Scalable and Highly Available Database Replication through Dynamic Multiversioning

by

Kaloian Manassiev

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Scalable and Highly Available Database Replication through Dynamic Multiversioning

Kaloian Manassiev

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

This dissertation describes *Dynamic Multiversioning*, a novel replication protocol, providing scalability, consistency and ease of reconfiguration for the back-end database in dynamic content servers. The key idea is the cooperation between a request *scheduler*, which orchestrates the replicated cluster, and a versioning-based replication technique integrated with the database fine-grained concurrency control engine. The scheduler distributes transactions on a set of lightweight database replicas and enforces serialization order by executing updates on a master node. The version-aware scheduling scheme guarantees strong 1-copy-serializability.

Experiments with up to 9 database replicas show throughput scaling factors of 14.6, 17.6 and 6.5 respectively for the browsing, shopping and even for the write-heavy ordering workload of the industry-standard TPC-W benchmark. In addition, our technique guarantees that any crucial data can quickly and easily be recovered, which facilitates almost instantaneous reconfiguration, without loss of data, in the case of single-node failures of any node in the system.

Acknowledgements

First and foremost, I want to thank my supervisor, Cristiana Amza, for taking me as a student and for showing me how to do research. Her constant direction, help and encouragement were invaluable, and without them, this thesis would not have been possible.

I am deeply thankful to my second reader, Michael Stumm, for taking time out of his busy schedule to read the first draft of this thesis, and for his numerous and invaluable corrections, comments and suggestions on the text.

I want to say a big thank you to my parents for bringing me to Canada and to my entire family for their confidence and trust in me.

Finally, but certainly not the least, I want to thank my officemates and other friends for their priceless friendship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis proposes and evaluates *Dynamic Multiversioning*, an asynchronous database replication technique for the back-end database server of dynamic-content web sites. Our replication system interposes an in-memory data replication tier between the application and the database servers, which eases the implementation of highly scalable, self-configuring and self-repairing database cluster servers, providing strong consistency guarantees (i.e., *strong one-copy serializability* [16]).

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 1.1). 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 possibly a set of parameters and values. 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.

With the advances in Internet technologies and e-commerce, more and more businesses are exposing their services through dynamic-content web-sites. Hence, these sites currently need to provide very high levels of availability and scalability. On-the-fly re-

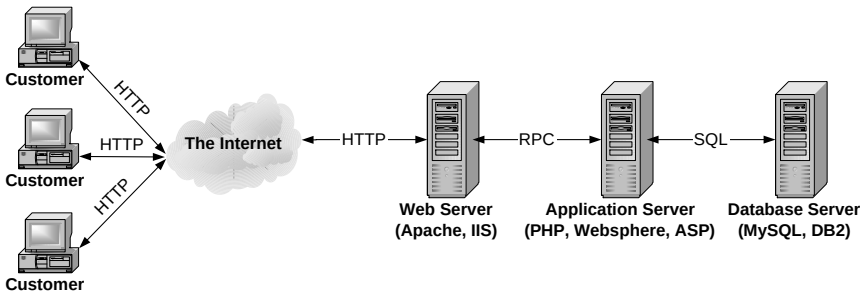


Figure 1.1: Common Architecture for Dynamic Content Sites

configuration may be needed to either adapt to failures or bursts of traffic and should be automatic, fast and transparent. In many current applications (e.g., e-commerce, bulletin boards, 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 the performance bottleneck [8]. At the same time, it is the database tier that makes scalability and fast reconfiguration hard to achieve. Above all, data consistency needs to be ensured during reconfiguration, typically through complex and lengthy recovery and data migration procedures [38].

Several research works have demonstrated that the database tier reconfiguration problem is exacerbated by the imposed requirement to scale it through asynchronous content replication solutions [9, 11, 17, 36, 21, 30]. These works have shown that an asynchronous operation of the different database replicas is 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 in case of failures. 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 [36]. Other highly scalable replicated cluster solutions [21, 33] trade-in strong consistency by using specialized consistency semantics, which is confusing to the application programmer. Alternatively, complex failure reconfiguration protocols, providing data consistency and integrity, imply reloading transactional

logs from disk and replaying them on a stale replica. Resuming servicing transactions at peak-throughput can take on the order of minutes [2, 26, 38] and possibly more; fail-over times are rarely formally measured and reported.

In this thesis, we introduce a replication solution that combines data consistency, transparent scaling and split-second reconfiguration. This research stems from the belief that, in order to be able to build a self-reconfiguring large-scale system out of complex components such as databases, the design assumptions and inner mechanisms of that system as a whole need to be reexamined. Our key idea is to interpose an *in-memory* middleware tier consisting of lightweight database engines providing scaling and seamless adaptation to failures on top of a traditional on-disk database back-end. Our transactional tier implements a protocol called *Dynamic Multiversioning*, a novel database replication solution, fundamentally redesigned from the outset to meet the pre-requisites for both scaling and ease of self-reconfiguration of the overall system. Specifically, our replication solution has the following desirable properties:

1. provides strong consistency semantics identical to a one-copy database (i.e., strong one-copy serializability [16]), 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

The in-memory aspect of our system is not essential in itself, and the on-disk database back-end could be re-engineered in a similar fashion. However, we find that fundamentally restructuring an existing highly tuned on-disk database engine with its myriad features is a formidable task. An in-memory transactional tier solution, such as the one presented in this thesis is thus attractive due to its high degree of flexibility in terms of design choices,

high speed during normal operation and inherent agility in reconfiguration during failures. Indeed, we believe that a large scale in-memory transactional tier is the way of the future for scaling the database throughput in dynamic content sites given: i) the fast projected growth in main-memory sizes for commodity servers and the quickly falling prices, and ii) the commonly encountered bottleneck is the database CPU rather than the disk due to the relatively modest database sizes commonly used in e-commerce [40] and their high degree of access locality [8]. In this context, our system, as designed, provides the much needed separation of the *CPU scaling* from the *data persistence* concerns in dynamic content serving. The in-memory replication scheme provides fast CPU scaling on potentially *many* lightweight database components. On the other hand, persistence is provided by executing the update queries on an on-disk back-end database. If storage availability is also desired, it can be provided by an orthogonal, replication scheme using more closely synchronized persistent updates onto a *few* on-disk databases.

Our in-memory replication scheme is itself *asynchronous* in order to provide scaling. The key idea is to redesign the database fine-grained concurrency control to work synergistically with data replication in an integrated solution that ensures ease of reconfiguration. Update transactions always occur on an in-memory *master* database, which broadcasts modifications to a set of in-memory *slave* database replicas. Each update to the master 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 tries to scatter to different replicas transactions, which may require different versions of the same memory

page. There, each creates the page versions it needs and the 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 an entire transactional log on a stale replica, we replicate only the changed pages having versions newer 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 and index structures, 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 [32], and the MySQL [4] 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 [40]. 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 [1], the PHP Web-scripting/application development language [6] and the MySQL database server [4]. We use MySQL servers with InnoDB tables [3] 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 scale 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 dissertation is organized as follows. Chapter 2 presents the necessary background information on the methods of replication and their benefits and trade-offs. Chapter 3 introduces our *Dynamic Multiversioning* solution and highlights the high level design of our system. Chapter 4 describes the fault-tolerance and fast-reconfiguration aspects that it offers. Chapter 5 gives details on our prototype implementation. Chapters 6 and 7 describe our experimental platform, benchmarks, methodology and present our results. Chapter 8 discusses related work done in the area of database replication and motivates this research. Chapter 9 gives our conclusions.

Chapter 2

Background

The theory of replicated databases is discussed throughly by Bernstein et al. in [14]. Gray et al. present a classification of the various methods of replication and analyze their benefits and trade-offs [24]. These classic readings have been the basis for most of the contemporary research in database replication. This chapter provides a brief overview of the foundations of replication and sets up a context for presenting our *Dynamic Multiversioning* solution.

2.1 Replication and Consistency Models

A replicated database is a distributed database in which multiple copies of the data items are present on several physically separated machines [14]. Replicating data across network nodes can increase performance and availability. However, these improvements do not come directly and are complicated by the presence of updates to the data items. In particular, when an update occurs to an item residing on certain node, this modification has to be propagated to every other node in the system, so that a subsequent read will see an up-to-date data. Thus, replication requires careful synchronization protocols and establishment of rules as to where updates to the data entries are permitted to occur and what is the item validity or staleness bound that subsequent read operations are

guaranteed to see.

In the general case, a replicated DBMS is an attractive choice if it hides all aspects of data replication from client applications. For the purposes of this dissertation, we will consider *strong one-copy serializability* as our model of consistency. The term *strong serializability* was first introduced by Breitbart et al. [16] and then extended to replicated databases by Khuzaima et al. [21]. The formal definition of *strong one-copy serializability* is:

Definition 1 *A transaction execution history H is strongly one-copy serializable iff it is one-copy serializable and for every pair of committed transactions T_i and T_j, such that T_i's COMMIT precedes T_j's BEGIN in H, there exists a serial one-copy history H' equivalent to H, in which T_i precedes T_j.*

This consistency model requires that each history of concurrent transaction executions produces results equivalent to some sequential execution of the same set of transactions on a single instance of the database and the relative ordering of non-concurrent transactions is preserved in the resulting serialized execution history. What this implies is that clients to the replicated cluster perceive it as a single-instance database and read operations always obtain the latest committed value of an item.

Throughout our descriptions, we will use the classic transaction execution sequence, in which clients initiate a transaction by sending a `BEGIN` request and follow it by one or more `READ/WRITE` requests. The transaction is ended by either a `COMMIT` or `ABORT` request. If `COMMIT` was sent and it was carried successfully, the modifications are persisted and are guaranteed to survive failures of the system. Otherwise, on `ABORT` or in case of a system failure, all effects of the transaction are canceled as if it had never executed. In addition, the executions of concurrent transactions never see each other's tentative changes.

2.2 Replication Schemes

Gray et al. defined a taxonomy of the various replication schemes, based on the nodes where updates to data are permitted to occur (*master* or *group* ownership) and on the manner in which modifications are propagated within the system (*lazy* or *eager*) [24]. This section summarizes this taxonomy.

Master Replication

In *master* replication, each object is assigned to a (*master*) node, which all update transactions contact, when they need to perform modifications to that object. The most recent version of the object is thus always available at the master node. The key benefit of this replication scheme is the presence of single instance, which establishes the conflict order of update operations to the object. This obviates the need for complex conflict resolution protocols. However, it has the major drawback of the master node, possibly becoming a bottleneck in cases of heavy update traffic or presence of hot-spots in database.

Group Ownership

In a replication system using *group ownership* there are no object masters and the system permits updates to any object to occur at any replica. This kind of replication has certain benefits for mobile or geographically dispersed applications, where connectivity with a master node is not always practical. In some situations it is also beneficial to applications with a high ratio of updates. However, an analysis by Gray et al. [24] points out that group ownership-based replication either requires extremely complex algorithms for synchronizing conflicting updates occurring concurrently on separate replicas, or skewing the data consistency semantics, thus permitting clients to read stale or even inconsistent data [35] for certain periods of time.

The above two data replication categories regulate where updates to replicated objects

are permitted to occur. After data has been modified, it is the responsibility of the replication system to propagate updates to the other nodes in the system. The *eager* and *lazy* replication models, explained below, determine when data modifications are actually delivered to and applied on the remaining replicas.

Eager Replication

Eager replication postulates that modifications to all replicated items should be delivered to all the remaining nodes synchronously, as part of the main updating transaction. Upon receipt of the update transaction's `COMMIT` statement, the node that executes it ensures all replicas that have not crashed have received and acknowledged the update, after which it deems the transaction committed. The major advantage of this replication model is that it automatically guarantees consistency and one-copy serializability, which in turn facilitates database crash recovery. This comes by the fact that any conflicts or failures of replica nodes can be discovered at commit time, and reported to the client who issued the `COMMIT`. A major drawback is the fact that the commit of the updating transaction is delayed until all nodes which are up and running have acknowledged that they have received the updates. Thus, *eager* replication may potentially severely limit scalability.

Lazy Replication

Lazy replication, on the other hand, propagates updates to replicated items asynchronously, outside of the main update transaction. Upon receiving of the transaction's `COMMIT` statement, the database implementing lazy replication commits the transaction locally and replies without delay. After some predefined period of time, the replication module spawns "refresh" transactions on each of the replica nodes in order to propagate the changes. This replication model is very scalable, because the commit of the main transaction is not delayed as it is in the *eager* case. It also facilitates disconnected operation and is thus suitable for mobile devices. However, this flexibility incurs windows of vul-

	Eager	Lazy
Group	+ Serializable execution - Deadlocks possible - does not support a lot of nodes	+ Scalability - System delusion - Does not support a lot of nodes
Master	+ Serializable execution - Deadlocks – does not support a lot of nodes	+ Serializable execution (if reads go to master)

Table 2.1: Properties of various replication techniques

nerability where committed updates could potentially be lost if the node that executed the update transaction crashes before it had the chance to update the remaining replicas.

The *master* and *lazy* replication models are used in conjunction with the two ownership strategies in order to build replication systems suitable for different classes of applications. Table 2.1 outlines the combinations of different replication techniques and object update strategies and their most important characteristics.

Hybrid Replication Solutions

State-of-the-art research solutions in database replication try to circumvent the major problems associated with purely lazy or purely eager replication, by combining beneficial features characteristic to both. Chapter 8 gives extended details on these solutions as they are closely related to and motivate our work.

Chapter 3

Dynamic Multiversioning

This chapter presents at a relatively high level, the architecture and operation of the *Dynamic Multiversioning* replication system. We describe the main components of the system and the way they work in synergy to provide a scalable and consistent solution. In the next chapter, we describe the reconfiguration techniques that *Dynamic Multiversioning* facilitates and how they provide ease of reconfiguration in the case of single node fail-stop scenarios.

3.1 Overview

The ultimate goal of Dynamic Multiversioning is to scale and provide fault-tolerance for the back-end database through an in-memory replication tier. This tier implements a novel distributed concurrency control mechanism that integrates per-page fine-grained concurrency control, consistent replication and version-aware scheduling. We interpose a scheduler between the front-end web/application server tier and the database tier. A set of in-memory master and slave databases memory-map an on-disk database at startup. In addition, one or more on-disk InnoDB replicas may be in charge of making the updates persistent on stable storage in order to permit recovery in case the entire cluster fails. The scheduler distributes requests on the replicated in-memory database cluster and

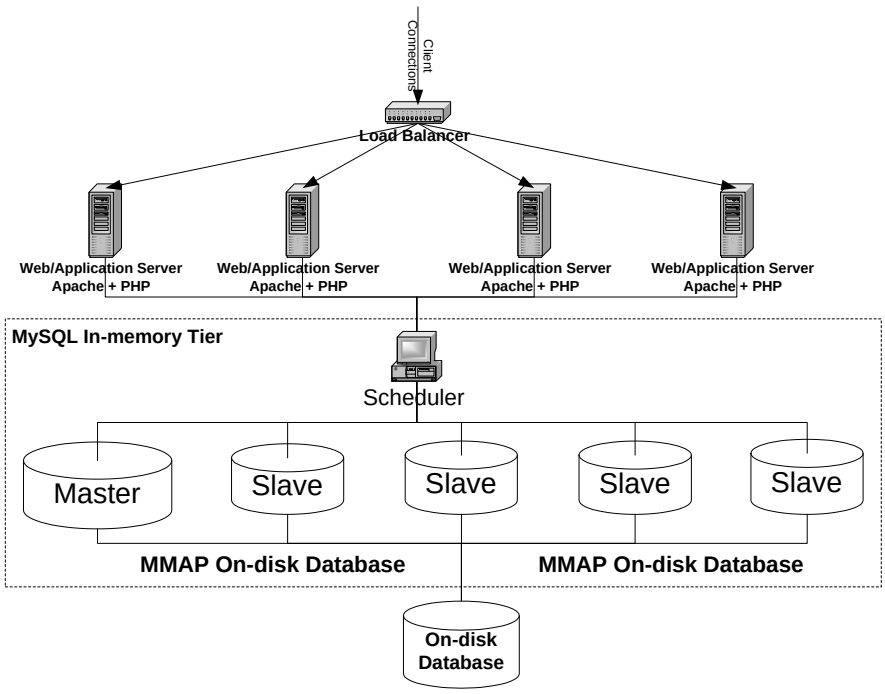


Figure 3.1: System design

may itself be replicated for performance or for availability (see Chapter 4). It sends write transactions to a master database and read-only transactions to a set of database slaves as shown in Figure 3.1.

The idea of isolating the execution of conflicting update and read-only transactions through multiversioning concurrency control is not new [14]. There exist stand-alone databases supporting multiversioning (e.g., Oracle and MySQL's InnoDB). However, they 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.

We augment a simple in-memory database with a replication module implementing a hybrid scheme that has all the benefits of both *eager* and *lazy* replication, without experiencing their chief problems. *Dynamic Multiversioning* 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 and ii) lazy, by delaying the application of modifications on slave replicas and creating item versions on-demand as needed for each read-only transaction.

The following sections give more detail on how our fine-grained dynamic multiversioning scheme works.

3.2 Components of the In-memory Replication Tier

Scheduler

The *scheduler* abstracts the replicated cluster, making it appear as a single coherent database to external clients. It accepts client connections, maintains sessions and dispatches received SQL requests to the appropriate database. In our prototype system, the scheduler communicates through the MySQL client-server protocol. That way, existing applications, coded to use MySQL as a database, do not need to be recompiled or modified in any way.

Upon receipt of an SQL request, the scheduler first determines whether this is a read-only query or an update. Based on that information, the scheduler either sends it to execute on the *master*, if it is an update, or sends it to one of the *slaves* if it is a query. The type of the request can be determined either automatically, by parsing the request string, or manually by requiring the client to explicitly specify a type. Generally, it is possible to parse single statement SQL requests and automatically extract their type, but in the case of multi-statement transactions, the developer needs to prefix the transaction with that meta-information.

Master Node

The *master* database is used to process *update* transactions. It decides the order of execution of write transactions based on its internal two-phase-locking per-page concurrency

control. Each update transaction committing on the master produces a new version of the database. The master synchronously flushes its modifications to disk and to the slave nodes, upon each transaction commit. The master then communicates the most recent version number produced locally to the scheduler, upon acknowledgment of the commit of each update transaction. The scheduler tags each read query with the version number that it is supposed to read (i.e., the most recent version produced by the master) and sends it to execute on a slave database. Capturing modifications at the master is done at the level of virtual memory pages using traditional virtual memory protection violation [27].

It should be noted here that the per-page concurrency control and replication granularity is not an inherent limitation of our system. The design of the protocol permits it to work with any item size, as long as the replication granularity is the same as that of the concurrency control engine.

Slave Nodes

Slave nodes in the system are used to process the typically heavier read-only query workload. When a read-only transaction is executing on the slave, for each page read request that it generates, the slave's data manager ensures that all local fine-grained modifications from the master, assigned to that page have been applied up to the desired version. Thus, the slave dynamically and incrementally builds a consistent snapshot of the database version that the read-only query is supposed to read. Specifically, the slave replica applies all local fine-grained updates received from the master on the necessary pages, up to and including version `Vn` that the executing read-only transaction has been tagged with. Since versions for each page are created on-demand by in-progress read-only transactions, read-only transactions with disjoint read sets will never touch the same page and thus can 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.

3.3 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 with the last version number communicated by the *master* replica and sending it to execute on a slave replica. The execution of read-only transactions on slave replicas is isolated from any concurrent updates executing on the master. This means that a read-only transaction will never see modifications on items that were written later than the version it was assigned.

The scheduler tags read-only queries with versions and thus it is aware of the data versions that are about to be created at each slave replica. It sends a read-only transaction to execute on a replica, where the chance of conflicts is the smallest. We have implemented a heuristic, which collects conflict statistics on-the-fly, and using that information tries to select a replica from the set of databases, where read-only transactions with the same version number as the one to be scheduled are currently executing, if such databases exist. Otherwise it picks the database where the conflict probability is lowest. The scheduler load balances across the database candidates thus selected using a shortest execution length estimate as in the SELF algorithm [11].

In the case of imperfect scheduling (e.g., due to insufficient replicas), a read-only transaction may need to wait for other read-only transactions using a previous version of an item to finish. A read-only transaction T1 may also need to be aborted if another read-only transaction T2 upgrades shared item to a version higher than the version that T1 already read for a different item, however we expect such cases to be rare.

Chapter 4

Fault Tolerance

In this chapter, we first define the failure model that we support and then describe our reconfiguration techniques in the case of master, slave or scheduler node failures. Finally, we describe our mechanisms for data persistence and availability of storage in the on-disk back-end database tier.

4.1 Failure Model

In this work, we assume a *fail-stop* failure semantics. We support *single-node* at a time failures with respect to the *master* and *scheduler* nodes, and multiple simultaneous failures of *slave* nodes. That is, we assume that if a certain node fails, no other failure is going to occur up until the moment the system has been brought back to a *safe state*. As we will see further in this chapter, this rule can be relaxed for the cases where more than one slave node crashes. We define *safe states* in our system to be the periods when the system is able to service requests and the requests are guaranteed to leave the database in a consistent state and return consistent results. If the system is not in a *safe state*, in order to avoid non-deterministic executions, certain client requests (e.g. updates) may be either rejected or queued, up to the moment the system enters the *safe state*.

We assume that the underlying communication system is responsible for detecting

node failures and that a failure or crash of a process is treated in the same way as a failure of the node itself. We rely on communication primitives which return an error if a request to the remote node cannot be delivered and that once such a primitive returns an error, subsequent attempts to connect to the node are also going to fail. During time periods where nodes are waiting for a response or are idle and do not exchange messages, failure of any individual node is detected through missed heartbeat messages.

Situations of permanent or transient network partitions can be handled by assigning *ranks* to request schedulers or using a *majority-view* model as in [41] such that only the partition containing a particular scheduler rank or the one having the majority of nodes continues operation.

4.2 Node Failures

4.2.1 Scheduler Failure

The scheduler node in our system 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 a single version number, 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 current master database asking it to abort all uncommitted transactions that were active at the time of failure. This may not be necessary for databases which automatically abort a transaction due to broken connections with their client. After the master executes the abort request, it replies back with the highest database version number it produced. Then, the new scheduler broadcasts a message to all the other nodes in the system, informing them of the new topology. After this process completes, the system is in a *safe state* and may resume servicing requests.

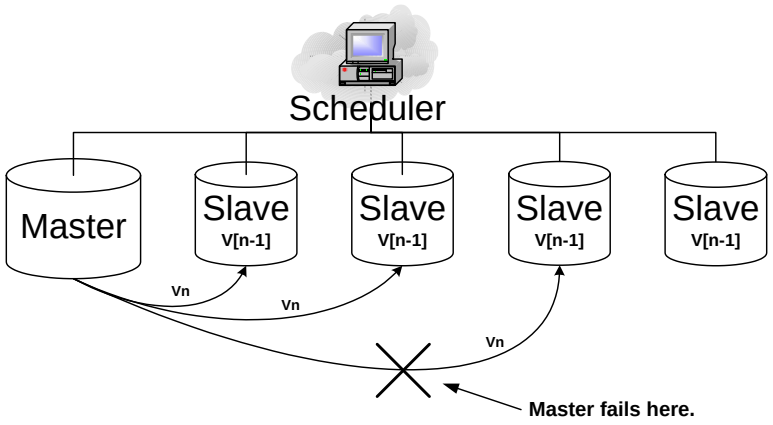


Figure 4.1: Master failure situation with partially propagated modifications log flush message

It should be noted here that while the system is not in a *safe state*, due to scheduler failure, clients are unable to connect to the cluster and thus cannot submit any requests. For that reason, the consistency of the data is guaranteed.

4.2.2 Master Failure

Upon detecting a master failure, the first scheduler takes charge of the recovery process. It asks all databases to discard their modifications log records, which have version numbers higher than the latest version number it has seen from the master. This procedure takes care of cleaning up transactions whose pre-commit modifications log flush message may have partially completed at a subset of the replicas, but the master has not acknowledged to the scheduler the commit of the transaction before failure. The situation is depicted in Figure 4.1. Since slave nodes only apply the modifications log on demand, as dictated by the version with which a read-only transaction has been tagged, partially propagated log records will still remain enqueued in the above situation, because their version is not known to the scheduler yet. Thus, the clean-up action at the slaves entails only dequeuing the particular log records.

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, after which the system enters a *safe state* and may continue servicing 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. An error is returned by the scheduler to the client for the failed transaction. The clients are free to resubmit. Our scheme does not attempt to ensure exactly-once transaction semantics (i.e., handling cases where the resubmission of the same update request may not generate idempotent results).

4.2.3 Slave Failure

The final case that needs to be considered is when a slave node fails. 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 a message is sent to the issuing client. The failed slave is then simply removed from the scheduler tables and a new topology is generated. Since a slave node failure requires updating only scheduler-local information, such as the list of outstanding transactions that were sent to this slave, the system does not leave the *safe state* while it is being reconfigured. For that reason, it is possible to handle multiple simultaneous scheduler failures.

4.3 Data Migration for Integrating Stale Nodes

Without support for reintegration of new or failed replicas, the capacity of the replicated cluster would gradually go down to the point of unavailability. In this section, we present the **data migration** algorithm for integrating a stale replica. New replicas are always integrated as slave nodes of the system, regardless of their function prior to failure. For

reintegration of nodes, we use a modification of the single-pass algorithm proposed by Kemme et al. in [28].

The *reintegrating node* (S_{join}) initially contacts one of the schedulers and obtains the identities of the current master and an arbitrary slave node. We refer to this slave node as the *support slave*. Next, S_{join} subscribes to the replication list of the master, obtains the current database version $DBVersion$ and starts receiving modification log records starting from the next version. The node stores these new modifications into its local queues, as any other active slave node, without actually applying them 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, $DBVersion$, as obtained from the master upon joining. The *support node* then selectively transmits to S_{join} only the pages that changed after the joining node's failure.

In order to minimize integration time, all slave nodes implement a simple fuzzy checkpoint algorithm [14], modified to suit our in-memory database. At regular time intervals, each slave starts a checkpoint thread, which iterates the dirty database pages and persists their current contents along with their current version onto stable storage. Our checkpoint 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 the 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.

Although it is currently not implemented, the same checkpoint algorithm can be used for the master node. In this case, we only make the assumption that dirty pages are

flushed to disk using a *no-steal/no-force* buffer management policy [25]. This paradigm requires that the checkpoint thread never writes to stable storage pages, modified by transactions that have not yet committed, and thus obviates the need for transaction rollbacks during recovery. In order to prevent the operating system memory management sub-system from flushing uncommitted pages to disk, we also assume that we have full control over the in-memory area of our database. Modern operating systems provide primitives for "pinning" memory regions, so that pages in them do not get swapped out to disk under pressure conditions.

In our current prototype, the scheduler delays sending reads to the newly integrated node until the node receives and applies all missing pages. However, even this delay can be shortened by implementing on-demand update requests for pages that have not yet been fetched by S_{join} from the *support slave*.

4.4 Fail-Over Reconfiguration Using Spare Backups

We define *failover* as integrating a new replica (called a backup) into the active computation to compensate for a fault with a goal to maintain the same level of overall performance irrespective of failures.

Failover time, the expected throughput drop and subsequent ramp-up time immediately after a failure generally consist of two phases: **data migration** for bringing the node up to date and the new node's **buffer cache warm-up**. 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 warm-up** 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 even if actively receiving the modification logs from the master.

Thus, the backup database may take a relatively long time to reach the peak throughput performance on the primary before the failure. In this section we show how, by using our flexible in-memory tier, these phases and the corresponding overall failover process can be significantly optimized compared to failover in traditional on-disk databases and replication schemes. In order to shorten or eliminate the *buffer cache warm-up* phase, a set of *warm* spare backups are maintained for a particular workload for overflow in case of failures (or potentially overload of the active replicas). These nodes may be either idle, e.g. previously failed nodes that have been reintegrated 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 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, the scheduler assigns a small number of read-only transactions to the spare backups, with the sole purpose of keeping their buffer caches reasonably 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 the most heavily used pages in their buffer cache to the backup nodes periodically. The backup simply touches the pages so that they are kept swapped into main memory.

In this way, the spare backup nodes' valuable CPU resources can be used to service a different workload. For spare backups running a different workload, care must be taken

to avoid interference [38] in the database's buffer cache for the two workloads. In case the circumstances do not permit an up-to-date replica to be kept, e.g. malignant interference between two workloads, the failover replica will need to be brought up-to-date by the means of the node reintegration algorithm described in the previous section. This aspect is, however, beyond the scope of this dissertation.

4.4.1 Transparency of Failures

While in our system every attempt is directed at seamless failures in terms of performance, we do not currently attempt to completely hide failures from the client. Transaction aborts due to failures are thus exposed to the client. Our position is that clean error semantics should include handling of client duplicate requests and it is best implemented as an automated end-to-end solution involving the client. Existing solutions, such as exactly-once transactions [22], which are orthogonal to our solution, can be used to automatically reissue aborted transactions.

We currently provide limited support for query re-execution only for the case of aborts for single-query read-only transactions previously sent to a failed slave. For such aborted read-only queries, the scheduler reissues the query onto another active slave.

4.4.2 Data Persistence and Availability in the Storage Database Tier

In this section, we describe our technique for providing on-disk data persistence and availability. We use a back-end on-disk database tier for data persistence in the unlikely event that *all* in-memory replicas fail. We could, in theory, maintain data persistence without any windows of vulnerability by adding synchronous disk flushes at our master commit; note that our periodic fuzzy checkpoints are not sufficient for this purpose. We prefer, however, to delegate persistence to a specialized on-disk database tier in order to

keep our in-memory tier as lightweight as possible.

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., one or two) on-disk replicas are needed. Once the update queries have been successfully logged, the scheduler can return the commit response to the client without waiting for responses from the on-disk databases. The query logging is performed as a lightweight database insert of the corresponding query strings into a database table [9, 38], while a read-write transaction may contain complex update queries using arbitrary WHERE clauses. Thus, the in-memory read-write transaction execution and in-memory commit plus the query logging is on average faster than the execution of the entire update transaction on the on-disk database. Any other technique for replication of updates is, however, equally usable to provide data availability in the on-disk database tier, working in conjunction with our in-memory database tier. This includes synchronous replication schemes or generic primary-backup database replication schemes with log shipping, as commonly used in high-performance commercial systems, such as IBM's DB2 High Availability and Disaster Recovery solutions [2, 26]. We choose this particular replication technique in the on-disk database back-end for convenience of reusing a technique we already implemented before [9, 38].

Chapter 5

Implementation

This chapter provides details on the prototype implementation of the *Dynamic Multiversioning* replication system. We present the algorithms and data structures that we use for each particular aspect of the scheme.

5.1 Overview

Client applications of the replicated *Dynamic Multiversioning* system operate by opening connections to the scheduler and submitting SQL transactions. The *scheduler* abstracts the replicated cluster, making it look like a single-instance database. It accepts client connections, maintains sessions and dispatches received SQL requests to the appropriate database. In our prototype system, the scheduler uses the the MySQL client-server network protocol for communication. That way, in the general case, existing applications, such as PHP [6], coded to use MySQL as a database, do not need to be recompiled or modified in any way.

We require that each multi-statement transaction starts with a `BEGIN TRANSACTION` and ends with a `COMMIT` or `ABORT` clause. 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 meta-information, stating whether

it is read-only or update. For multi-statement transactions we currently do that by concatenating the `BEGIN TRANSACTION` command with a `READ` or `WRITE` keyword. The type of transactions comprised of a single statement is determined by parsing.

5.2 Implementation Details

For simplicity of implementation, our database replication technique is implemented inside an in-memory database tier. Based on the standard MySQL HEAP table we have added a separate table type called `REPLICATED_HEAP` to MySQL. This new table type is seamlessly integrated with the MySQL's DDL[1] and users may create tables of this kind and use them transparently as they would any other table type. 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 the TreadMarks Distributed Virtual Memory [27] runtime system. The modifications log is accumulated using traditional virtual memory page protection violations, twinning and diffing [10]. The unit of transactional concurrency control is the memory page as well.

Figure 5.1 depicts the layout of the database image and the data structures we use to implement the versioning and replication system.

5.2.1 Integrated Fine-grained Concurrency and Replication

Throughout the execution of an update transaction, the database storage manager at the master 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 each update transaction, write-sets created by the master are encapsulated in *diffs*, which are word level run-length encodings of the modifications performed by

[1] Data Definition Language

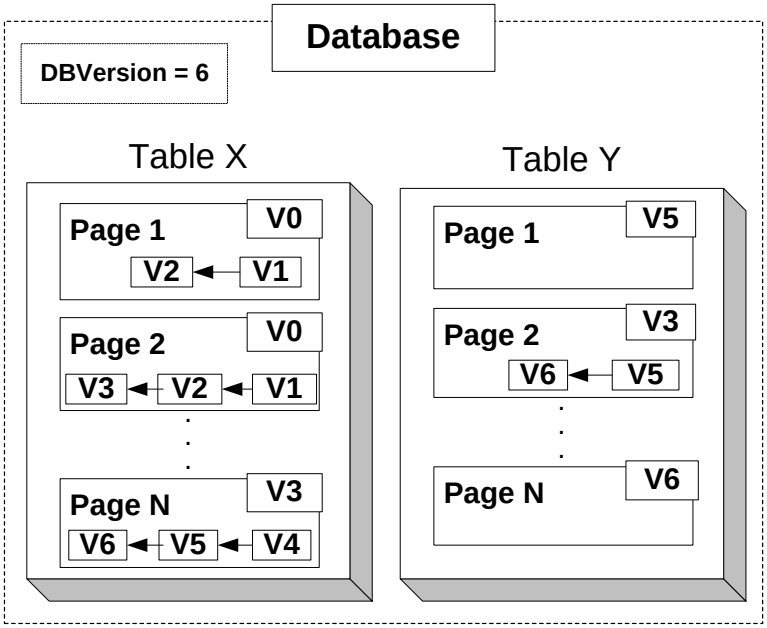


Figure 5.1: Per-page versioning

the data engine on a per-page basis. The master broadcasts the fine-grained modifications produced locally to all active slave replicas, as a pre-commit action for each update transaction. Once the modifications are received at a particular replica, the slave replica sends an acknowledgment immediately back to the master. Upon receiving acknowledgments from all remote replicas, the master commits the transaction locally and reports it to the scheduler. At each remote slave replica, the application of the modifications is however delayed until those modifications are actually needed by a subsequent read-only transaction.

In order to provide consistency and globally serializable transaction execution order, we augment the database engine with a *database version* field. The *master* node increments this version number as a pre-commit action of each committing transaction. Thus, the database version specifies the number of update transactions that have successfully committed. The first function in Figure 5.2 shows the pseudo-code for pre-committing a transaction on the master node.

The parameter `PS` (from Page Set) in the figure is an array-like data structure main-

```
1  MasterPreCommit(PageSet[] PS):
2      WriteSet[] WS = CreateWriteSet(PS)
3      NewDBVersion = DBVersion++
4      For Each Replica R Do:
5          SendUpdate(R, WS, DBVersion)
6          WaitForAcknowledgment(R)
7      Return NewDBVersion
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         ReleaseLock(P.Mutex)
17 End.
```

Figure 5.2: Master node pre-commit, commit and post-commit actions

taining all the pages that the transaction modified during its execution. At *pre-commit*, the master node uses the `MasterPreCommit` function to generate the *write-set* message with diff entries for each modified page. It then increments the database version and sends the write-set tagged with the new version number corresponding to the current transaction to all the slave replicas in a *diff flush* message.

The increment of `DBVersion` in line 3 is implemented as an atomic operation, so that even concurrent transactions always produce different versions. We assume that the communications channel used preserves the total order of messages, so that they are received at the slaves in the same order in which they were sent. 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 number on the reply packet. The scheduler remembers the new version number and uses it to tag all subsequent read-only transactions.

Finally, all page locks held by the current transaction are released in a single *post-commit* action, as demonstrated in the third procedure in Figure 5.2.

5.2.2 Modifications Log Handling at the Slave Nodes

This section discusses the way slave database nodes handle the transaction modifications log flushes when they are received from the master node.

In order to prevent flushes from interfering with readers who might currently be running on the slave node, each page is augmented with `VersionID` and `DiffQueue` fields. This is depicted in Figure 5.1. The `VersionID` designates the database version that the page content currently corresponds to. The `DiffQueue` is a singly-linked list with entries holding the physical modifications to the page data (diffs), which correspond to the evolution of the page, but have not yet been applied. The `DiffQueue` is ordered and is applied in an increasing version id order, because the modifications are incremental.

For example, consider Page 1 of Table X in Figure 5.1. If diff entry `V1` is applied to

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

Figure 5.3: Algorithm for handling the receipt of transaction flushes

the page, the page content will correspond to `V1` of the database. Since `V2` was produced after `V1`, it should only be applied after `V1` has been applied, which will then turn the page into `V2`.

Upon receipt of a *write-set* message from the master node, slaves 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 slave node reports to the master that it has successfully received and stored the diffs. Figure 5.3 depicts this process. The per-page mutex is required to ensure that no concurrent transaction running on the read-only node is accessing the diff queue at the same time.

The `DiffEntry` structure used in the pseudo-code is a plain linked list entry containing the data for that diff (in row-length encoding) and the database version that it will turn the respective page's content into.

5.2.3 Read-Only Transactions

Read-only queries should always read the most recent version of data items as determined by the scheduler and should not read data corresponding to different database versions. For multi-query read-only transactions, the scheduler tags all queries with the same version and reference counting is used to ensure that a page that has been read once does not get upgraded before the transaction completes.

The scheduler does the forwarding of read-only transactions in a load-balancing fashion, also striving to accommodate as many queries as possible having the same version on the same database. Figure 5.4 shows the algorithm that the slave nodes 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.

If the version requested is greater than the current version of the page, the reading transaction has several options, shown in lines 6 through 16 of the algorithm in Figure 5.4. The condition on lines 6 through 8 regulates whether the transaction may proceed with upgrading the page version. Clearly, if there are uncommitted transaction that are still accessing the page, this should not be possible. The conditions in lines 7 and 8 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 the page. This waiting is implemented by augmenting each page meta-data entry with a *condition variable*.

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 a newer version. Since versions of the pages are not kept around after they are applied, this causes

```
1  ReadPage(Tx, Page):
2      AcquireLock(Page.Mutex)
3      If (Tx.Version == Page.Version):
4          Page.RefCount++
5      Else If (Tx.Version > Page.Version):
6          If (Page.RefCount > 0 AND
7              NOT Page.DiffQueue.Empty() AND
8              Page.DiffQueue.Head().Version <= Tx.Version):
9              ReleaseLock(Page.Mutex)
10             Wait On Condition Until (Page.RefCount == 0)
11             GoTo Line 2
12         Page.RefCount++
13         For Each Diff in Page.DiffQueue Starting From Head:
14             If (Diff.Version <= Tx.Version):
15                 Page.ApplyDiff(Diff)
16                 Page.Version = Diff.Version
17                 Page.DiffQueue.DequeueHead()
18     Else:   // A version conflict occurred
19         ReleaseLock(Page.Mutex)
20         Return ERROR_ABORT
21     ReleaseLock(Page.Mutex)
22     Return SUCCESS
23 End.
```

Figure 5.4: Read-only transaction page access

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, so that it does not have to be aborted.

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, so subsequent reads from the same page need not acquire lock or test the versions.

Finally, when the read-only 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.

It should be noted here that the need to abort read-only transactions could have been avoided if the system kept around older versions of the same page. However, such a solution effectively converges to a *multiversion concurrency control* [14], which requires complex garbage collection mechanisms. In addition, it has been shown that using physical versioning in an in-memory database [37] leads to decrease in the performance [19] because of the poor cache utilization and the increased memory consumption. Instead, in our system, we optimistically rely on the assumption that typical e-commerce workloads should naturally exhibit low conflict rates.

5.2.4 Diff Queue Garbage Collection

As new versions are dynamically created, older versions are overwritten and diffs are discarded at each slave replica when necessary for the purposes of an on-going transaction. Since it is highly likely that the items being written by the master nodes will eventually be required by read-only transactions, no complex garbage collection algorithm for old versions is needed on the slave nodes in our system. However, in order to handle the

unlikely case of a data item being continuously written by the master but never read at a particular replica, we have implemented a periodic examiner thread. Upon activation, the thread inspects the list of active transactions and finds the one requiring the lowest version vector V_{min}. Then, it visits all the pages in the database and for each page, having version less than V_{min} applies all outstanding diffs from its queue up to and including V_{min}. This operation is safe, because the presence of transactions with version V_{min} guarantees that the scheduler will never produce a transaction with a lower version.

Since the *diffs* are generally very small in size and since the chance of a page being updated but never read is very low, the examiner thread need not be run so often. Given our experimental results, we expect that periods on the order of 30 minutes or more will be sufficient.

5.3 Support for Multiple Schedulers

Our scheduler is implemented as an event-driven loop that multiplexes requests and responses between the web/application server and the database replicas. It has been previously shown that, with this implementation, one can handle thousands of connections at a single scheduler [11].

Furthermore, the scheduler has built-in support for distribution, should the need arise for scalability reasons. To ensure database version number consistency across multiple schedulers, schedulers have to communicate during each transaction's commit. Upon receiving a new version number from the master database, at a transaction commit, the scheduler replicates this information in the memory of all other schedulers through a `replicate_end_transaction` message. The originator of the message waits for the acknowledgments before returning the response to the client. This simple scheme ensures that all schedulers tag any subsequent read-only queries that they distribute with the most recent version of the database that they need to see.

5.4 Stale Replica Reintegration

Section 4.3 presented an overview of the stale replica reintegration algorithm. In this section we present the lower level details and optimizations that the reintegration algorithm uses.

When a database needs to be started on a new node (S_{join}), with the intent to integrate it in the current workload handling, the database process has to be launched in *reintegration* mode. In order to do that, the user passes the network addresses of the scheduler, the current master and one slave node, called the *support* slave. After it starts, at the very first step, S_{join} looks for a checkpoint database image on its persistent storage. If such image is found, S_{join} *memory-maps* it and reads the page meta-data corresponding to this image. This meta-data specifies, for each page `P`, the database version, `P.Version`, that `P`'s content, `P.Data`, correspond to. `P.Version` can be `NULL` for a page if no contents for that page are present in the on-disk image. This may happen in cases where there is no checkpoint available for the page, e.g. the database has expanded in size since the time when the node failed. If the joining node is a completely new node, no database image will be found on its persistent storage. In such cases, S_{join} simply fetches the current database size, in number of pages, from the master node and creates an empty database image with all pages' versions set to `NULL`. This will later cause all pages to be updated.

After this initialization step, S_{join} contacts the master node and subscribes to its internal replication lists so that S_{join} can receive modification log flushes. Along with subscribing to the *master* replication lists, the joining node also obtains a value called `DBMinVersion`, which is equal to the latest database version that the master generated. The rationale behind this value is explained further in this section.

Figures 5.5 and 5.6 show the sequences that the joining node and the support slave execute to synchronize the database image. S_{join} enters the `JoiningState_Client` procedure, shown in Figure 5.5, and for each page `P` in its current image sends a request

```
1  JoiningState_Client(SupportID):
2      For Each Page P In The Database Image Do:
3          PRes = RequestPage(SupportID, P.ID, P.Version, DBMinVersion)
4          If (PRes Is NULL):
5              Continue
6          P.Data = PReq.Data
7          While (NOT P.DiffQueue.IsEmpty() AND
8                      P.DiffQueue.GetHead().Version <= PRes.Version) Do:
9              Diff = P.DiffQueue.DequeueHead()
10             P.ApplyDiff(Diff)
11             P.Version = Diff.Version
12 End.
```

Figure 5.5: Joining slave page upgrade loop

to the *support slave* containing the page id, its current version (`P.Version`) and the `DBMinVersion`. The *support slave* handles page requests by registering a handler function `JoiningState_Server` with pseudocode shown in Figure 5.6. Upon receipt of a page request, it checks whether the requested version, `PageVersion`, is greater than the version of the same page on the *support slave*. If this is not the case, then this page has *not* seen any modifications since the time S_{join} failed or was removed from the cluster, thus there is no need to transfer it. In any other situation, the page needs to be updated.

There are several precautions, shown in lines 6 through 16, that need to be taken by the *support slave* before it sends a page. Although S_{join} is subscribed to the *master* node's replication lists, all modification log records that it is going to receive as of that point will start from version `DBMinVersion + 1`. Since the application of these records is incremental, the page needs to include all modifications up to and including `DBMinVersion`.

This check and page upgrade is performed in lines 10 through 15 and occurs on a copy of the page so as to avoid interference and conflicts with currently running transactions. The page mutex needs to be acquired so that an atomic access to the page's `DiffQueue` is guaranteed.

After all database pages have been synchronized, S_{join} sends a message to the scheduler, informing it of its status. The scheduler, in turn, includes the new node in the active computation.

5.5 Optimization of Abort Rates

As mentioned before, in the cases of imperfect scheduler knowledge about a transaction's working set or insufficient number of replicas, a read-only transaction may need to wait for other read-only transactions, using a previous version of a page, to finish. In rare cases, a read-only transaction T_1 may need to be restarted if another read-only transaction T_2 first upgrades a shared page to a version higher than that required by T_1.

In order to keep the level of aborted read-only transactions to a minimum, we investigate the use of the scheduler algorithms described in this section. The scheduler tracks transaction conflict classes probabilistically, by recording active transactions that were running on a database node when an abort occurs. The scheduler is able to discern each read-only query type by parsing it and comparing the query string against a set of stored templates. Typically, the transaction types are repeatable, mostly consisting of single read-only queries. Over time, for each transaction pair X and Y, the scheduler collects the probability with which a transaction X aborts transaction Y if both are scheduled on the same machine and they request different versions of common data.

At the scheduler, we put an integer matrix $MConflicts[NUM\_TX][NUM\_TX]$, where NUM_TX is the number of different read-only transactions in the workload. Initially, all values in the matrix are zeroes. When a conflict error is returned to the

```
1  JoiningState_Server(JoinID):
2      While EOF Packet Is Received Do:
3          ReceiveRequest(JoinID, &PageID, &PageVersion, &DBMinVersion)
4          PCopy = PageMgr.GetPage(PageID).MakeCopy()
5          If (PCopy.Version > PageVersion):
6              AcquireLock(PageMgr.GetPage(PageID).Mutex)
7              If (PCopy.Version < DBMinVersion AND
8                  NOT PCopy.DiffQueue.IsEmpty() AND
9                  PCopy.DiffQueue.GetHead().Version <= DBMinVersion):
10                 Diff = PCopy.DiffQueue.GetHead()
11                 While (NOT Diff Is NULL AND
12                         Diff.Version <= DBMinVersion) Do:
13                     PCopy.ApplyDiff(Diff)
14                     PCopy.Version = Diff.Version
15                     Diff = Diff.Next
16             ReleaseLock(PageMgr.GetPage(PageID).Mutex)
17             SendPageData(JoinID, PCopy)
18             Destroy(PCopy)
19         Else:
20             SendPageData(JoinID, NULL)
21 End.
```

Figure 5.6: Support slave page transfer loop

scheduler from any of the slave databases, the error status message contains the page id where the abort occurred, the type of transaction that was aborted and the types of other transactions that were reading the page at the time the failure occurred. Then, if transaction of type Y caused transaction of type X to abort, we increment the value $MConflicts[X][Y]$ by one and also increment a counter called $Num\_Aborted\_Tx$, which indicates the total number of transactions that got aborted.

That, way after some reasonable running time, the value:

$$P(X\text{is aborted by}Y) = \frac{MConflics[X][Y]}{Num\_Aborted\_Tx} \tag{5.1}$$

reaches steady-state and represents the probability with which transaction of type Y will cause an abort, if scheduled to run on the same machine, where a transaction of type X, requiring lower version of the database than Y does is currently running. Thus, when a new transaction request comes, the scheduler uses the stored probabilities and additional information such as the list of outstanding transactions at each replica, their types and the version they access, in order to schedule a read-only query at a location where the probability of it causing an abort is the lowest. The scheduler algorithm thus attempts to find a good compromise between two potentially competing goals: accumulating non-conflicting transactions at the same replica, and preserving the load balance across replicas.

By default, the scheduler attempts to select a replica from the set of databases running read-only transactions with the same version number as the one to be scheduled, if such databases exist. If several replicas with a non-conflicting version are available, the scheduler selects any database based on load balancing. Load balancing across the database candidates uses a shortest execution length load estimate as in the SELF algorithm introduced in [11].

We further introduce three possible scheduler variations that differ in the policy for choosing a replica when the set of non-conflicting replicas is empty.

Lbal In the simplest case, the scheduler just picks the least loaded replica with a conflicting version, without taking into account the conflict probabilities.

Lbal+Oldest The scheduler attempts to minimize the overlap of conflicting transactions in terms of time. It picks the replica where, if any potentially conflicting request is running, any such transaction has been running for the longest time compared to other replicas.

Lbal+n The scheduler attempts to decrease conflicts by restricting the number of concurrently required versions at each replica to n. The scheduler picks a replica where conflicting queries use versions that are at most n versions older than the version of the incoming query. If the non-conflicting set is still empty, the scheduler resorts to *Lbal+Oldest*.

Subsequent chapters give details on the performance of this implementation of the *Dynamic Multiversioning* algorithm.

Chapter 6

Experimental Methodology

In this section we provide details about the experimental environment and methodology we use for evaluating the performance of the *Dynamic Multiversioning* replication system.

6.1 TPC-W Benchmark

We evaluate our solution using the TPC-W benchmark from the Transaction Processing Council (TPC) [40]. The benchmark simulates an on-line bookstore.

The database contains eight tables: customer, address, orders, order_line, credit_info, item, author, and country. The most frequently used ones 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 used a PHP [6] implementation of 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 store's home page, listing of new books and bestsellers, requests for book details and three interactions, involving searches, based on the book's title, category or author. The update interactions include new user registration, updates to the shopping cart, two order-placement transac-

tions and one used for store administration purposes. 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.

We use the standard inventory size with 288000 customers and 100000 books, which results in a database size of about 600MB. The memory-resident set of the workload is about 360MB and it consists of the most-frequently accessed sections of the database. The inventory images, totaling about 180MB, reside on the web servers. The complexity of the client interactions varies widely, with interactions taking between 20 and 700 milliseconds on an unloaded machine. The complexity of the queries varies widely as well.

We use the three workload mixes of the TPC-W benchmark: browsing, shopping and ordering. These workloads are characterized by increasing fraction of writes. The least write-intensive workload is the browsing mix, which contains 5% update transactions and 95% read-only queries. The most commonly used workload is the shopping mix with 20% of updates and it also is considered to most-closely represent a real-world scenario. The most update intensive workload is the ordering one which contains 50% updates and it is indicative for shopping-spree periods such as Christmas and other holidays.

6.2 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 [23], which emulates client interactions as specified in the TPC-W document. In order to fully stress the database-side operations

of the benchmark, we configure our client emulator not to request inventory images, which are served by the web server and do not load the database.

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. To determine the peak throughput for each cluster configuration we run a step-function workload, whereby we gradually increase the number of clients from 0 to 1000. We then report the peak throughput in web interactions per second (shown as WIPS on the charts in the subsequent chapter), the standard TPC-W metric, for each configuration. At the beginning of each experiment, the master and the slave databases memory-map 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 `REPLICATED_HEAP` table format before initial memory-mapping. We run each experiment for a sufficient period of 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.

Chapter 7

Results

In our evaluation, we first perform preliminary experiments measuring baseline factors that influence scaling such as the ratio of read versus write transaction costs and their access pattern in our experimental system (Section 7.1). Next, we show the performance benefits brought about by our fast in-memory transactional layer compared to a stand-alone on-disk InnoDB database, in Section 7.3. Assessment of the read-only query version conflict rates experienced at the slave nodes is shown in Section 7.3.2. Finally, we demonstrate fast reconfiguration under failures in our in-memory tier versus a stand-alone InnoDB replicated tier in Section 7.4.

7.1 Preliminary Experiments

The goal of this experiment is to derive and compare the ratio of write versus read interactions in each of the Ordering, Shopping and Browsing mixes of the transactional workload. We report results for two different systems: an on disk InnoDB database and our in-memory multiversioning scheme. We also measure the cost of the basic overhead in our in-memory system, the increase in the execution time of write transactions, caused by the flushing of the modifications log from master to slaves.

For these purposes we run a workload session with only one emulated client, which

sequentially submits 50000 TPC-W requests to a single database instance. The client sums the perceived response time for each web interaction that it simulates, classifies the interactions into read-only and update, and at the end of the experiment, outputs the aggregate time it spent waiting for the read-only and update interactions, respectively, to complete. We divide these two numbers by the total number of interactions that executed during the experiment (i.e., 50000) and refer to the resulting number as the "cost" (or "weight") of the read-only or update interactions in the respective workload mix. The sum of these two costs denotes the total cost for execution of the workload. Figure 7.1 depicts our findings, normalized to the total cost of the Ordering workload mix with the InnoDB database.

The database indexes are chosen so that optimal single database performance is achieved. The white part of a bar corresponds to the cost of write interactions and the black part to the cost of read-only interactions. The *Ord*, *Shp* and *Brw* abbreviations in the bar captions specify which workload mix the respective bar corresponds to. The numbers in the brackets denote the ratio of write versus read cost for the particular configuration.

The first group of three bars corresponds to a master with no slaves configuration using the *dynamic multiversioning* in-memory engine. Since there is only one replica, no update log flushes occur. The second group of bars shows the query durations when all requests are still handled by the master, but in addition, upon commit, the master flushes the modifications log for each update transaction to 8 slave replicas and waits for them to acknowledge. The last three bars correspond to the case when a single InnoDB database is used as a storage manager for the same database schema.

We can see from the graph that the cost of write interactions is almost two times higher in our in-memory system as compared to InnoDB. This discrepancy is caused by the increased cost of index update operations in MySQL with heap tables combined with the cost of the page faults used by the underlying TreadMarks run-time system, we use to

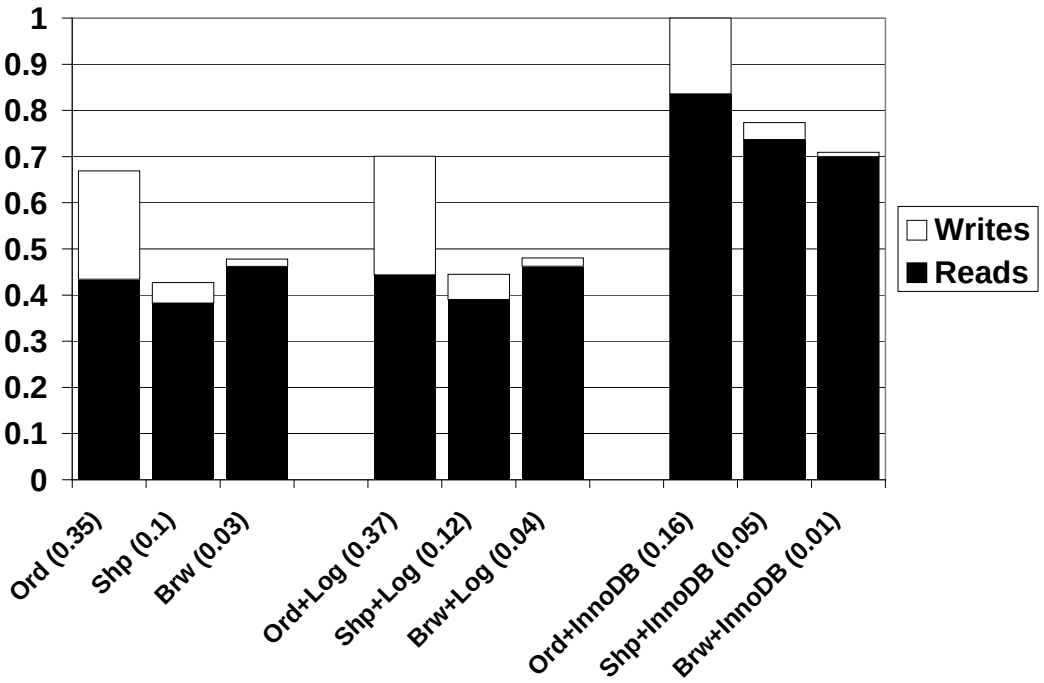


Figure 7.1: Workload execution costs, normalized to the total cost of the InnoDB ordering workload mix

collect the modifications log. Since our system builds upon the original MySQL *HEAP* table engine, we reused its *RB-Tree* [20] 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, compared to a B^+-*Tree* index, which has relatively infrequent node splits and merges. Thus, update transactions become heavier when RB-Tree indexes are used as opposed to InnoDB, which uses a B^+-*Tree* index structure.

Comparing the first and second group of bars in Figure 7.1 and the respective write parts for the Ordering, Shopping and Browsing mixes in the *Dynamic Multiversioning* system, it can be seen that the overall modifications log flush overhead is minimal in our system. This comes about, because slave replicas reply immediately after receiving the log, without actually applying it and thus delaying the master.

From the figure it can also be seen that the relative cost of the read-only fraction of the workload is significantly higher than that of the update part. Update interactions usually modify/insert a small number of rows in the table and the rows being modified are always located using the primary key. The costliest part of an update transaction is the index re-balancing. On the contrary, read-only interactions involve searches and complex joins on multiple big tables. Since there is no unique index configuration, which suits all read-only queries, some of them result in full table scans and sorting of the results.

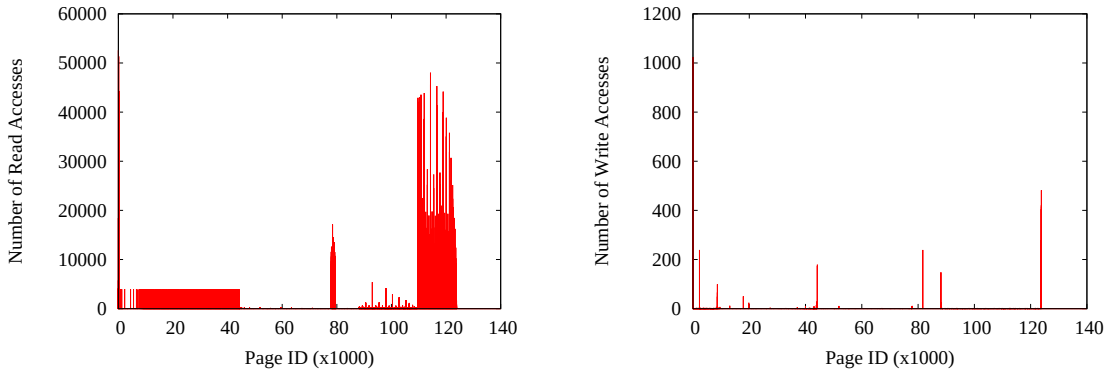
7.2 Data Access Pattern

In order to validate our hypothesis that e-commerce workloads, such as the one modeled by TPC-W, exhibit high degree of locality, we instrumented the storage engine code to count the number of transactions that read and write each of the database pages. Figure 7.2 shows the read and write access patterns that we obtained running the browsing mix with the standard TPC-W database size having 288000 customers and 100000 items. The X-Axis in both charts shows the respective page id in the database image and the Y-Axis corresponds to the number of transactions which accessed that page.

The chart in Figure 7.2(a) shows a very high number of localized accesses for pages belonging to the *items* and *orders* tables and a relatively uniform read-only activity for the remaining pages. The pattern of the update transactions, shown in Figure 7.2(b), displays very high locality as well, consisting mostly of updates to the *order_line*, *item*, *customer* and *shopping_cart* tables.

7.3 Performance Experiments

Figure 7.3 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



(a) Read pattern

(b) Write pattern

Figure 7.2: TPC-W page access patterns (288K customers, 100K items)

concurrency control. We performed experiments with 1, 2, 4 and 8 slave replicas respectively. We see that the system throughput effectively scales close to linearly, even with large cluster configurations.

Overall, we improve throughput performance over a stand-alone InnoDB database by factors of 6.5, 17.6 and 14.6 in our largest configuration with 8 slaves for the ordering, shopping and browsing mixes, respectively. Furthermore, we can see that a performance jump from the throughput of a single InnoDB database 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. The poor scaling of the ordering mix is caused by saturation of our master database with update transactions. This saturation stems mainly from the costly index updates in our system (see Section 7.1) and also includes lock contention on the master. We attribute the superlinear scaling from 4 to 8 slave databases, obtained for the browsing mix, to a better CPU cache utilization at the slaves. Our conjecture is that better cache utilization gets brought by the fact that more queries, with the same access pattern, get concentrated on the same replica, which facilitates cache line reuse.

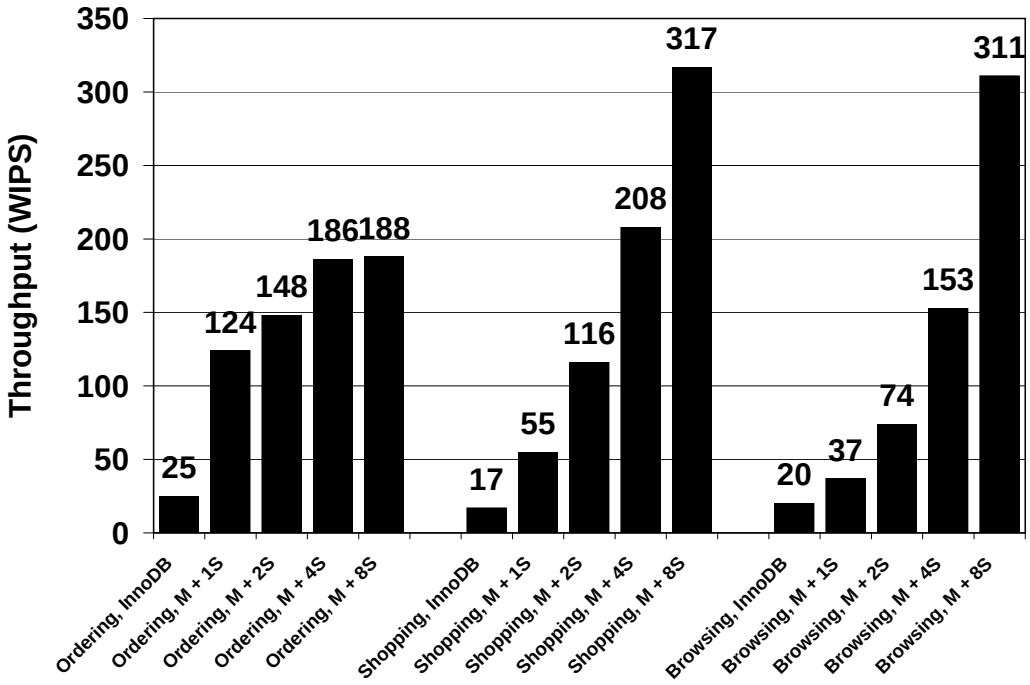


Figure 7.3: Throughput performance comparison of the different Dynamic Multiversioning configurations against a single on-disk InnoDB database

7.3.1 Scalability for the Configuration without Indexes

In another experiment, we validate the scalability of the dynamic versioning algorithm with a different update to read-only workload cost distribution and different application access patterns. In particular, we ran the scaling and baseline experiments using a TPC-W database setup, where the performance-critical indexes have been removed. Specifically, the schema only contains the primary keys for each of the relations. This causes the search and bestseller queries to perform full-table scans on the *items* and *author* tables, and slightly increases the duration of the user registration update transaction. Figure 7.4 shows the ratio of update to read-only workload costs in this configuration. It can be seen that once the performance-critical indexes are removed, the ratios decrease almost twice, as compared to the fully-indexed database setup (compare Figure 7.4 against the first three bars in Figure 7.1). Decreased ratio of update to read-only workload costs means that read-only queries have become lengthier that the updates on average. This is

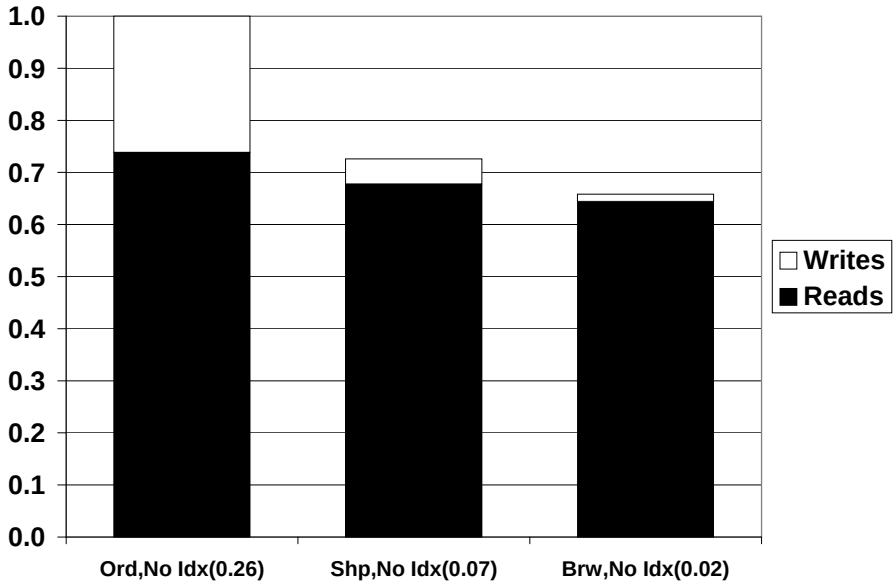


Figure 7.4: Workload execution costs in the configuration without indexes, normalized to the total cost of the ordering mix

favorable for the scaling capabilities of our system, because the master node will saturate with updates at a much slower pace. On the other hand, the side effect of the full-table scans over the database is a worst-case scenario, because it increases the chances for version conflicts (see Section 5.2.3).

Running the throughput scaling experiment with the modified database configuration yielded the scalability results shown in Figure 7.5. These results show almost linear scalability for all the workload mixes and again show the impracticality of the *RB-Tree* used as a database index structure.

7.3.2 Version Conflicts

In any optimistic concurrency control scheme, the number of transactions aborted, because a non-serializable condition has been reached, is mainly affected by the following three factors: i) the frequency of update transactions, ii) the extent to which update transactions write to hot-spots in the data [39] and iii) the duration of each transaction. In order to avoid keeping multiple versions of the same page on the slave replicas, *dy-*

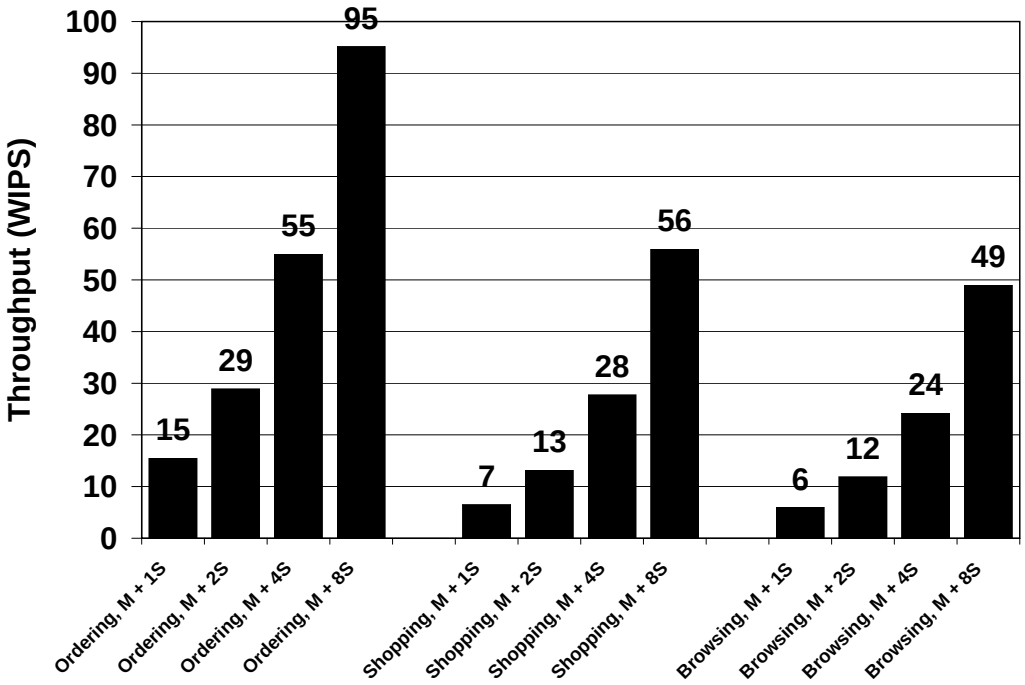


Figure 7.5: Throughput scaling in the database configuration with no indexes for the browsing, shopping and ordering TPC-W mixes

namic multiversioning uses an optimistic assumption that on average, read-only queries seeking *different* versions of the database will either have disjoint sets or they will be running on different replicas. Because of this assumption, the above three factors are also applicable to it. In particular, Figure 7.2 shows the access locality and the high number of update transactions, writing to localized spots of the database, which are at the same time subject to heavy read activity. Since this access pattern represents a situation, which has a negative effect for an optimistic concurrency control scheme, one might expect that the level of version conflicts with *dynamic multiversioning* should be high on average, all other things being equal. However, in our case, there is an advantage gained by the presence of multiple replicas in the cluster, permitting several data item versions to exist simultaneously on different nodes. The presence of multiple data versions mitigates the conflicts level, similarly to the way a *multiversioning* concurrency control database does [36].

Thus, in summary, our conjecture is that the level of version conflicts in the *dynamic*

# of Slaves	Ordering	Shopping	Browsing
1	1.15%	1.44%	0.63%
2	0.35%	2.27%	1.34%
4	0.07%	1.70%	2.34%
6	0.02%	0.41%	2.68%
8	0.00%	0.22%	2.83%

Table 7.1: Level of aborts encountered due to version inconsistency (indexed configuration)

multiversioning system should be dependent on the combined effect of the following factors: i) the frequency of update transactions, ii) the duration of read-only queries, iii) the number of slave replicas and iv) the algorithm used to schedule read-only queries on the set of available slaves.

Version Conflict Frequencies

Table 7.1 shows the average number of read-only queries that had to be re-issued during the test run due to version inconsistency. The numbers are presented as a percentage of the total number of queries that executed during the test session with a fully-indexed TPC-W database and are representative for the base case; e.g., when no special scheduling policy is applied.

We see that for the ordering and shopping mixes, the level of aborts tends to decrease with the number of slave replicas, which is in line with our conjecture about the factors determining it, in the *dynamic multiversioning* scheme. The probability that queries seeking different database versions hit the same replica and need to be restarted declines with more replicas until the abort rate at 8 slave databases is close to 0% for both the ordering and the shopping mixes. In the browsing mix, however, we observe that the abort rate is slightly higher (2.83% at 8 replicas). Looking at the probabilistic conflicts

# of Slaves	Ordering	Shopping	Browsing
1	4.75%	2.05%	0.87%
2	3.83%	2.23%	1.50%
4	3.46%	3.90%	2.28%
6	2.88%	3.59%	1.74%
8	2.47%	3.48%	1.53%

Table 7.2: Level of aborts due to version inconsistency (non-indexed configuration)

classes matrix $MConflicts$, maintained at the scheduler (see Section 5.5), we found that *BestSellers* queries abort each other with the highest probability. The *Bestsellers* query is the most complex read-only transaction in the TPC-W benchmark. It consists of two statements and performs join operation on three tables: *order_line*, *item* and *author*. The first two of these tables are modified by virtually all update transactions, so the potential for conflicts on pages belonging to them is high. Since the *bestsellers* queries i) take several seconds to execute on average under load, ii) access vast amounts of data, and iii) are more frequent in the browsing mix, they are also the ones most likely to interfere. Here, the negative effects of update transaction frequency and query duration outweigh the benefits brought by the presence of multiple replicas in the cluster and the level of aborts starts to increase. Further in this section we show how we are able to mitigate this effect, by using the scheduling policies introduced in Section 5.5.

Table 7.2 lists the abort rates for the TPC-W database configuration without the performance-critical indexes. As discussed in Section 7.3.1, without these indexes, the update transactions are much shorter, compared to the read-only ones, and the read-only queries perform full-table scans on tables that are frequently modified. Thus, the version conflicts effect is most clearly observable with the ordering mix, because it generates, on average, one update per read-only transaction. This causes an increased number of queries, each accessing similar data, but requesting different database versions to

accumulate on the same node, which makes occasional conflicts inevitable. However, it can be seen in the table that as the number of slave replicas increases, the number of conflicts goes down.

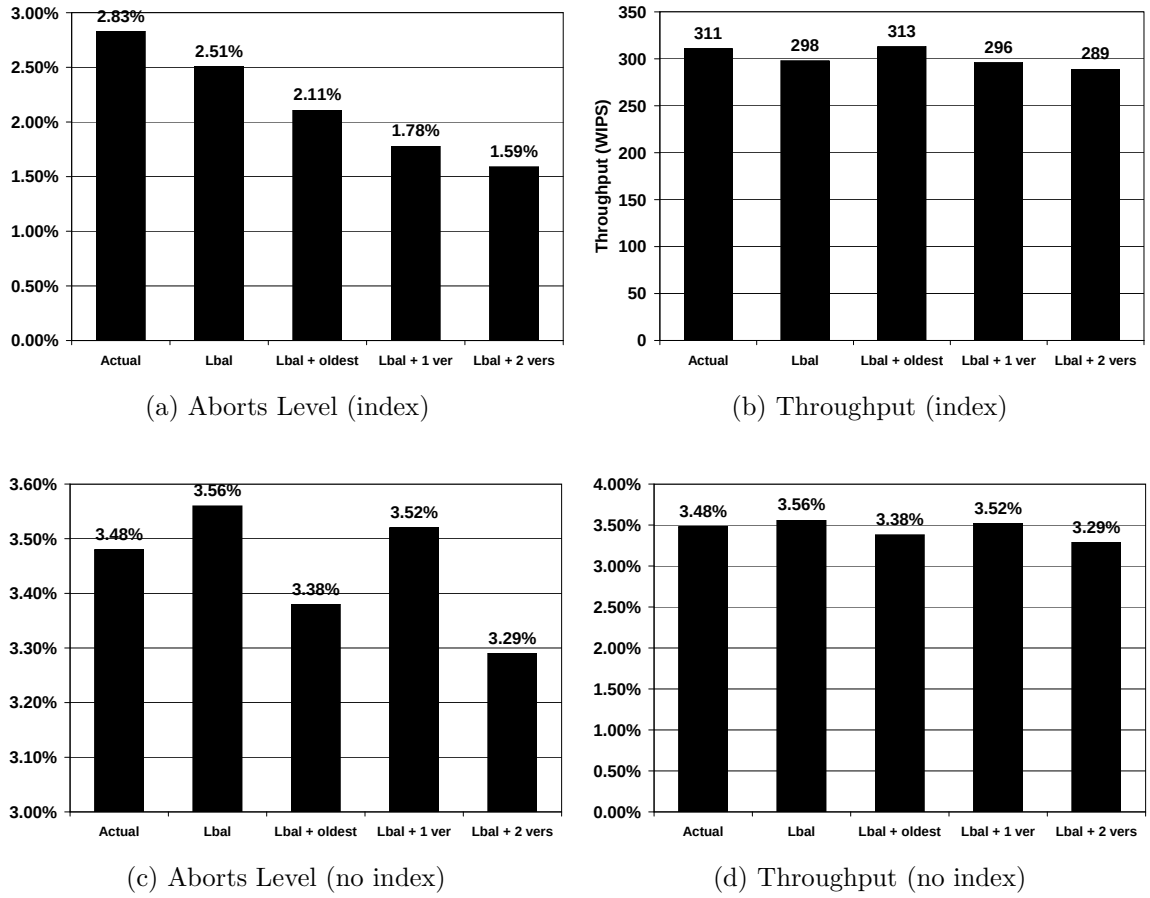
For the shopping and browsing mixes, the number of aborts increases up to the 4-slave cluster configuration after which it gradually declines. We attribute this to a reached abort level "peak" between two competing effects - the per-database update to read ratio and the number of slave replicas.

Version Conflict Optimizations

Following the observation that bestseller queries are the cause for most of the version conflicts in the system, we implemented a *bestseller* specialization of the automated conflict detection algorithm described in Section 5.5 and tried to decrease the abort levels for the browsing mix with a fully-indexed database schema and the shopping mix with a lightly-indexed schema, both with the 8-slave cluster configuration.

Figures 7.6(a) and 7.6(c) show the results obtained when the scheduler employs these techniques for reducing the abort rates, based on probabilistic conflict classes. The first bar in both figures, *Actual*, represents the base level of aborts with a conflict-oblivious scheduler that simply load balances read-only requests across all database replicas. The remaining bars reflect the level of aborts corresponding to our scheduler optimizations. In all techniques, the scheduler sends requests to a replica with a non-conflicting version, if available. When the set of non-conflicting replicas is empty, `Lbal` picks the least loaded replica, `Lbal+Oldest` picks the replica with the longest running conflicting query and `Lbal+1`, `Lbal+2` restrict the number of concurrently run conflicting versions to a maximum of 2 and 3, respectively.

We can see from the figure that for the browsing mix, using the above heuristics decreases the aborts level down to 1.59% representing almost a factor of 2 reduction in the best scheme, `Lbal+2 vers`, compared to a conflict oblivious scheduler. In all cases,



(a) Aborts Level (index)

(b) Throughput (index)

(c) Aborts Level (no index)

(d) Throughput (no index)

Figure 7.6: Decreasing the level of aborts for the browsing and shopping mixes

the additional heuristics are necessary and actually used at run-time due to the lack of non-conflicting replicas.

Figures 7.6(b) and 7.6(d) also show, that the reduction in abort rate does not necessarily correlate with an increase in throughput due to the conflict versus load balancing trade-off. However, the overall throughput is not significantly affected.

The results in this section demonstrate that the level of conflicts in the *dynamic multiversioning* scheme is always kept within a tolerable range. Thus, we expect our system to be suitable for any database size and read-only transaction access pattern.

7.4 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 warm-up. 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.

7.4.1 Fault Tolerance with Node Reintegration Experiment

In this section, we evaluate the performance of the node reintegration algorithm we introduced in Chapter 4. The algorithm permits any failed node to be reallocated to the

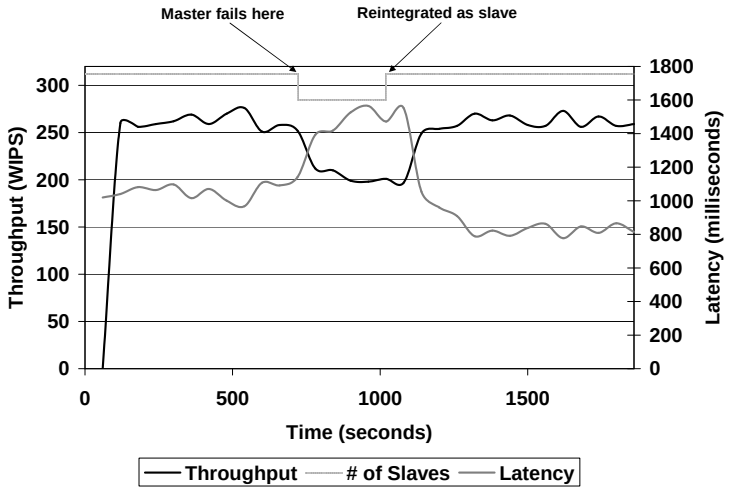


Figure 7.7: Node reintegration

workload after recovery. This implies a period of node down-time (e.g., due to node reboot). For the purposes of this experiment we used the smaller standard database having 288000 customers and 10000 books. The rationale behind this is to make transaction execution times shorter and thus have more accumulated updates during the period of downtime.

We use the master database and 4 slave replicas in the test cluster configuration. Figure 7.7 shows the effect of reintegration on both throughput and latency. The top line in the chart serves as a guide showing the period corresponding to the number of slave replicas at each point of measurement.

We consider the most complex recovery case, namely that of master failure by killing the master database at 720 seconds by initiating a machine reboot. We see from the graph that the system adapts to this situation instantaneously with the throughput and latency gracefully degrading by 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.

7.4.2 Failover Experiments

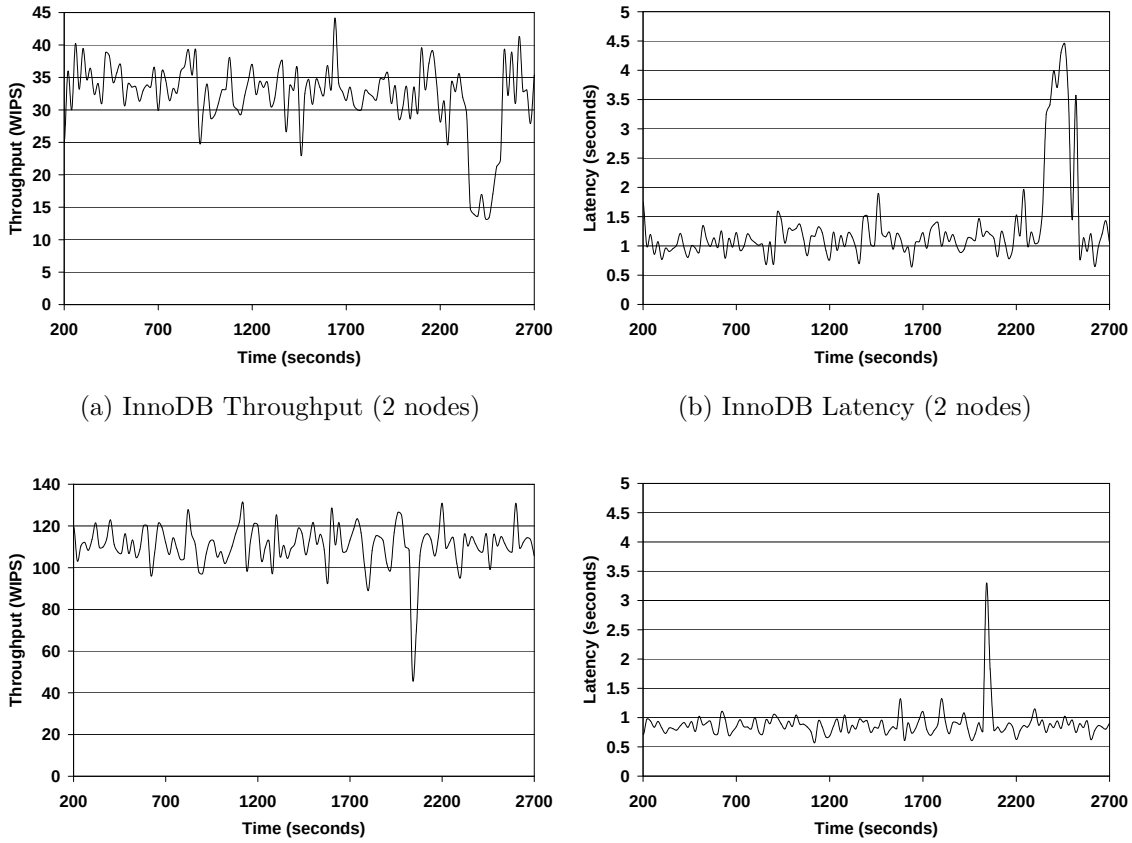
In this section we evaluate the performance of our automatic reconfiguration using fail-over on spare back-up nodes. In all of the following experiments we designate several databases as backup nodes and bring an active node down. The system immediately reconfigures by integrating a spare node into the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



(a) InnoDB Throughput (2 nodes)

(b) InnoDB Latency (2 nodes)

(c) DMV Throughput (M + 2S)

(d) DMV Latency (M + 2S)

Figure 7.8: Failover onto stale backup: comparison of InnoDB and DMV databases

for state-of-the-art replicated solutions where asynchrony is used for scaling a workload to a number of replicas. In all log shipping schemes, the primary database processes transactions and periodically sends the redo log to the backups, which apply it either synchronously or asynchronously, dependent on the desired data availability guarantees that the system provides. There is a trade-off here between the frequency of log transfers and speed of recovery. Frequent log transfers cause interference with database operation in the fault-free case, whereas sporadic ones cause long recovery times due to the need to replay a long history of updates.

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 [9] and both process read-only queries. The spare node is updated every 30 minutes.

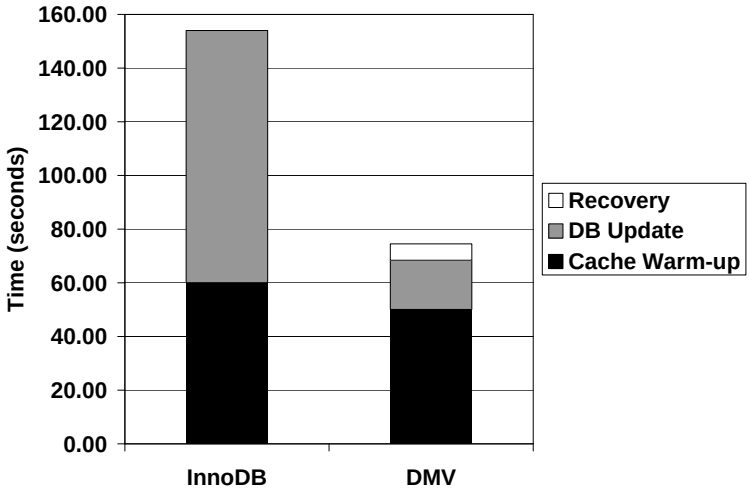


Figure 7.9: Failover stage weights: cleanup (Recovery), data migration (DB Update) and Buffer cache warm-up (Cache Warm-up)

Figures 7.8(a) and 7.8(b) show the failover effect in an experiment where we kill one of the active nodes after 30 minutes of execution time. In this situation, all the redo log from the active nodes needs to be replayed on the backup that takes over. 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 *Dynamic Multiversioning* tier, having a master, two active slaves and one 30 minute stale backup. We used two active slaves, because 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, so that we also reflect the time needed to abort possible partially propagated transactions. The results are presented in figures 7.8(c) and 7.8(d). In this case, the total failover time is about 70 seconds, less than half 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 7.9, we can see that most of the fail-over time in our in-memory Dynamic Multiversioning 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 the subsequent master reconfiguration.

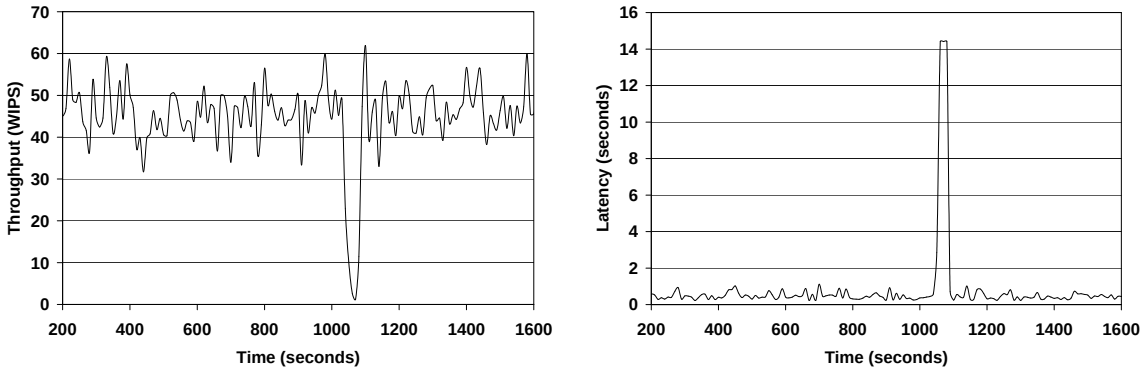
Up-to-date Cold Backup

In this suite of experiments, the spare node is always kept in sync with the rest of the system by sending it the modifications log. In this case, the failover duration will be affected by the size of the buffer-cache and the amount of time it takes to warm it up.

In order to emphasize the buffer warm-up 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.

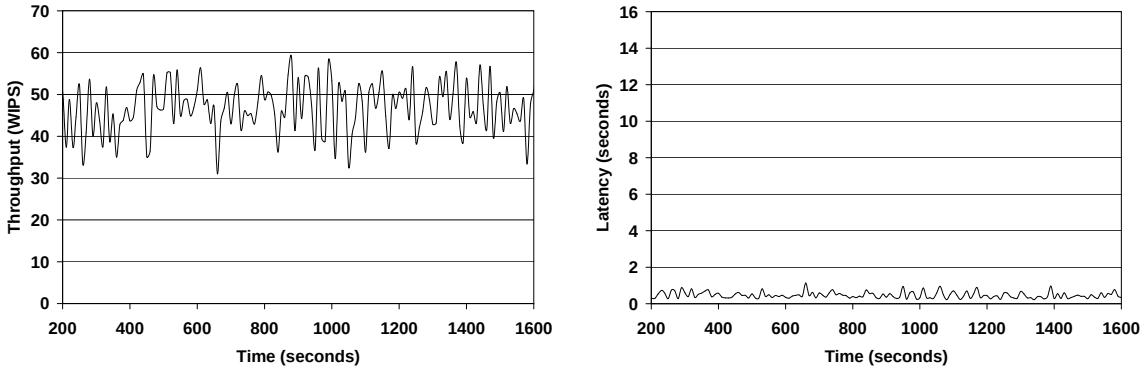
In this first experiment, the buffer cache of the spare node is cold, so upon fail-over the database needs to swap-in a significant amount of data, before achieving peak performance. We run the TPC-W shopping mix and after approximately 17 minutes (1030 seconds) of running time, we kill the active slave database forcing the system to start integrating the cold backup. Figure 7.10 shows the client-perceived throughput and latency for the duration of the cold backup experiment.

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.



(a) Throughput (b) Latency

Figure 7.10: Failover onto cold up-to-date DMV backup



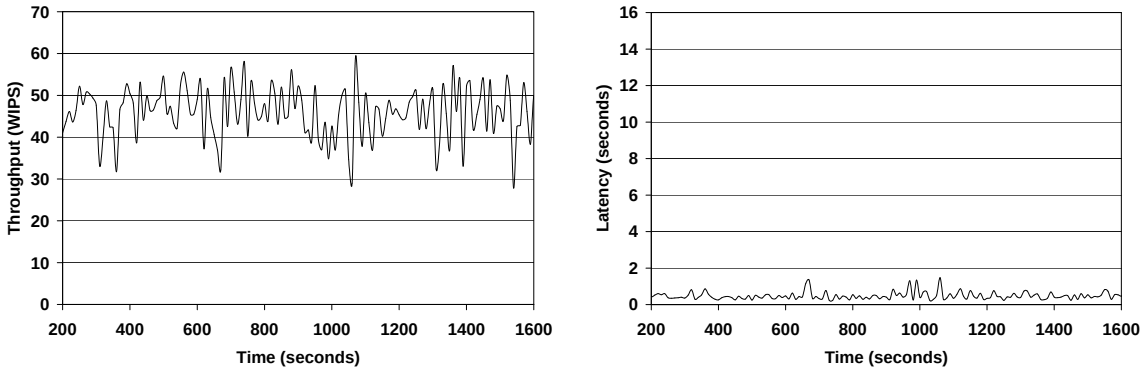
(a) Throughput (b) Latency

Figure 7.11: Failover onto warm DMV backup with 1% query-execution warm-up

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 7.11 shows the throughput and latency 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 7.12: Failover onto warm DMV backup with page id transfer

Although executing a small fraction of the read-only workload seems a promising solution, it may not be a proper choice for certain situations. For example, if the spare node is used by a different workload and the read-only transactions are computationally intensive (e.g., do joins or sorting of huge datasets), using the above scheme could potentially contend with the alternate workload on the CPU resource. For that reason, we implemented a second solution, whereby a designated slave node collects statistics about the access pattern of the resident data set of the primary workload and sends only the ids of the pages found in the buffer cache to the backup after each 100^{th} transaction. The backup only touches the pages so that they are kept swapped into main memory.

Figure 7.12 shows the failover effect for our alternate backup warm-up scheme using page id transfers from an active slave. We see that the performance in this case is the same as that for periodic query execution allowing for seamless failure handling.

Chapter 8

Related Work

A number of solutions exist for replication of relational databases that aim to provide scaling, strong consistency and easy reconfiguration in response to failures. In this chapter we give a survey of these schemes, describe their benefits and drawbacks, and relate them to the *Dynamic Multiversioning* system presented in this dissertation.

Previous work in the area of primary-backup replication [12, 42] has mostly followed a "passive backup as a hot-standby" approach where the backup simply mirrors the updates of the primary. These solutions either cause 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 [12, 42] 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.

Recently, a number of solutions have been developed that try to mitigate the problems associated with *passive backup* replication. They range from industry-established ones, such as the Oracle RAC [5] and the IBM DB2 HADR suite [2], to research and open-source prototypes, such as Distributed Versioning [9], C-JDBC [17, 18], Postgres-R [29]

and Ganymed [36].

The industry solutions provide both high availability and good scalability, but they are costly and require specialized hardware such as the Shared Network Disk [5]. The research prototypes use commodity software and hardware, but they either have limited scaling for moderately heavy write workloads due to their use of coarse-grained concurrency control implemented in a dedicated request scheduler node [9, 11, 17], or sacrifice failure transparency and data availability by introducing single points of failure [36]. Even the scheduler-based solutions that do offer transparent fail-over and data availability do so by means of complex protocols due to the crucial data that resides inside the scheduler tier [11]. 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.

The most successful *active backup* solutions, providing strong consistency database replication, try to circumvent the problems associated with purely lazy or purely eager replication [24] by combining beneficial features from the two approaches. Since these approaches are closest to our work, we will present them in more detail in the subsequent sections. For better organization, we will classify these methods into two major categories - group communication-based and scheduler-based ones.

8.1 Group Communication Methods

Most solutions implementing this class of replication methods use *group ownership* of the data items combined with *eager* update propagation model. The implementation of eager replication in these schemes uses *Reliable Broadcast* [15] primitives to enforce a total execution order of transactions. The main factor which affects performance in this case is communication latency caused by an increased number of message exchanges, proportional to the number of nodes in the system. The fault-tolerance guarantees that

these methods provide depend on the specifics of the communications library used.

Postgres-R

B. Kemme et al. [29] proposed a scheme for an "update anywhere" replication of transactional workloads. Replication happens through reliable broadcasts from the node that executed an update transaction to all the nodes in the system. The serialization order of update transactions is enforced by the global message delivery order guaranteed by the group communication library. This implies that a node, which executes an update should do so locally, on a shadow copy, and then broadcast the results to all replicas (including itself) in order to make them permanent and consider the transaction committed.

The system extracts per-row write sets from each update transaction using Postgres [7] triggers and submits them to the replicas by generating an SQL query which modifies only the changed tuples. Write conflicts between a locally running update and an update coming from another node are resolved by aborting one or more of the conflicting transactions. Since a transaction is not visible outside of the node before it commits and the tentative execution is performed on a shadow copy, this scheme provides good fault-tolerance for node crashes.

As set of experiments, performed with a simple synthetic workload and database schema, shows very small communications overhead and a good scale-up. However, with a more realistic workload, having both read and write transactions, the level of conflicts caused by the update-anywhere nature of the system becomes significant. Another potential cost of this scheme is the need to maintain shadow copies of data items, which incurs copy-on-write overhead in the database and requires occasional garbage collection.

Pronto

The *Pronto* protocol, by F. Pedone et al. [34], is a hybrid replication system using *master* ownership of the data items. In the system, one primary database executes the update

and then, using reliable broadcast primitives, broadcasts the transaction's statements to the backups, which execute them locally. The total order imposed by the group communication library dictates the order in which statements are to be scheduled for execution at the replicas. This scheduling order guarantees that the global execution will result in 1-copy serializability.

In order to provide fault-tolerance and transparent failover, the protocol requires that the client module has access to all the statements of the transaction. If a failure occurs, the client engine simply replays the entire transaction on another node. Failure detection and primary re-election is achieved at the level of replicas, using the notion of *epochs*. When a replica suspects that the primary (master) node has failed, it sends an epoch change packet which causes a new primary to be elected. The protocol guarantees that the effects on the data will be as if in each epoch there was only one primary node to process updates.

To the best of our knowledge, there is no implementation of a real system which uses this protocol. In addition, the reliable broadcast communication *Pronto* depends on uses an average of $4(n-1)$ messages per transaction, where n is the number of nodes. The scalability of such system in practice is unclear.

8.2 Scheduler Methods

This class of systems relies on the presence of a dedicated *scheduler* node, which coordinates the replication process. The existence of a central view-point in the system permits various optimizations to be implemented. Drawbacks of these scheduler-based methods are the duplication of database functionality and the possibility that the scheduler might become a single point of failure.

Conflict-Aware Scheduling

Conflict-aware scheduling is a solution for scaling the back-end database of dynamic-content web-sites [11]. The scaling is achieved through full replication of the database state by executing updates on all nodes in the cluster. The system performs replication in a lazy manner and uses a scheduler to track the progress at each replica, so that 1-copy serializability guarantees are provided. The technique requires that transactions specify the tables they access and their mode of access (read or write) before any other statement. Thus, the scheduler handles conflicts at the granularity of a table. The replication protocol uses *read-one-write-all-available* principle combined with strict 2-phase locking to provide 1-copy serializability. Read queries from a transaction are executed at replicas, where previous dependent write transactions have finished. This scheme permits high scalability due to the relieved requirement to wait for all updates to complete on every node.

The proposed scheme maintains significant amount of state at the scheduler. This includes status and locking queues, and out-of-order execution queues for queries whose tags come out-of-order. Coordination of the replication also depends on a sequencer that tags transactions with sequence numbers, used to enforce the serialization order. These characteristics of the system produce the need for the complicated fault-tolerance and high availability protocol that has been presented.

Performance experiments show close to linear scaling for the three TPC-W mixes, plus low overheads in the failure-free case of execution. However, no experiments are performed to evaluate the cost of re-configuration in case of failure and in particular, when the scheduler fails.

Distributed Versioning

The *distributed versioning* [9] is an improvement of the conflict-aware scheduling protocol. It overcomes a limitation of the earlier approach, which required conservative 2-phase locking at table granularity. The new approach employs the notion of a table version. As in the previous scheme, each incoming transaction is required to start with a pre-declaration of the tables that it is going to access and the mode of access. Each transaction's update access of a table increments the current version of the table. If two transactions have conflicting operations on certain tables, then the distributed versioning scheme will ensure that the table versions of the shared tables, that the first transaction accesses will be strictly lower than those of the second transaction. The transaction execution is ordered in an increasing order of the versions.

The new scheme also introduces the notion of early releases. The transaction developer may give a hint to the system (by placing a special statement in the transaction body), where is the last use of a table. In such cases, the table version produced by this transaction is released earlier, which permits transaction statements which depend on this table to proceed. In both this and the previous work, the scheduler effectively takes the role of the database concurrency control engine. That way, these schemes do not utilize the capabilities of low-granularity concurrency control techniques available in contemporary off-the-shelf and open source databases.

Ganymed

Compared to the distributed versioning scheme, *Ganymed* by C. Plattner et al. [36] eliminates the duplication of database functionality and the need to re-execute full SQL updates on each replica. Similarly to our scheme, it uses the notion of master and slave nodes. The master executes update transactions, whereas slave replicas apply write-sets produced by the master in a lazy fashion and execute the heavier read-only queries.

Thus, readers are never blocked by incoming write requests, but it comes at the cost of copy-on-write and the need for garbage collection of older versions.

Ganymed uses a scheduler with a slightly more complex role, which categorizes incoming transactions into read-only and update ones and extracts and distributes write sets of updates running on the master. In order to ensure that queries always read the latest state of the database, the scheduler of *Ganymed* tags each of them with a global database version number. When updating the replicas, scheduler keeps track of the database version numbers of the replicas. Read-queries are blocked at the scheduler until the respective database version number they should see becomes available. However, since the system is not integrated with the database concurrency control engine, as in our scheme, it is unable to enforce strict versions. Thus, it provides *snapshot isolation* [13] as opposed to *conflict serializability*.

Although no experimental results have been presented for this work, these tasks and the internal state the scheduler keeps still add complexity enough to make the provision of fault-tolerance and automatic reconfiguration hard.

Chapter 9

Conclusions

In this thesis, we introduced a suite of novel lightweight scaling and 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 guarantees and ease of reconfiguration at the same time.

Dynamic Multiversioning offers high concurrency by exploiting the data versions naturally arising 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. With the utilization of a "roll-forward" per-page diff queue, we avoid the copy-on-write overhead associated with systems that use stand-alone database multiversioning to offer snapshot isolation. In order to do this, we apply an optimistic approach in the version management, which assumes 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. We demonstrated that the use of a clever scheduling scheme helps the system to keep the aborts due to version conflicts at negligible rates.

Our evaluation showed 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 or variations in the load. We improved the performance of a dynamic-content web site, which uses an InnoDB on-disk database back-end, 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 demonstrated 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.

The per-page granularity of our integrated concurrency control and replication algorithm is not an inherent limitation of the *Dynamic Multiversioning* system. The proposed algorithm applies, without significant modifications, to databases, which use lower levels of granularity, e.g. table row. The only requirement that the scheme imposes is that the unit of replication needs to be the same as that of the database concurrency control and that the storage manager must use two-phase locking to control parallelism, so that only committed values are propagated to the slave nodes. This requirement is readily fulfilled by modern databases, such as MySQL, DB2 and Oracle, which use two-phase locking in order to provide serializable executions.

On the other hand, the two-phase locking requirement poses a limitation on the current design of our system, by preventing it to operate in the presence of semantically rich lock modes [31]. These lock modes permit very high concurrency to be achieved for update intensive workloads. They do so by exploiting the semantics of commutative operations, such as increment and decrement, and permit one transaction to modify the same data that was just modified by another transaction that has not yet committed. Using the modifications log (redo log) generated by the execution of such transactions may potentially cause uncommitted data to be read at the slave nodes in case the second transaction aborts after the first transaction committed.

Updating the slave nodes using modifications log, based on the logical database operations, instead of the physical ones, will eliminate the above limitation of our current system, however, at the potential prospect of increasing the overhead of log processing on the slaves. Evaluating this trade-off is an interesting (and daring) task we plan to look at as part of our future work.

Through experimental and theoretical analysis, we have shown that the benefits brought by our dynamic replication scheme far outweigh its drawbacks. Thus, in conclusion, we consider that *Dynamic Multiversioning* shows high promise for transparent scaling and automatic reconfiguration in any database application domain where database replication is suitable.

Bibliography

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

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

[3] InnoDB. `http://www.innodb.com/`.

[4] MySQL Database Server. `http://www.mysql.com/`.

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

[6] PHP hypertext preprocessor. `http://www.php.net/`.

[7] Postgres Database Management System. `http://www.postgres.com/`.

[8] 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.

[9] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *4th ACM/IFIP/Usenix International Middleware Conference*, June 2003.

[10] C. Amza, A.L. 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, 1996.

[11] Cristiana Amza, Alan Cox, and Willy Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, March 2003.

[12] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, 2000.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95, San Jose, CA USA*, pages 1–10, 1995.

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

[15] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[16] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multi-database transaction management. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 23–56. IBM Press, 1992.

[17] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *Proceedings of the USENIX 2004 Annual Technical Conference*, June 2004.

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

[19] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo 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.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[21] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *20th International Conference on Data Engineering, Boston, Massachusetts*.

[22] S. Frolund and R. Guerraoui. e-transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, pages 378–395, 2002.

[23] Gokul Soundararajan. Chameleon Project at the University of Toronto. TPC-W Client Emulator. `http://www.eecg.toronto.edu/~gokul/chameleon.html`.

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

[25] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

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

[27] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[28] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks, Goteborg, Sweden*, June 2001.

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

[30] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. In *ACM-TODS*, volume 25, September 2000.

[31] Henry F. Korth. Locking primitives in a database system. *ACM Journal*, 30(1):55–79, 1983.

[32] D. Lowell and P.M. Chen. Free transactions with Rio Vista. In *SOSP16*, pages 120–133, October 1997.

[33] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 126–137, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[34] Fernando Pedone and Svend Frølund. 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.

[35] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, New York, NY, USA, 1997. ACM Press.

[36] Christian Plattner and Gustavo 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.

[37] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 86–95. Morgan Kaufmann, 1997.

[38] Gokul Soundararajan and Cristiana Amza. Autonomic provisioning of backend databases in dynamic web servers. Technical Report CSRG TR-521, Department of Electrical and Computer Engineering, University of Toronto, Canada, January 2005.

[39] Alexander Thomasian. Concurrency control: methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[40] Transaction Processing Performance Council. TPC benchmark W standard specification, revision 1.8. `http://www.tpc.org/tpcw`.

[41] Alexey I. Vaysburd. *Building reliable interoperable distributed objects with the maestro tools*. PhD thesis, 1998. Adviser-Ken Birman.

[42] Yuanyuan Zhou, Peter Chen, and Kai Li. Fast cluster failover using virtual memory-mapped communication. In *Proceedings of the International Conference on Supercomputing*, June 1999.