuzzy checkpoint algorithm [8], modified to suit our in-memory database. At regular intervals, each slave starts a checkpointing thread, which iterates the database pages and persists their current contents together with their current version onto local stable storage. A flush of a page and its version number is *atomic*. Dirty pages, which have been written to but not committed, are not included in the flush. However, our checkpointing scheme is flexible and efficient, because it does not require the system to be quiescent during checkpoints. Since our in-memory database normally works with its data pages having different versions at the same time, a checkpoint does not have to be synchronous either across replicas or across the pages checkpointed at each replica. Furthermore, a stale node only receives the changed pages since its last checkpointed version of each page. These pages might have collapsed long chains of modifications to database rows registering high update activity. Hence, our scheme allows for potentially faster reintegration of stale nodes into the computation compared to replaying a log of update transactions.

4.5. Fail-Over Reconfiguration Using Spare Backups

Database fail-over time consists of two phases: **data migration** for bringing the node up to date and the new node's **buffer cache warmup**. An additional phase occurs only in the case of master failure due to the need to abort unacknowledged and partially propagated updates, as described in the master failure scenario above.

The **data migration phase** proceeds as in the algorithm for stale node integration described in the previous section. In the **buffer cache warmup phase**, the backup database needs to warm up its buffer cache and other internal data structures until the state of these in-memory data structures approximates their corresponding state on the failed database replica. The backup database has its in-memory buffer cache only partially warmed up, if at all, because it is

normally not executing any reads for the workload.

In order to shorten or eliminate the buffer cache warmup phase, a set of warm spare backups are maintained for a particular workload for overflow in case of failures (or potentially overload of active replicas). These nodes may be either idle e.g., previously failed nodes that have just recovered or intentionally maintained relatively unused e.g., for power savings or because they may be actively running a different workload. The spare backups subscribe to and receive the regular modification broadcasts from the master replicas just like active slave replicas. In addition, we use two alternative techniques for warming up the spare backup buffer caches during normal operation.

In the first technique, spare backups are assigned a number of read-only transactions with the sole purpose of keeping their buffer caches warm. The number of periodic read-only requests serviced by spare backups is kept at a minimum.

In the second technique, a spare backup does not receive any requests for the workload. Instead, one or more designated slave nodes collect statistics about the access pattern of their resident data set and send the set of page identifiers for pages in their buffer cache to the backup periodically. The backup simply touches the pages so that they are kept swapped into main memory. In this way, the backup's valuable CPU resource can be used to service a different workload. For spare backups running a different workload, care must be taken to avoid interference [21] in the database's buffer cache for the two workloads. This aspect is, however, beyond the scope of this paper.

4.6. Data Persistence and Availability in the Storage Database Tier

We use a back-end on-disk database tier for data persistence in the unlikely case that *all* in-memory replicas fail.

Upon each commit returned by the in-memory master database for an update transaction, the scheduler logs the update queries corresponding to this transaction and, at the same time, sends these as a batch to be executed on one or more on-disk back-end databases. Replication in the case of the on-disk databases is for data persistence and availability of data and not for CPU scaling; only a few (e.g., two) on-disk replicas are needed. Once the update queries have been successfully logged, the scheduler can return the commit response to the application server without waiting for responses from all the on-disk databases. The query logging is performed as a lightweight database insert of the corresponding query strings into a database table [21]. In case of failure, any on-disk database can be brought up to date by replaying the log of missing updates.

5. Evaluation

5.1. TPC-W Benchmark

We evaluate our solution using the TPC-W benchmark from the Transaction Processing Council (TPC) [22], that simulates an on-line bookstore.

The database contains eight tables: customer, address, orders, order_line, credit_info, item, author, and country. We implemented the fourteen different interactions specified in the TPC-W benchmark specification. The most complex read-only interactions are BestSellers, NewProducts and Search by Subject which contain complex joins.

We use the standard size with 288K customers and 100K books, which results in a database size of about 610MB. The memory-resident set of the workload is about 360MB and it consists of the most-frequently accessed sections of the database.

We use the three workload mixes of the TPC-W benchmark: browsing, shopping and ordering. These workloads are characterized by increasing fraction of writes from the browsing mix (5%) to the most commonly used workload, the shopping mix (20%) to the ordering mix (50%).

5.2. Experimental Setup

We run our experiments on a cluster of 19 dual AMD Athlons with 512MB of RAM and 1.9GHz CPU, running the RedHat Fedora Linux operating system. We run the scheduler and each of nine database replicas on separate machines. We use 10 machines to operate the Apache 1.3.31 web-server, which runs a PHP implementation of the business logic of the TPC-W benchmark and use a client emulator, which emulates client interactions as specified in the TPC-W document.

To determine the peak throughput for each cluster configuration we run a step-function workload, whereby we gradually increase 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 database. Although persistence is ensured through an InnoDB database, our prototype currently requires a translation of the database from the InnoDB table format into the MySQL heap table format before initial mmap-ing. We run each experiment for a sufficient time such that the benchmark's operating data set becomes memory resident and we exclude the initial cache warm-up time from the measurements. Our experiments focus on demonstrating the system scalability, resiliency and efficiency of failover.

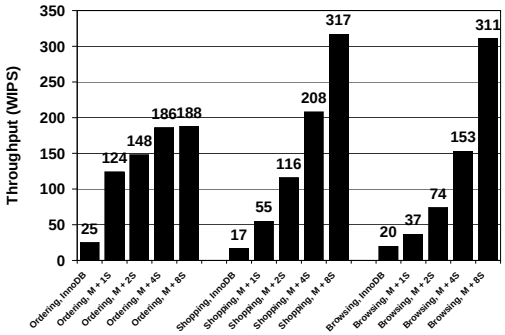


Figure 3. Comparison against InnoDB.

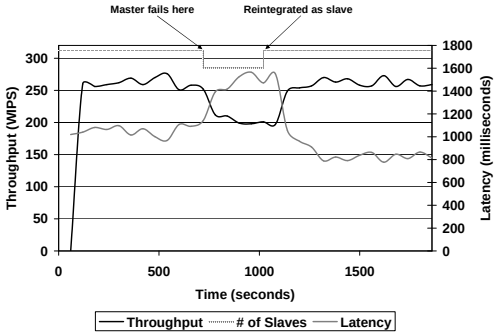


Figure 4. Node reintegration (shopping mix).

6. Experimental Results

In our experimental evaluation, we first show the performance benefits brought about by our fast in-memory transactional layer, compared to a stand-alone on-disk InnoDB database, in Section 6.1. Then, we demonstrate fast reconfiguration under failures in our in-memory tier versus a stand-alone InnoDB replicated tier in Section 6.2.

6.1. Performance Experiments

Figure 3 shows the throughput scaling we obtained over the fine-tuned single InnoDB on-disk database back-end. In the experiment InnoDB was configured for serializable concurrency control. We performed experiments with 1, 2, 4 and 8 slave replicas respectively. Overall, we improve performance over stand-alone InnoDB by factors of 6.5, 17.6 and 14.6 in our largest configuration with 8 slaves with InnoDB for the ordering, shopping and browsing mixes respectively. Furthermore, we can see that a performance jump is seen from adding the in-memory tier even in the smallest configuration due to its superior speed. Finally, system throughput scales close to linearly with increases in in-memory tier size for the browsing and shopping mixes and less well for the ordering mix. This is caused by saturation of our master database with update transactions including lock waiting on the master as a side-effect of the costly index updates in our system (due to rebalancing for inserts in the RB-tree index data structure). The read-only transactions aborted due to version inconsistency are below 2.5% out of the total number of transactions in all experiments.

6.2. Failure Reconfiguration Experiments

In this section, we first show a fault tolerance experiment with reintegration of a failed node after recovery. Next, we concentrate on fail-over onto backup nodes and we show the impact of the three components of the fail-over path on performance in our system: cleanup of partially propagated updates for aborted transactions, data migration and buffer cache warmup. For this purpose, we inject faults of either master or slaves in the in-memory tier and show reconfiguration times in the following scenarios:

i) **Stale backup case:** Master or active slave failure with reintegration of the failed node or integration of a stale backup node. ii) **Up-to-date cold backup case:** Master or active slave failure followed by integration of an up-to-date spare backup node with cold buffer cache. iii) **Up-to-date warm backup case:** Master or active slave failure followed by integration of an up-to-date and warm spare backup node. We also compare our fail-over times with the fail-over time of a stand-alone on-disk InnoDB database.

6.2.1 Fault Tolerance with Node Reintegration Experiment

In this section, we evaluate the performance of the node reintegration algorithm we introduced in section 4. The algorithm permits any failed node to be reallocated to the workload after recovery. This implies a period of node down-time (e.g., due to node reboot).

We use the master database and 4 slave replicas in the test cluster configuration, running the shopping TPC-W workload. Figure 4 shows the effect of reintegration on both throughput and latency.

We consider the most complex recovery case, that of master failure by killing the master database at 720 seconds by initiating a machines reboot. We see from the graph that the system adapts to this situation instantaneously with the throughput and latency gracefully degrading by a fraction of 20%. Since all slave databases are active and execute transactions their buffer caches are implicitly warm. Hence throughput drops no lower than the level supported by the fewer remaining slave replicas.

After 6 minutes of reboot time (depicted by the line in the upper part of the graph), the failed node is up and running again, and after it subscribes with the scheduler, the scheduler initiates its reintegration into workload processing as a slave replica. Since we used a checkpoint period of

40 minutes, this experiment shows the *worst case* scenario where all modifications since the start of the run need to be transfered to the joining node. It takes about 5 seconds for the joining node to catch up with the missed database updates. After the node has been reintegrated, it takes another 50 to 60 seconds to warm-up the in-memory buffer cache of the new node, after which the throughput is back to normal. The next section provides a more precise breakdown of the different recovery phases.

6.3. Failover Experiments

In this section we evaluate the performance of our automatic reconfiguration using fail-over on spare back-up nodes. In all of the following experiments we designate several databases as backup nodes and bring an active node down. The system immediately reconfigures by integrating a spare node into the computation immediately after the failure.

We measure the effect of the failure as the time to restore operation at peak performance. We run the TPC-W shopping mix and measure the throughput and latency that the client perceives, averaged over 20 second intervals.

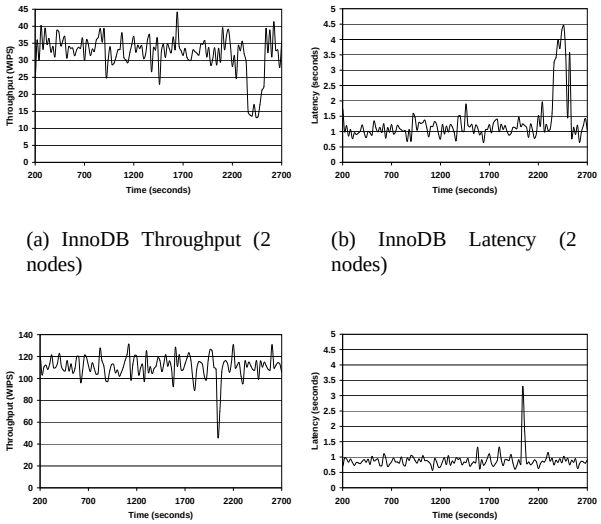
Depending on the state of the spare backup, we differentiate failover scenarios into: *stale backup*, *up-to-date cold backup* and *up-to-date warm backup*. In the *stale backup* experiments, the spare node may be behind the rest of the nodes in the system, so both catch-up time and buffer warm-up time are involved on fail-over. In the *up-to-date* experiments, the spare node is in sync with the rest of the nodes in the system, but a certain amount of buffer warm-up time may be involved.

Stale Backup

As a baseline for comparison, we first show the results of fail-over in a dynamic content server using a replicated on-disk InnoDB back-end. This system is representative for state-of-the-art replicated solutions where asynchrony is used for scaling a workload to a number of replicas.

In this experiment, the InnoDB replicated tier contains two active nodes and one passive backup. The two active nodes are kept up-to-date using a conflict-aware scheduler [6] and both process read-only queries. The spare node is updated every 30 minutes. Figures 5(a) and 5(b) show the failover effect in an experiment where we kill one of the active nodes after 30 minutes of execution time. We can see from the figure that service is at half its capacity for close to 3 minutes in terms of lowered throughput, and higher latency correspondingly.

We conduct a similar experiment interposing our in-memory tier, having a master and two active slaves and one 30 minute stale backup. We used two active slaves, because



(a) InnoDB Throughput (2 nodes)

(b) InnoDB Latency (2 nodes)

(c) DMV Throughput (M + 2S)

(d) DMV Latency (M + 2S)

Figure 5. Failover onto stale backup: comparison of InnoDB and DMV databases.

in the normal operation of our scheme the master is kept lightly loaded and does not execute read-only transactions. Subsequently, we kill the master node to generate the worst-case fail-over scenario that includes master reconfiguration. The results are presented in figures 5(c) and 5(d). In this case, the total failover time is about 70 seconds, less than a third of the InnoDB fail-over time in the previous experiment.

Furthermore, from the breakdown of the time spent in the three fail-over stages presented in Figure 6, we can see that most of the fail-over time in our in-memory Dynamic Multiversioning based system is caused by the buffer-cache warm-up effect. The figure also compares the durations of the failover stages between the InnoDB and in-memory Dynamic Multiversioning cases. We can see that the database update time during which the database log is replayed onto the backup (DB Update) constitutes a significant 94 second fraction of the total fail-over time in the InnoDB case. This time reflects the cost of reading and replaying on-disk logs. In contrast, the catch up stage is considerably reduced in our in-memory tier where only in-memory pages are transfered to the backup node. The cache warm-up times are similar for both schemes. For the DMV case, there is an additional 6 second clean-up period (Recovery), during which partially committed update transactions need to be aborted due to the master failure and master reconfiguration occurs.

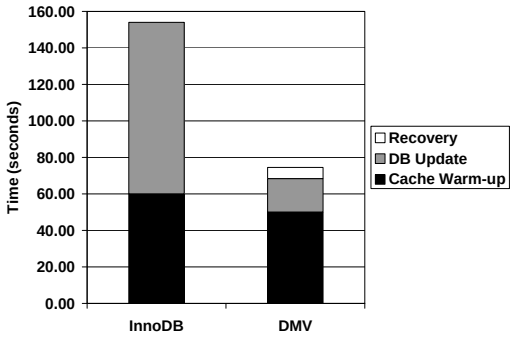


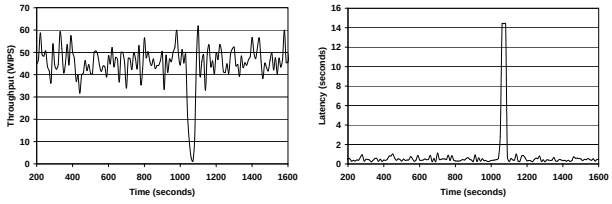
Figure 6. Failover stage weights: cleanup (Recovery), data migration (DB Update) and Buffer cache warmup (Cache Warmup)

Up-to-date Cold Backup

In this suite of experiments, the spare node is always kept in sync with the rest of the system by sending it the log of modifications.

In order to emphasize the buffer warmup phase during failover, we used a slightly larger database configuration comprised of 400K customers and 100K items. This yielded a database size of 800MB and a resident working set of approximately 460MB. We use a three-node cluster: one master, one active slave and one backup.

In this first experiment, the buffer cache of the spare node is cold, so upon fail-over the database needs to swap-in a significant amount of data, before achieving peak performance. We run the TPC-W shopping mix and after approximately 17 minutes (1030 seconds) of running time, we kill the active slave database forcing the system to start integrating the cold backup. Figure 7 shows the perceived throughput for the duration of the cold backup experiment.



(a) Throughput (b) Latency

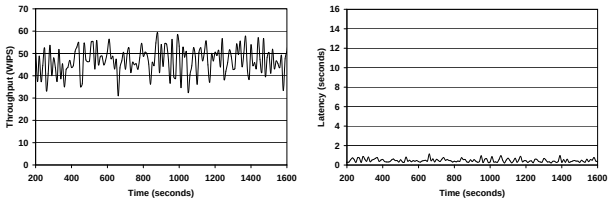
Figure 7. Fail-over onto cold up-to-date DMV backup.

We can see that the drop in throughput is significant in this case due to the need to warm-up the entire database cache on the cold backup. It takes more than 1 minute until the peak throughput is restored.

Up-to-date Warm Backup

In this section, we investigate the effect on fail-over performance of our techniques for mitigating the warm-up effect.

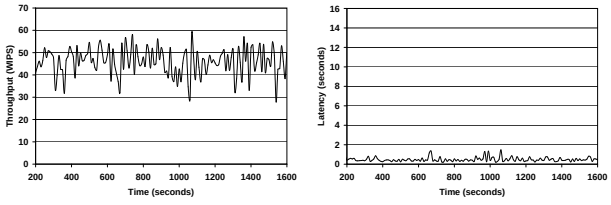
In the first case, the scheduler sends 1% of the read-only workload to the spare backup node. We conduct the same experiment with the same configuration as above and we kill the active slave database at the same point during the run as in the previous experiment. As before, the system reconfigures to include the spare backup. Figure 8 shows the throughput for this case. The effect of the failure is almost unnoticeable due to the fact that the most frequently referenced pages are in the cache.



(a) Throughput (b) Latency

Figure 8. Fail-over onto warm DMV backup with 1% query-execution warm-up.

Figure 9 shows the failover effect for our alternate backup warmup scheme using page id transfers from an active slave. The active slaves transfers page ids to the backup every 100 transactions while the backup touches these pages. We see that the performance in this case is the same as that for periodic query execution allowing for seamless failure handling.



(a) Throughput (b) Latency

Figure 9. Fail-over onto warm DMV backup with page id transfer.

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 the IBM DB2 HADR suite [1] to research and open-source prototypes, such as MySQL Cluster [17], C-JDBC [9, 10], Postgres-R [14] and Ganymed [20].

The industry solutions provide both high availability and good scalability, but they are costly and require specialized hardware such as Shared Network Disk [3]. MySQL Cluster [17] provides very fast in-memory replicated storage engine with lazy logging of updates, similar to the one in our system prototype. However, it uses traditional two-phase locking for concurrency control which may stall readers accessing data that's being modified. In contrast, our solution resolves read/write conflicts optimistically and hence avoids thread blocking. Existing research prototypes use commodity software and hardware, but they either have limited scaling for moderately heavy write workloads [5, 10] due to their use of coarse-grained concurrency control implemented in the scheduler, or sacrifice failure transparency and data availability by introducing single points of failure [20]. Even the solutions that offer transparent fail-over and data availability [5] do so by means of complex protocols due to the crucial data that resides inside the scheduler tier. In contrast, our solution provides transparent scalability as well as fast, transparent failover. The scheduler node is minimal in functionality, which permits extremely fast reconfiguration in the case of single node fail-stop scenarios.

Previous work in the area of primary-backup replication [24] has mostly followed a "passive backup as a hot-standby" approach where the backup simply mirrors the updates of the primary. These solutions either enforce a fully synchronous application of updates to the backup or do not enforce strict consistency although the backup does maintain a copy of the database on the primary. The backup is either idle during failure free system execution [24] or could execute a different set of applications/tasks. In contrast to these classic solutions, in our replicated cluster, while backups are used for seamless fail-over, a potentially large set of active slaves are actively executing read-only transactions with strong consistency guarantees.

More recent efforts towards integration of database fine-grained concurrency control and replication techniques use snapshot isolation [11, 23, 20] to minimize consistency maintenance overheads. These solutions depend on support for multiversioning within each database replica. In contrast, our solution dynamically creates the required versions on a set of distributed replicas.

8. Conclusions

In this paper, we introduce novel lightweight reconfiguration techniques for the database tier in dynamic content web sites. Our solution is based on an in-memory replication algorithm, called Dynamic Multiversioning, which provides transparent scaling with strong consistency and ease of reconfiguration at the same time.

Dynamic Multiversioning offers high concurrency by exploiting the naturally arising versions across asynchronous database replicas. We avoid duplication of database functionality in the scheduler for consistency maintenance by integrating the replication process with the database concurrency control. Furthermore, we avoid copy-on-write overheads associated with systems that use stand-alone database multiversioning offering snapshot isolation. We show how a version-aware scheduler algorithm distributes transactions requesting different version numbers across different nodes, thus keeping aborts due to version conflicts at negligible rates.

Our evaluation shows that our system is flexible and efficient. While a primary replica is always needed in our in-memory tier, a set of active slaves can be adaptively and transparently expanded to seamlessly accommodate faults. We scale a web site using an InnoDB on-disk database backend by factors of 14.6, 17.6 and 6.5 for the TPC-W browsing, shopping and ordering mixes, respectively when interposing our intermediate in-memory tier with 9 replicas. We also show that our in-memory tier has the flexibility to incorporate a spare backup after a fault without any noticeable impact on performance due to reconfiguration.

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] 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.

[5] 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.

[6] 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.

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

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

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Jun 2004.

[10] 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.

[11] S. Elnikety, F. Pedone, and W. Zwaenepoel. Generalized snapshot isolation and a prefix-consistent implementation. Technical Report IC/2004/21, EPFL, 2004.

[12] IBM. High availability with DB2 UDB and Steeleye Lifekeeper. IBM Center for Advanced Studies Conference (CASCON): Technology Showcase, Toronto, Canada, Oct 2003.

[13] B. Kemme and G. 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.

[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] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPOPP*, pages 198–208, 2006.

[17] MySQL Cluster. `http://www.mysql.com/products/database/cluster/`.

[18] 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.

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

[20] 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.

[21] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *EuroSys'06: Proceedings of the EuroSys 2006 Conference*, pages 89–102. ACM, 2006.

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

[23] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency controlbased on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering*, Apr 2005.

[24] Y. Zhou, P. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. In *Proc. of the Int'l Conference on Supercomputing*, June 1999.