RICE UNIVERSITY

Conflict-Aware Replication for Dynamic Content Web Sites

by

Cristiana Amza

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

______________________________
Willy Zwaenepoel
Hasselmann Professor, Chair
Computer Science and Electrical and
Computer Engineering

______________________________
Alan L. Cox
Associate Professor
Computer Science

______________________________
Vijay S. Pai
Assistant Professor
Electrical and Computer Engineering and
Computer Science

______________________________
Peter Druschel
Professor
Computer Science

Houston, Texas
May, 2003
Conflict-Aware Replication for Dynamic Content Web Sites

Cristiana Amza

Abstract

Conflict-aware replication is a novel lazy replication technique for scaling the back-end database of a dynamic content web server using a cluster of commodity computers. This technique provides both throughput scaling and 1-copy serializability. It has generally been believed that this combination is hard to achieve through replication because of the growth of the number of conflicts.

Conflict-aware replication interposes a (possibly replicated) scheduler between the database and application server tiers. The conflict-aware scheduler directs incoming queries in such a way that the overall execution is serializable and the number of conflicts is reduced. The technique requires that the incoming transactions specify the tables that they access at the beginning of the transaction. Using this information, conflict-aware replication provides both scaling and 1-copy serializability, while it avoids making any changes to the application server or database.

We have implemented a prototype of the conflict-aware scheduler in a cluster-based dynamic content site. We have also implemented various other scheduler algorithms in this prototype for comparison purposes, including conflict-aware and oblivious, with 1-copy serializability and with different looser consistency models.

We have evaluated this method using the industry standard TPC-W e-commerce benchmark, an auction site benchmark, modeled after eBay.com, and a bulletin board benchmark, modeled after slashdot.org. For these applications, we have found that pre-specifying what tables are accessed involves very little work on behalf of the pro-
grammer and could easily be automated. For clusters with small number of database machines (up to 8) we have measured an implementation of the algorithms. We use simulation to extend our measurement results to larger clusters, faster database engines, and lower conflict rates.

This dissertation shows that conflict-awareness brings considerable benefits in terms of both overall throughput scaling and latency reduction compared to both eager and conflict-oblivious lazy replication for a large range of cluster configurations and conflict rates. Furthermore, for all our applications, except those with very high conflict rates, the performance of conflict-aware replication equals or approaches that of looser consistency models. The dissertation also shows that the cost of conflict-aware replication is minimal in terms of data availability and fault tolerance.
Acknowledgments

I want to thank the members of my committee, Dr. Willy Zwaenepoel, Dr. Alan Cox, Dr. Vijay Pai and Dr. Peter Druschel, for their guidance and support in writing this thesis and all along the way.

I especially want to thank my advisors Dr. Willy Zwaenepoel and Dr. Alan Cox for encouraging me and staying beside me through thick and thin. I thank you for allowing me the freedom to always choose the topics that caught my imagination. I have been really fortunate to take part in very fulfilling projects and in writing some exciting papers together with you. I thank you for correcting me when I was wrong, and essentially always being there like true friends.

I want to thank the members of the DynaServer team for putting in the hours and the effort necessary for helping me develop and test the applications that I used to test my ideas. I am especially thankful to Karthick, Emmanuel and Anupam for their hard work and for their friendly support.

Last, but not the least, I would like to thank my parents and my family, especially my husband for their constant support and for believing in me. And, of course, to my kids: you make it all worthwhile.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations ix
List of Tables xiii

1 Introduction 1

2 Background 7
  2.1 General Characteristics of Dynamic Content Applications 7
  2.2 Replication ................................................................ 8
     2.2.1 1-Copy Serializability .......................................... 9
     2.2.2 Classic Replication ............................................ 9
  2.3 Summary ............................................................... 11

3 Conflict-Aware Replication 12
  3.1 Consistency Model .................................................. 12
  3.2 Programming Model .............................................. 13
  3.3 Cluster Design ..................................................... 13
  3.4 Key Idea .............................................................. 13
     3.4.1 Serializability with Asynchronous Updates .......... 14
     3.4.2 Conflict-Aware Optimizations ......................... 14
     3.4.3 Choice of Concurrency Control ..................... 15
  3.5 Summary .......................................................... 16
4 Implementations

4.1 Cluster Dynamic Content Web Site Implementation ............... 17
4.2 Conflict-Aware Scheduling Using Conservative 2PL .............. 18
  4.2.1 Lazy Read-one, Write-all Replication ..................... 19
  4.2.2 1-Copy Serializability .................................... 19
  4.2.3 Conflict-Aware Scheduling ................................. 20
  4.2.4 Prototype Implementation ................................. 20
4.3 Distributed Versioning: Conflict-Aware Scheduling Using Versioning . 26
  4.3.1 Programming Model ....................................... 26
  4.3.2 Lazy Read-one, Write-all Replication ..................... 27
  4.3.3 Assigning and Using Version Numbers .................... 27
  4.3.4 1-Copy Serializability and Absence of Deadlock .......... 28
  4.3.5 Limiting the Number of Conflicts ....................... 28
  4.3.6 Reducing Conflict Duration ............................ 28
  4.3.7 Prototype Implementation ............................... 29
4.4 Comparison of Conflict-Aware Scheduling Using Conservative 2PL and Versioning ........................................ 36
4.5 Fault Tolerance and Data Availability .......................... 39
  4.5.1 Fault Model ............................................. 40
  4.5.2 Fault Tolerance of the Sequencer .......................... 40
  4.5.3 Atomicity and Durability of Writes ....................... 40
4.6 Implementation Details ........................................ 43
  4.6.1 Load Balancing Algorithm ............................... 43
  4.6.2 Connection Handling ..................................... 44
4.7 Summary .................................................. 44

5 Experimental Methods ........................................ 46

5.1 Other Methods Used for Comparison ............................. 46
5.1.1 Serializable Methods .......................... 46
5.1.2 Non-serializable Methods with Specialized Consistency .... 48
5.2 Benchmarks ........................................ 51
5.2.1 TPC-W Benchmark ................................ 51
5.2.2 Auction Site Benchmark .......................... 56
5.2.3 Bulletin Board Benchmark ......................... 58
5.2.4 Client Emulation ................................. 60
5.3 Experimental Environment .......................... 60
5.4 Simulation Method .................................. 61
5.5 Summary ........................................... 62

6 Experimental Results .................................. 63
6.1 Baseline Experiments ............................... 64
6.2 Experimental Comparison of Serializable Methods for Conservative 2PL 66
  6.2.1 Experimental Throughput Comparison .............. 67
  6.2.2 Experimental Response Time Comparison .......... 70
  6.2.3 Breakdown of Query Time ........................ 71
6.3 Simulation of Serializable Methods for Conservative 2PL .......... 73
  6.3.1 Large Database Clusters ........................ 73
  6.3.2 Faster Replicas ................................ 79
  6.3.3 Varying Conflict Rates .......................... 80
6.4 Comparison of Conflict Awareness Using Conservative 2PL versus Versioning .................. 82
6.5 Comparison of Specialized Consistency and Serializability ........ 84
6.6 Cost of Fault Tolerance and Data Availability ................ 88
6.7 Summary ......................................... 89

7 Related Work ........................................ 91
7.1 General Purpose Cluster Servers .................... 91
7.2 Cluster Servers for Static Content Web Sites .......................... 91
7.3 Cluster Servers for Dynamic Content Web Sites ................... 93
7.4 Partitioning in Cluster Database Servers .............................. 95
7.5 Replication in Database Clusters ................................. 95
7.6 Caching Dynamic Content ...................................... 98
7.7 Multiversion Concurrency Control .............................. 100

8 Conclusions and Future Work ........................................ 101

Bibliography .......................................................... 104
Illustrations

1.1 Common architecture for dynamic content sites .................. 1
1.2 A cluster-based dynamic content web site .................... 3

4.1 Cluster components ............................................. 18
4.2 Scheduler actions for each type of query and query response .... 24
4.3 Conflict-aware replication with conservative 2PL: Protocol steps for each type of query and query response .................. 25
4.4 Conflict-aware replication with versioning: Protocol steps for each type of query and query response .................. 30
4.5 Sequencer assigned version numbers for a series of operations .... 31
4.6 Sequencer assigned version numbers for a series of operations and the version number produced at the database proxy after each operation commits .................................................. 36
4.7 Scheduler actions for each type of query and query response .... 37
4.8 Sequence of updates in a transaction ............................ 38
4.9 Serial execution in conservative 2PL (left), versus increased parallelism in distributed versioning with late version acquires and early version releases (right) ............................. 39

5.1 Example TPC-W transaction: Placing an order ................. 55
5.2 Example auction transaction: Placing a bid .................... 58
6.1 CPU utilization for the web server and database server in the 1-1 configuration for the TPC-W browsing, shopping and ordering mixes, the bulletin board, and auction site benchmarks ............... 65
6.2 Fraction of writes, reads and idle time at the database server in the 1-1 configuration for the TPC-W browsing, shopping and ordering mixes, the bulletin board, and auction site benchmarks ............... 66
6.3 Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the TPC-W browsing mix ............... 67
6.4 Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the TPC-W shopping mix ............... 68
6.5 Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the TPC-W ordering mix ............... 68
6.6 Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the auction site benchmark ............... 69
6.7 Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the bulletin board benchmark ............... 69
6.8 Web interaction response time for all schedulers and all workloads, normalized to response time of the Eager scheduler ............... 71
6.9 Query time breakdown for all schedulers for the TPC-W browsing mix, normalized by the query time of the Eager scheduler ............... 72
6.10 Query time breakdown for all schedulers for the TPC-W shopping mix, normalized by the query time of the Eager scheduler ............... 72
6.11 Query time breakdown for all schedulers for the TPC-W ordering mix, normalized by the query time of the Eager scheduler ............... 72
6.12 Simulated throughput results for the TPC-W browsing mix ............... 74
6.13 Simulated throughput results for the TPC-W shopping mix ............... 74
6.14 Simulated throughput results for the TPC-W ordering mix ............... 75
6.15 Simulated throughput results for the auction benchmark ............... 75
6.16  Simulated throughput results for the bulletin board benchmark  

6.17  Breakdown of the average database CPU time into idle time, time for 
reads, and time for writes, for the TPC-W browsing, shopping and 
ordering mixes  

6.18  Breakdown of the average database CPU time into idle time, time for 
reads, and time for writes, for the bulletin board and auction 
benchmarks.  

6.19  Simulated throughput results with faster database replicas for the 
TPC-W shopping mix  

6.20  Simulated throughput results for various conflict rates for the 
conflict-aware and the Eager scheduler for the TPC-W shopping mix  

6.21  Simulated throughput results for various conflict rates for the 
conflict-aware and the FR-L scheduler for the TPC-W shopping mix.  

6.22  Comparison of distributed versioning and conservative 2PL for the 
TPC-W shopping mix  

6.23  Comparison of distributed versioning and conservative 2PL for the 
TPC-W ordering mix  

6.24  Comparison of distributed versioning and conservative 2PL for the 
auction site  

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

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

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

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

6.29 Comparison of all consistency levels for the auction site. Level 2 is
the specialized consistency level ............................... 87
6.30 Overhead of providing fault tolerance and various degrees of
availability for the conflict-aware scheduler for all workloads. All
results represent throughput normalized to the throughput of the
Unreliable scheduler for that workload. .......................... 89
Tables

5.1 Database table characteristics ........................................... 52
5.2 Web interaction mix and characteristics ................................. 54

6.1 Average resource usage at the scheduler for the TPC-W shopping mix, the bulletin board and the auction site at the largest respective configurations. One scheduler is used for the TPC-W and auction benchmarks, two schedulers are used for the bulletin board benchmark. 79
Chapter 1

Introduction

This dissertation studies replication in database clusters [7, 54, 55, 87, 94] serving as back-ends in dynamic content sites.

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 application programs provide access to that content. A client request for dynamic content causes the web server to invoke a method in the application server. The application server executes the application program, issues SQL queries, one at a time, to the database and formats the results as an HTML page. The web server then returns this page in an HTTP response to the client.

In many current applications (e.g., e-commerce, auction sites), the application programs are quite simple in comparison to most of the database queries that they generate, resulting in a potential database bottleneck [3], even when dynamic content caching [29, 72, 96] is used [86].

This dissertation focuses on scaling the database tier by using data replication on

Figure 1.1 : Common architecture for dynamic content sites
clusters, while maintaining 1-copy serializability. We avoid making any changes to the database, web server or application server to support replication by interposing a (possibly replicated) scheduler between the application server and the database (see Figure 1.2).

Furthermore, throughout this dissertation, we restrict our attention to read-one, write-all replication schemes based on their higher flexibility and data availability [55, 93]. Thus, the scheduler directs writes to all database replicas, but selects a single replica for a read. It receives the responses for both reads and writes from the databases, and forwards a single response to the application server.

Database replication techniques can be categorized as eager [41] or lazy [45, 74, 89]. In eager replication, updates are synchronously applied to all replicas before the updating transaction commits. Eager replication maintains (1-copy) serializability, but performs poorly, mainly because of conflicts [41]. These conflicts cause large synchronization delays and frequent deadlocks. For this reason, most work has focused on lazy replication. Lazy replication algorithms asynchronously propagate replica updates, even after the updating transaction commits. They do not provide 1-copy serializability, since, for instance, writes may be propagated in different orders to different replicas. Past approaches have used reconciliation [78, 89] to achieve eventual replica consistency. Predictions of the number of reconciliations with a large number of replicas have, however, been discouraging [41]. More recent work [55] suggests that for small clusters, both 1-copy serializability and acceptable performance can be achieved by aborting conflicting transactions.

The goal of this dissertation is also to use lazy (asynchronous) replication and achieve serializability, but instead of resolving conflicts by aborting conflicting transactions, the scheduler that distributes load across the cluster is augmented to guarantee consistent results (see Figure 1.2).

In particular, the scheduler directs operations to those replicas where it knows they will find a copy of the data consistent with a serializable execution. For example, a
write on a data item may have been sent to all replicas, but it may have completed only at a subset of them. The scheduler keeps track of the completion status of each of the writes. Using this knowledge, the scheduler makes sure that a read that needs to happen after a particular write (in order to achieve serializability) is sent to a replica where it knows the write has already completed. We will refer to such a scheduling algorithm as conflict-aware scheduling.

A concern with the method outlined above is the extra state and the extra computation in the scheduler, raising the possibility of it becoming the bottleneck or making it an issue in terms of availability and fault tolerance of the overall cluster. We present a lightweight implementation of the conflict-aware scheduling algorithm, that allows a single scheduler to support a large number of databases. We also demonstrate how to replicate the scheduler for added scalability and availability.

We have implemented various versions of scheduling algorithm in order to better understand the benefits and overheads of the different parts of the conflict-aware scheduler. In our conflict-aware algorithm, the scheduler makes a separate scheduling decision for each query, and needs to maintain state for each outstanding query in a transaction. We compare this scheme against a transaction-level conflict-aware scheme and both transaction-level and query-level oblivious schemes. In the latter, the scheduler optimizes load balancing, and pays no attention to conflicts. We also
compare all these algorithms to a baseline eager replication algorithm.

Next, we vary the concurrency control used at each database replica. In particular, we have implemented two versions of the main query-level conflict-aware algorithm: one in conjunction with traditional conservative two-phase locking (2PL) [5, 16] and one in conjunction with a novel concurrency control based on explicit versioning [6].

Finally, we compare conflict-aware replication with recently proposed alternatives to lazy and eager consistency which provide specialized consistency models, tailored for each application’s needs. The Neptune [81] and Continuous Consistency [98] projects argue that either several distinct consistency models or a tunable consistency model should be supplied for dynamic content applications, since each has its own consistency requirements. For instance, the application can be provided in-order writes, but its reads can access only copies that satisfy a specific staleness bound. The programmer would then have to define the appropriate consistency abstraction (e.g., the consistency unit and appropriate parameters for the consistency model) for each individual application.

We show that these models are not very useful in state-of-the-art clusters, because the write operations usually arrive either in the “correct order”, or otherwise with minimal delays. Thus, the overhead of consistency maintenance comes mostly from conflicts between replicated transactions, the very problem addressed by our conflict-aware scheduling algorithm.

Our implementation uses common software platforms such as the Apache Web server [1], the PHP scripting language [69], and the MySQL relational database [62]. We use three dynamic content applications in the evaluation which model three common types of web sites with dynamic content: an online bookstore, an auction site and a bulletin board. For the online bookstore, we use the TPC-W specification [91] for e-commerce sites. The auction site and the bulletin board are modeled after eBay.com [37] and slashdot.org [85], respectively. The TPC-W benchmark specifies three different workload mixes, differing in the ratio of read-only to read-write inter-
actions. The browsing mix contains 95% read-only interactions, the shopping mix 80%, and the ordering mix 50%. For the other two benchmarks we use a typical workload of a real site of the corresponding type [3].

The evaluation uses measurement of the implementation for small clusters (up to 8 database machines). Larger clusters (up to 60 database machines) are evaluated using a simulator that was validated against the implementation.

Our results show that:

1. All three applications scale well. The TPC-W browsing workload, the TPC-W most common (shopping) workload and the bulletin board scale well up to the maximum simulated configuration. The TPC-W ordering workload scales only to 16 databases. The auction site scales up to 40 databases.

2. The benefits of conflict awareness in the scheduler are substantial. Compared to a lazy protocol without this optimization, the improvements are up to a factor of 2 on our largest experimental platform, and up to a factor of 3 in the simulations. Compared to an eager protocol, the improvements are even higher (up to a factor of 3.5 in the experiments, and up to a factor of 4.4 in the simulations).

3. Simulations of more powerful database machines and varying conflict rates validate the performance advantages of conflict-aware scheduling on a range of software/hardware environments.

4. Two schedulers are enough to support scaling for up to 60 MySQL engines for all workloads, assuming equivalent nodes are used for the scheduler and MySQL engine.

5. Explicit versioning improves concurrency, hence performance compared to conservative 2PL, especially for the workloads with high conflict rates.
6. For all our applications, except those with very high conflict rates (i.e., the TPC-W ordering mix), the performance of conflict-aware replication equals or approaches that of looser consistency models. This allows for a simple programming model that users are already familiar with, and familiar abstractions such as transactional isolation and atomicity.

7. The cost of maintaining the extra state in the scheduler is minimal in terms of scaling, availability and fault tolerance.

The outline of this dissertation is as follows. Chapter 2 provides the necessary background on the general characteristics of dynamic content workloads and database replication techniques. Chapter 3 introduces the conflict-aware solution. Chapter 4 describes our prototype implementation and the conflict-aware scheduling techniques explored in the dissertation. Chapter 5 presents the other scheduling techniques used for comparison to understand the benefits and overheads of various parts of the conflict-aware algorithm. In particular, the chapter discusses scheduling algorithms with and without conflict-awareness and serializability enforcement. This chapter also presents the benchmarks, the experimental platform, and the simulation model. We investigate, experimentally and by simulation, how conflict-aware scheduling techniques affect scaling in Chapter 6. Chapter 7 discusses related work. Chapter 8 concludes the dissertation and outlines avenues for future work.
Chapter 2

Background

In this chapter we provide an intuitive motivation for the techniques used in this dissertation. This motivation is based on observing the characteristics of the workloads imposed on the database of a dynamic content site by a number of benchmarks [3].

We describe these characteristics and argue that a replication scheme is best suited for scaling these applications, then introduce the necessary background on replication.

2.1 General Characteristics of Dynamic Content Applications

For traditional database applications such as on-line transaction processing (OLTP) workloads [91], data partitioning across the cluster [17, 34] is necessary to alleviate the massive I/O needs of these applications through in-memory data caching. In contrast, for the usual application sizes, there is little disk I/O in dynamic content applications [3], due to the locality exhibited by these applications. For example, in on-line shopping [91], bestsellers, promotional items and new products are accessed with high frequency. Similarly, the stories of the day and items on auction are hot objects in bulletin board and on-line bidding applications, respectively [3]. This makes replication much more promising for scaling dynamic content applications.

As a bonus, replication brings with it high data availability, and is considerably easier to use than data partitioning which implies rather complex optimizers to minimize reconfigurations and data movement between machines [31]. On the down-side, replication introduces the cost of replicating the execution of update queries for keeping the table replicas consistent. Fortunately, in dynamic content applications, the
queries that update the database are usually lightweight compared to read-only requests (see Section 5.2.1). For instance, in e-commerce, typically only the record pertaining to a particular customer or product is updated, while any given customer may browse the product database using complex search criteria. More importantly, the locality in access patterns of dynamic content applications may imply higher conflict rates relative to traditional applications, given a sufficiently high fraction of writes. For instance, the probability that a “best seller” book is being bought concurrently by two different customers, incurring a conflict on that item’s stock, is much higher than the probability that both husband and wife access their joint account at the same time. Thus, intuitively, e-commerce applications have potentially higher conflict rates than traditional OLTP applications. Furthermore, each conflict’s duration is also arguably longer than in traditional OLTP database applications. For instance, placing an e-commerce order may involve a long atomic sequence of operations (e.g., the item stock is updated in the product table for all items purchased, a new customer record with personal data such as name and address is recorded, a new order record is created, the credit card information is inserted, etc). In contrast, an OLTP transaction is generally a fine-grained update to an individual record (e.g., decrementing a customer’s balance).

2.2 Replication

In this dissertation, we focus on read-one, write-all replication schemes [93] on a cluster of databases. In these replication schemes, each read query in the workload is executed by only one node in the cluster, while the update queries are applied by all (available) nodes in the cluster. In the following we define 1-copy serializability in a replicated system, and we describe classic replication approaches.
2.2.1 1-Copy Serializability

A replicated system looks like one copy to the user (1-copy serializability [16]), if the system behaves equivalent to a serial execution. Intuitively, a replicated system looks like one physical copy to the user if all conflicting transactions are ordered consistently on all replicas.

The more formal definition of serializability, according to Gray and Reuter [42], involves avoidance of the following phenomena:

- **P0** ("Lost Update"): $T_2 r(x), T_1 w(x), T_2 w(x)$ based on value read, $T_1$’s update is lost

- **P1** ("Dirty Read"): $T_1 w(x), T_2 r(x)$, if $T_1$ aborts $T_2$ is inconsistent

- **P2** ("Un-repeatable read"): $T_1 r(x), T_2 w(x), T_1 r(x)$ again and sees a different value.

- **P3** ("Read Skew"): $T_1 r(x), T_2 w(x), T_2 w(y), T_1 r(y)$. If there exists a constraint between $x$ and $y$, $T_1$ might read versions of $x$ and $y$ that do not fulfill this constraint.

- **P4** ("Write Skew"): $T_1 r(x) T_2 r(y) T_1 w(y) T_2 w(x)$. If there exists a constraint between $x$ and $y$, it might be violated by the two writes.

2.2.2 Classic Replication

According to when the updates are propagated and applied, replication schemes have traditionally been classified into eager replication, in which updates are applied synchronously before the local transaction commits, and lazy replication, in which updates are applied asynchronously after the local transaction commits.
Eager Replication

Eager replication is a classic replication scheme which enforces 1-copy serializability between replicas.

The algorithm [93] is fully synchronous. The execution of every lock and write operation is replicated and completes on all database back-ends, before proceeding to the next operation. Furthermore, the commit decision is returned to the user only after all databases have committed the update transaction. Hence, databases are fully consistent at all times.

At any given point, replicas may have different conflict waiting times and different loads. Thus, acquiring locks synchronously on all replicas implies that lock waiting times increase with the number of replicas. Furthermore, the fraction of writes and the conflict rate for a node increase with the cluster size (due to additional incoming remote updates). An increase in conflicts brings about longer delays and an increase in the deadlock probability for any particular transaction [41]. Thus, eager replication schemes that enforce 1-copy serializability between replicas are regarded as non-scalable.

Lazy Replication

Lazy replication [45, 89] with delayed propagation of modifications was introduced to address the scaling problems of eager replication. In traditional lazy schemes, writes are applied asynchronously at all other replicas after the local update transaction commits. Thus, writes can arrive out-of-order at different sites and reads can access inconsistent data. These inconsistencies are usually addressed in ad-hoc ways. For example reads can access only out-of-date copies that satisfy a specific freshness bound, and reconciliation [78, 89] is used for out-of-order writes.

However, the increasing fraction of writes that each node has to handle, results in an increasing rate of reconciliations with the cluster size. Hence, this scheme has also been regarded as non-practical for scaling [41], especially for applications with
strong consistency requirements (e.g., e-commerce applications).

2.3 Summary

The main characteristics of dynamic content applications differ in many ways from those of traditional database applications inviting a departure from traditional approaches to scaling these applications. There is a high degree of locality in the database access patterns. The read queries in the typical workload are more frequent and more complex than the write queries.

These characteristics favor read-one, write-all replication: the expensive components of the workloads (reads) are executed on a single machine, and only the inexpensive components (writes) need to be executed on all machines. Classic replication schemes can provide either scaling (through lazy replication), or consistency (through eager replication), but not both, mainly due to conflicts, which cause out-of-order writes in lazy replication and waiting times in eager replication. This is the problem that we address using the methods described in the next chapter.
Chapter 3

Conflict-Aware Replication

This dissertation centers around the key observation that to achieve good scalability and availability in a dynamic content site, it is necessary and sufficient to decouple the server’s internal view of the data from the user’s view of the data. In particular, we take advantage of the presence in a database cluster of a scheduler through which all incoming requests pass. This scheduler augmented with reliable state allows the server to use a series of performance optimizations based on loose replica consistency internally, while externally it conforms to serializability as perceived by the user.

For all transactions, the tables accessed are specified beforehand. The scheduler uses this information both for avoiding deadlocks and to guide two replication optimizations: asynchronous updates and conflict avoidance, which, as this dissertation will show, are essential for scaling. At the same time, the scheduler maintains consistency as perceived by the user through a conflict ordering scheme.

In the following, we introduce the consistency and programming model, our cluster design and an overview of our solution.

3.1 Consistency Model

The consistency model used for all protocols is strong consistency or 1-copy serializability [16] which makes the system look like one copy to the user.

With 1-copy serializability, conflicting operations of different transactions execute in the same order on all replicas (i.e., the execution of all transactions is equivalent to a serial execution on a single machine).
3.2 Programming Model

A single (client) web interaction may include one or more transactions, and a single transaction may include one or more read or write queries. The application writer specifies where in the application code transactions begin and end. In the absence of transaction delimiters, each single query is considered a transaction and is automatically committed (so called “auto-commit” mode). At the beginning of each transaction consisting of more than one query, the application writer inserts a *pre-declaration* of all tables accessed in the transaction and their access types (read or write). This step is currently done by hand, but could be automated. Tables for single-operation transactions do not need to be specified.

3.3 Cluster Design

We consider a cluster architecture for a dynamic content site, where a scheduler distributes incoming requests to a cluster of database replicas and delivers the responses to the application server (see Figure 1.2). The scheduler may itself be replicated for performance or for availability.

The application server interacts directly only with the schedulers. If there is more than one scheduler in a particular configuration, the application server is assigned a particular scheduler at the beginning of a transaction by round-robin. For each query of this transaction, the application server only interacts with this single scheduler, as if it were a regular database. These interactions are synchronous: for each query, the application server blocks until it receives a response from the scheduler. This type of database interaction represents the common operation of dynamic content web sites.

3.4 Key Idea

The key idea in this dissertation is to augment the scheduler that distributes requests to the cluster of database back-ends to guarantee consistent results and reduce conflict
waiting times.

### 3.4.1 Serializability with Asynchronous Updates

The scheduler establishes a total order for conflicting transactions on all replicas, by assigning appropriate unique sequence numbers to read and write queries. It then tags each query with its sequence number and sends write queries to all replicas and read queries only to a single replica.

Executing the operations in the order of their assigned sequence numbers at each replica ensures serializability, while it allows for asynchronous execution of writes at each replica.

### 3.4.2 Conflict-Aware Optimizations

Due to the asynchronous updates, at any given point some replicas may be up-to-date with the application of writes, while other replicas may contain stale information. Sending a subsequent read query to any database would still result in a serializable execution if the sequence-number order is respected. However, the scheduler optimizes waiting time for each read by sending it only to an up-to-date replica. We call this optimization conflict-aware scheduling.

Furthermore, since each query in a transaction is serviced synchronously, from the time the first read in a transaction executes to the time a subsequent read arrives for scheduling, the load on the database replicas could have changed. The scheduler dynamically adjusts to changes in load, by making fine-grained load balancing decisions at the level of every query. In particular, the scheduler sends a read to the least loaded replica from the set of up-to-date replicas.

To implement the above optimizations the scheduler needs to maintain internal state for all outstanding replicated operations in each transaction. We show with a prototype implementation, that a lightweight scheduler with minimal resource consumption in terms of CPU, memory and disk usage is possible. We further show that
adding persistence and availability to the scheduler state incurs very little overhead.

3.4.3 Choice of Concurrency Control

The choice of concurrency control at each database replica is orthogonal to the main scheduling algorithm.

Conflict-aware query scheduling can be used in conjunction with any traditional concurrency control method such as timestamp-based or two-phase locking (2PL) [16]. As long as there is a means of conflict detection, our query-level scheduler can store and use that information to avoid conflicts.

On the other hand, the deadlock probability for a replicated database cluster becomes prohibitive at large clusters due to the extra updates that each node performs on behalf of the other nodes [41]. Thus, we deem it necessary to use a deadlock-free scheme for scaling to large clusters. Hence, we do not use implicit 2PL which is generally adopted, because it can lead to deadlocks.

Timestamp-based protocols avoid deadlocks and solve conflicts through transaction aborts. Thus, a transaction may be restarted over and over wasting resources, especially if the application has long transactions. This problem escalates in larger clusters because the conflict frequency and thus the abort rate increases with cluster size [41, 55].

Deadlock can be prevented by using conservative 2PL in which all locks in a transaction are acquired at the beginning of the transaction. The problem with this method is a limitation in parallelism because the initial lock call blocks until all read and write locks are available.

In the following chapters, we show how conflict-aware replication can be applied in the context of the traditional conservative 2PL algorithm (Chapter 4.2), and how parallelism can be improved through a new version-based concurrency control algorithm that we introduce in Chapter 4.3.
3.5 Summary

In this chapter, we introduce conflict-aware replication. The scheduler that distributes requests to the cluster of database back-ends uses internal optimizations for scheduling queries across replicas, while externally it conforms to 1-copy serializability as perceived by the client. The choice of concurrency control at each database replica is orthogonal to the scheduling scheme. We implement two schedulers which combine conflict-awareness with two different deadlock-free concurrency control methods: conservative 2PL and a new explicit versioning scheme.
Chapter 4

Implementations

This chapter describes the prototype implementation of our conflict-aware scheduler and is organized as follows. First, in Section 4.1, we discuss the design of the dynamic content site that forms the basis for the implementation of all algorithms discussed in this dissertation. Next, in Sections 4.2 and 4.3, we discuss two versions of the conflict-aware scheduling algorithm: a conflict-aware scheduler in conjunction with conservative 2PL concurrency control, and a conflict-aware scheduler in conjunction with concurrency control based on explicit versioning, respectively. We then present a comparison between the two versions of conflict-aware scheduler in Section 4.4. Finally, we present the implementation of the fault tolerance and availability aspects of our solution in Section 4.5, and we discuss implementation aspects orthogonal to conflict-awareness such as the choice of load balancing algorithm in Section 4.6.

4.1 Cluster Dynamic Content Web Site Implementation

The web server cluster design consists of three types of processes (see Figure 4.1): scheduler processes (one per scheduler machine), a sequencer process (one for the entire cluster), and database proxy processes (one for each database replica). The scheduler requests the sequencer to assign unique sequence numbers, and uses the assigned sequence numbers to enforce the total order. A database proxy regulates access to its database server by only letting a query proceed if the database has already processed all conflicting operations that precede it in the total order, and all operations that precede it in the same transaction. The schedulers form the core of the implementation. They receive the various operations from the application servers
(table declaration, read and write queries, and commit or abort transaction), forward them as appropriate to the sequencer or one or more of the database proxies, and relay the responses back to the application servers.

To the web/application servers, a scheduler looks like a database engine. At the other end, each database engine interacts with the database proxy as if it were a regular web/application server. As a result, we can use any off-the-shelf web server (e.g., Apache), any off-the-shelf application server (e.g., PHP), and any off-the-shelf database (e.g., MySQL) without modification. The system is easy to configure and reconfigures itself automatically in case of failures (see Section 4.5).Schedulers and database proxies read a configuration file at startup, and set up connections accordingly.

### 4.2 Conflict-Aware Scheduling Using Conservative 2PL

In this section we describe the implementation of conflict-aware scheduling when conservative 2PL [16] is used as the concurrency control algorithm at each database replica. The programming model remains the same as described in Section 3.2. The
predeclaration of all tables accessed in the transaction and their access types (read or write) for each transaction consisting of more than one query takes the form of a lock acquire for the corresponding tables and access types.

4.2.1 Lazy Read-one, Write-all Replication

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

4.2.2 1-Copy Serializability

The scheduler maintains 1-copy serializability by assigning a unique sequence number to each transaction. This assignment is done at the beginning of the transaction. For a multiple-operation transaction, the sequence number is assigned when that transaction’s lock request arrives at the scheduler. For a single-operation transaction, sequence number assignment is done when the transaction arrives at the scheduler (except for single-read transactions, see below). Lock requests are sent to all replicas, executed in order of their assigned sequence numbers, and held until commit, thus forcing all conflicting operations to execute in a total order identical at all replicas, and thus enforcing 1-copy serializability.

Transactions consisting of a single read query are treated differently. A single-read transaction holds locks only on the single replica where it executes. This optimization results in a very substantial performance improvement without violating 1-copy serializability.
4.2.3 Conflict-Aware Scheduling

Due to the asynchrony of replication, at any given point, some replicas may have fallen behind with the application of writes. Furthermore, some replicas may have not been able to acquire the locks for a particular transaction, due to conflicts. For reads, other than reads in single-read transactions, the scheduler first determines the set of replicas where the locks for its enclosing transaction have been acquired and where all previous writes in the transaction have completed. It then selects the least loaded replica from this set as the replica to receive the read query. The scheduler tries to find a conflict-free replica for single-read transactions as well, but may not be able to find one.

Conflict-aware scheduling requires that the scheduler maintains the completion status of lock requests and writes, for all database replicas.

4.2.4 Prototype Implementation

In the following, we describe the state maintained at the scheduler and at the database proxy to support failure-free execution and the protocol steps executed on receipt of each type of operation and its response. Additional state maintained for fault-tolerance purposes is described in Section 4.5.

The Scheduler’s State

The scheduler maintains for each active transaction its sequence number and the locks requested by that transaction. In addition, it maintains a record for each operation that is outstanding with one or more database proxies. A record is created when an operation is received from the application server, and updated when it is sent to the database engines, or when a reply is received from one of them. The record for a read operation is deleted as soon as the response is received and delivered to the application server. For every replicated operation (i.e., lock request, write, commit or abort), the corresponding record is deleted only when all databases have returned
a response.

The scheduler records the current load of each database (see Section 4.6.1). This value is updated with new information included in each reply from a database proxy.

**The Database Proxy’s State**

The database proxy maintains a reader-writer lock [93] queue for each table. These lock queues are maintained in order of sequence numbers. Furthermore, the database proxy maintains transaction queues, one per transaction, and an out-of-order queue for all operations that arrive out of sequence number order.

For each transaction queue, a head-of-queue record maintains the current number of locks granted to that transaction. Each transaction queue record maintains the operation to be executed. Transaction queue records are maintained in order of arrival.

**Lock Request**

For each lock request, the scheduler obtains a sequence number from the sequencer and stores this information together with the locks requested for the length of the transaction (see Figure 4.3-a). The scheduler then tags the lock request with its sequence number and sends it to all database proxies. Each database proxy executes the lock request locally and returns an answer to the scheduler when the lock request is granted. The lock request is not forwarded to the database engine.

A lock request that arrives at the database proxy in sequence number order is split into separate requests for each of the locks requested. When all locks for a particular transaction have been granted, the proxy responds to the scheduler. The scheduler updates its record for that transaction, and responds to the application server, if this is the first response to the lock request for that transaction. A lock request that arrives out-of-order (i.e., does not have a sequence number that is one more than the last processed lock request) is put in the out-of-order queue. Upon further lock
request arrivals, the proxy checks whether any request from the out-of-order queue can now be processed. If so, that request is removed from the out-of-order queue and processed as described above.

**Reads and Writes**

As the application executes the transaction, it sends read and write operations to the scheduler. The scheduler tags each operation with the sequence number that was assigned to the transaction. It then sends write operations to all database proxies, while reads are sent to only one database proxy (see Figures 4.3-b and c).

The scheduler sends each read query to one of the replicas where the lock request and the previous writes of the enclosing transaction have completed. If more than one such replica exists, the scheduler picks the replica with the lowest load.

The database proxy forwards a read or write operation to its database only when all previous operations in the same transaction (including lock requests) have been executed. If an operation is not ready to execute, it is queued in the corresponding transaction queue.

**Completion of Reads and Writes**

On the completion of a read or a write at the database, the database proxy receives the response and forwards it to the scheduler (see Figures 4.3-b and c). The proxy then submits the next operation waiting in the transaction queue, if any.

The scheduler returns the response to the application server if this is the first response it received for a write query or if it is the response to a read query. Upon receiving a response for a write from a database proxy, the scheduler updates its corresponding record to reflect the reply.
Commit/Abort

The scheduler tags the commit/abort received from the application server with the sequence number and locks requested at the start of the corresponding transaction, and forwards the commit/abort to all replicas (see Figure 4.3-d).

If other operations from this transaction are pending in the transaction queue, the commit/abort is inserted at the tail of the queue. Otherwise, it is submitted to the database. Upon completion of the operation at the database, the database proxy releases each lock held by the transaction, and checks whether any lock requests in the queues can be granted as a result. Finally, it forwards the response to the scheduler.

Upon receiving a response from a database proxy, the scheduler updates the corresponding record to reflect the reply. If this is the first reply, the scheduler forwards the response to the application server.

Single-Read Transactions

The read is forwarded to a database proxy, where it executes after previous conflicting transactions have finished. In particular, requests for individual locks are queued in the corresponding lock queues, as with any other transaction, and the transaction is executed when all of its locks are available.

To choose a replica for the read, the scheduler first selects the set of replicas where the earlier update transactions in the same client web interaction, if any, have completed. It then determines the subset of this set at which no conflicting locks are held. This set may be empty. It selects a replica with the lowest load from the latter set, if it is not empty, and otherwise from the former set.

Single-Update Transactions

Single-update transactions are logically equivalent to multiple-operation transactions, but in the implementation they need to be treated a little differently. The necessary locks are not specified by the application logic, hence there is no explicit lock request.
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockAcquire</td>
<td>Obtain a sequence number, tag and broadcast lock</td>
</tr>
<tr>
<td>LockAcqResp</td>
<td>Update backlog record, respond to client if first response</td>
</tr>
<tr>
<td>WriteQ</td>
<td>Tag write with sequence number &amp; broadcast</td>
</tr>
<tr>
<td>WriteResp</td>
<td>Respond to client if first replica response</td>
</tr>
<tr>
<td>ReadQ</td>
<td>Tag read with sequence number,</td>
</tr>
<tr>
<td></td>
<td>Pick replica with no conflicts, send read</td>
</tr>
<tr>
<td>ReadResp</td>
<td>Respond to client</td>
</tr>
<tr>
<td>Commit</td>
<td>Tag with sequence number, piggyback tables, broadcast</td>
</tr>
<tr>
<td>CommitResp</td>
<td>Update backlog record, respond to client if first replica response</td>
</tr>
</tbody>
</table>

Figure 4.2: Scheduler actions for each type of query and query response

When the scheduler receives a single-update transaction, it computes the necessary locks and obtains a sequence number for the transaction. The transaction is then forwarded to all replicas with that additional information. Thus, the lock request is implicit rather than sent in a separate lock request message to the database proxy, but otherwise the database proxy treats a single-update transaction in the same way as any multiple-update transaction.

Summary

Figure 4.2 shows the actions taken by the scheduler when each type of query is received for processing and when its response is received from the database.

The scheduler obtains a unique sequence number for the initial LockAcquire from the sequencer, makes a record for the LockAcquire, tags and broadcasts the LockAc-
Figure 4.3: Conflict-aware replication with conservative 2PL: Protocol steps for each type of query and query response

quire. When the response is received, the scheduler updates the LockAcquire record to reflect it and sends the response to the client if this is the first response.

For each write in the transaction, the scheduler tags it with the same sequence number obtained at the LockAcquire and broadcasts it. The scheduler creates a record for the write in its internal data structures and updates it whenever it receives a response. The scheduler sends each read query to one of the databases where the locks have already been acquired and the previous writes in the same transaction have finished. If more than one up-to-date replicas without conflicts exist, the scheduler uses a load balancing scheme to pick a replica. For both writes and commits, the response is sent to the application server (client) as soon as the first replica responds.
4.3 Distributed Versioning: Conflict-Aware Scheduling Using Versioning

In this section, we show how conflict-awareness can be used in conjunction with a concurrency control method based on explicit versioning [6]. The goal of this version is to improve parallelism at the level of each database, by delaying the point where a transaction needs to wait for a particular table version to become available. Instead of assigning a single sequence number for a transaction, each transaction is atomically assigned per-table versions for each table it accesses. Each transaction produces new versions for the tables it accesses after each table’s corresponding last use in the transaction. This allows us to establish a total order for conflicting transactions on all replicas, while at the same time enabling us to wait for each table version independently.

In the following we describe how distributed versioning achieves the desirable properties of conflict-awareness, 1-copy serializability and absence of deadlock, with a higher degree of concurrency and only minor changes to the programming model.

4.3.1 Programming Model

As before, at the beginning of each transaction consisting of more than one query, the programmer inserts a pre-declaration of the tables accessed in the transaction and their modes of access. As before, the tables accessed by single-operation transactions do not need to be pre-declared.

Additionally, the programmer optionally inserts a last use annotation after the last use of a particular table in a transaction. These last-use annotations are not necessary for the algorithm’s correctness, but improve its performance.
4.3.2 Lazy Read-one, Write-all Replication

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

4.3.3 Assigning and Using Version Numbers

A separate version number is maintained for each table in the database. A transaction is assigned a version number for each table that it accesses. As discussed in Section 4.3.1, each multi-query transaction declares what tables it will read or write before it starts execution; the tables accessed by single-query transactions are implicitly declared by the query itself. Based on this information, the scheduler assigns table versions to be accessed by the queries in that transaction (except for single-read transactions, see below). This assignment is done atomically, i.e., the scheduler assigns all version numbers for one transaction, before it assigns any version numbers for the next transaction. Version number assignment is done in such a way that, if there is a conflict between the current transaction and an earlier one, the version numbers given to the current transaction for the tables involved in the conflicts are higher than the version numbers received by the earlier conflicting transaction. All operations on a particular table are executed at all replicas in version number order.

Transactions consisting of a single read query are treated differently. A single-read transaction is assigned an ad-hoc position in the total order of read-write transactions. In particular, the single read transaction executes after all previous conflicting read-write transactions finish at the particular replica where the read is scheduled. This optimization results in a very substantial performance improvement without violating 1-copy serializability.
4.3.4 1-Copy Serializability and Absence of Deadlock

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

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

4.3.5 Limiting the Number of Conflicts

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

4.3.6 Reducing Conflict Duration

A read or a write query only waits if its version is not available by the time it is scheduled. The duration of such conflict waits is reduced by releasing versions early.
Specifically, after the last use of a particular table in a transaction, we insert an explicit release operation for that table in the application code. This optimization does not compromise 1-copy serializability. All versions are atomically pre-assigned at the beginning of each transaction, and a version release occurs only after the last use of a particular table. Hence, the total ordering of conflicting transactions for the replicated system is the same as the one in the system without early releases. Furthermore, the underlying consistency maintenance of each database, which remains unchanged, ensures serializability at each individual database. In conclusion, when early version releases are used, the overall system guarantees 1-copy serializability.

4.3.7 Prototype Implementation

Overview

Given the cluster design described in Chapter 3, the roles of each component (scheduler, sequencer and database proxy processes) change as described below.

The sequencer assigns unique version numbers to the tables accessed by each transaction. The database proxy regulates access to its database server by only letting a query proceed if the database has the right versions for the tables named in the query. The schedulers receive the various operations from the application server (table declaration at begin transaction, read and write queries, and commit or abort transaction) forward them as appropriate to the sequencer and/or one or all of the database proxies, and relay the responses back to the application server.

Transaction Start

The client informs the scheduler of all tables that are going to be accessed, and whether a particular table is read or written. The scheduler forwards this message to the sequencer (see Figure 4.4-a). The sequencer assigns version numbers to each of the tables for this transaction, and returns the result to the scheduler. The scheduler stores this information for the length of the transaction. It then responds to the client
so that it can continue with the transaction. The version numbers are not passed to the client. For each table, the sequencer remembers two values: the sequence number next-for-read, to be assigned if the next request is for a read, and the sequence number next-for-write, to be assigned if the next request is for a write. When the sequencer receives a request from the scheduler for a set of version numbers for tables accessed in a particular transaction, the sequencer returns for each table the next-for-read or the next-for-write sequence number, depending on whether that particular table is to be read or written in that transaction. After a sequence number is assigned for a write, next-for-write is incremented and next-for-read is set to the new value of next-for-write. After a sequence number is assigned for a read, only next-for-write is incremented.
operation       w w r w r r w
version assigned 0 1 2 3 4 4 7
next_for_read    0 1 2 2 4 4 4 7
next_for_write   0 1 2 3 4 5 6 7 7

Figure 4.5: Sequencer assigned version numbers for a series of operations

The intuition behind this version number assignment is that the version number assigned to a transaction for a particular table increases by one every time the new transaction contains an operation that conflicts with the previous transaction to access that table. For example, Figure 4.5 shows a series of read and write operations on a particular table, each belonging to a different transaction, in the order of arrival of the transaction’s version number request at the sequencer. The figure also shows the version numbers assigned by the sequencer for that table to each transaction and the values of next-for-read and next-for-write. As long as the successive accesses are reads, their transactions are assigned the same version number. Whenever there is a read-write, write-read, or write-write conflict, a higher version number is assigned. The assignment of version numbers for a particular transaction is atomic. In other words, all version numbers for a given transaction are assigned before any version number for a subsequent transaction is assigned. As a result, the version numbers for all tables accessed by a particular transaction are either less than or equal to the version numbers for the same tables for any subsequent transaction. They are only equal if the transactions do not conflict (they either do not access the same tables or the operations in both transactions are reads).
Reads and Writes

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

Enforcing 1-Copy Serializability

Both for read and write queries, the scheduler tags each table with the version number that was assigned to that table for this transaction. It then sends write queries to all replicas, while read queries are sent only to one replica (see Figures 4.4-b and c).

The following rules govern the execution of a query:

- A write query is executed only when the version numbers for each table at the database match the version numbers in the query.

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

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

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

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

Since a single-read transaction executes only at one replica, there is no need to assign cluster-wide version numbers to such a transaction. Thus, the scheduler forwards the transaction to the chosen replica, without assigning (new) version numbers. The chosen database proxy establishes an ad-hoc order for the read that ensures data freshness. The read query executes after the update transaction with the highest version numbers for the corresponding tables in the proxy’s queues releases these table versions.

Because the order of execution for a single-read transaction is ultimately decided by the database proxy, the scheduler does not block such queries. In case of conflict, the read query waits at the database proxy. The scheduler attempts to optimize this wait by selecting a replica that has achieved at least the highest version number that the scheduler has previously assigned for the corresponding tables, if such a replica exists.

Completion of Reads and Writes

On the completion of a read or a write at the database (see Figures 4.4-b and c), the database proxy receives the response and forwards it back to the scheduler.

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

Early Version Releases

The application server sends the last use annotation to the scheduler just like any other query. The scheduler uses this information to send an explicit version_release message that increments the specified table’s version at each database.
Commit/Abort

The scheduler tags the commit/abort with the tables accessed in the transaction, their version numbers and a corresponding \texttt{version\_release} flag, and forwards the commit/abort to all replicas (see Figure 4.4-d). The transaction’s commit carries a version release only for the tables where early version releases have not already been performed. Single-update transactions carry an implicit commit (and \texttt{version\_release}).

Upon completion at any database, the corresponding database proxy increments all specified versions for which a \texttt{version\_release} is indicated and returns the answer to the scheduler. The scheduler updates its state to reflect the reply. If this is the first reply, the scheduler immediately forwards the response to the client.

1-Copy Serializability

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

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

1. \texttt{next-for-write} counts all the earlier transactions that access the same table.
   This value is assigned as the version number for the table for this transaction.

2. The database proxy increments its version number every time a transaction that accesses that table completes.

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

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

2. The current transaction is not allowed to execute at a database proxy before the version number for that table at that database proxy reaches (at least) the transaction’s version number for this table.

3. The algorithm also allows a read query to execute at a database proxy if the database proxy’s version number for the table is higher than that of the transaction. The only way this can happen is as a result of a sequence of transactions with reads on the table, and these can execute in parallel without violating the total order on conflicting operations.

Continuing the example of Figure 4.5, Figure 4.6 shows the sequencer assigned versions and the versions each operation’s commit produces at the database proxy for the table accessed by that operation. All three reads assigned version number 4 can also read versions 5 and 6 (i.e., versions produced by other concurrent readers). On the other hand, a write is required to wait until all previous readers are done and the version at the database has been incremented to match its own (e.g., the write assigned version number 7).
operation \textit{w w r w r r w}

version assigned \textit{0 1 2 3 4 4 7}

version produced \textit{1 2 3 4 5 6 7 8}

Figure 4.6: Sequencer assigned version numbers for a series of operations and the version number produced at the database proxy after each operation commits

Summary

Figure 4.7 summarizes the actions taken by the scheduler when each type of query is received for processing and when its response is received from the database.

The scheduler obtains the version numbers for all accessed tables at the beginning of the transaction. It then tags every read and write query it sends to the databases with their appropriate version numbers.

The scheduler keeps track of the current version numbers of all replicas. It sends each read query to one of the databases where the needed versions have already been produced. The scheduler picks one of these replicas based on load balancing. Writes are sent to all replicas.

4.4 Comparison of Conflict-Aware Scheduling Using Conservative 2PL and Versioning

In this section we show that, in general, versioning leads to increased concurrency compared to conservative 2PL.

In both conservative 2PL and distributed versioning, the declaration of which tables are going to be accessed by a transaction is done at the beginning of the transaction. The behavior of the two schemes in terms of waiting for conflicts to be resolved is, however, totally different. In particular, conflict waiting times are poten-
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>obtain &amp; set version numbers for tables, respond to client</td>
</tr>
<tr>
<td>BeginResp</td>
<td>n/a</td>
</tr>
<tr>
<td>WriteQ</td>
<td>tag write with version number &amp; broadcast</td>
</tr>
<tr>
<td>WriteResp</td>
<td>respond to client if first replica response</td>
</tr>
<tr>
<td>ReadQ</td>
<td>tag read with version number(s), select replica with actual version(s) as required, send read</td>
</tr>
<tr>
<td>ReadResp</td>
<td>respond to client</td>
</tr>
<tr>
<td>Commit</td>
<td>tag with version numbers, piggyback version_release, broadcast</td>
</tr>
<tr>
<td>CommitResp</td>
<td>update state, respond to client if first replica response</td>
</tr>
</tbody>
</table>

Figure 4.7: Scheduler actions for each type of query and query response
begin
write a
write b
write c
end

Figure 4.8: Sequence of updates in a transaction

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

We illustrate the increase in concurrency with an example. Assume that transactions \(T_0\), and \(T_1\) both execute the code shown in Figure 4.8, writing three different tables. Assume also that transaction \(T_0\) is serialized before transaction \(T_1\). In conservative 2PL, transaction \(T_1\) waits for the locks on all three tables to be freed by \(T_0\) before it starts executing (see Figure 4.9). In contrast, with distributed versioning the operations on the different tables are pipelined. This example also clearly demonstrates that, in general, both features of distributed versioning (selective per-table
waiting and early availability of versions) are essential. Any single feature in isolation would produce the same behavior as conservative 2PL and thus less concurrency.

One may wonder if similar benefits are not available with alternative 2PL schemes. This is not the case. Selective waiting for locks can be achieved by implicit 2PL, in which locks are acquired immediately before each operation. Implicit 2PL achieves selective waiting, but at the expense of potential deadlocks. Given that the probability of deadlock increases approximately quadratically with the number of replicas [41], any concurrency control algorithm that allows deadlock is undesirable for large clusters. Even if a deadlock-avoidance scheme could be used in conjunction with selective waiting for locks, early releases of locks are limited by the two-phase nature of 2PL, necessary for achieving serializability.

In conclusion, version-based concurrency control offers increased concurrency compared to conservative 2PL without the danger of re-introducing deadlocks or sacrificing 1-copy serializability.

4.5 Fault Tolerance and Data Availability

In this section, we describe the fault tolerance aspects of our solution. We describe the fault tolerance and recovery algorithms assuming a conflict-aware scheduler with conservative 2PL. The algorithms are very similar for both versions of conflict-aware scheduler.
4.5.1 Fault Model

For ease of implementation, we assume a fail-stop fault model [76]. However, our fault tolerance algorithm could be generalized to more complex fault models.

4.5.2 Fault Tolerance of the Sequencer

At the beginning of each transaction, a scheduler requests a sequence number from the sequencer. Afterwards, the scheduler sends to all other available schedulers a replicate-start-transaction message containing the sequence number for this transaction, and waits for the acknowledgments. The other schedulers create a record with the sequence number and the coordinating scheduler of this transaction. No disk logging is done at this point.

If the sequencer fails, replication of the sequence numbers on all schedulers allows for restarting the sequencer with the last sequence number on another machine. In case all schedulers fail, the sequence numbers are reset everywhere (sequencer, schedulers, database proxies).

4.5.3 Atomicity and Durability of Writes

To ensure that all writes are eventually executed, regardless of any sequence of failures in the schedulers and the databases, each scheduler keeps a persistent log of all write queries of committed transactions that it handled, tagged with the transaction’s sequence number. The log is kept per table in sequence number order, to facilitate recovery (see below). To add data availability, the scheduler replicates this information in the memory of all other available schedulers. Each database proxy maintains the sequence number for the last write it committed on each table. The database proxy does not log its state to disk.

In more detail, before a commit query is issued to any database, the scheduler sends a replicate_end_transaction message to the other schedulers. This message contains the write queries that have occurred during the transaction, and the trans-
action's sequence number. All other schedulers record the writes, augment the corresponding remote transaction record with the commit decision, and respond with an acknowledgment. The originator of the `replicate_end_transaction` message waits for acknowledgments up to the point where a majority of schedulers have agreed to commit the transaction [30], then logs the write queries to disk. After the disk logging has completed, the commit is issued to the database replicas. For read-only transactions, the commit decision is replicated but not logged to disk.

**Scheduler Failure**

In the case of a single scheduler failure, all transactions of the failed scheduler for which the live schedulers do not have a commit decision are aborted. A transaction for which a commit record exists, but for which a database proxy has not yet received the commit decision is aborted at that particular replica, and then its writes are replayed. The latter case is, however, very rare.

In more detail, a fail-over scheduler contacts all available database proxies. The database proxy waits until all queued operations finish at its database, including any pending commits. The proxy returns to the scheduler the sequence number for the last committed write on each database table, and the highest sequence number of any lock request received by the database proxy. The fail-over scheduler determines all the failed scheduler's transactions for which a commit record exists and for which a replica has not committed the transaction. The reason that a replica has not committed a transaction may be either that it did not receive the transaction's lock request or that it did not receive the commit request. The first case is detected by the sequence number of the transaction being larger than the highest lock sequence number received by the proxy. In this case, all the transaction's writes and its commit are replayed to the proxy. In the second case, the fail-over scheduler first aborts the transaction and then sends the writes and the commit. The database proxy also advances its value of the highest lock sequence number received and its value of the
last sequence number of a committed write on a particular table, as appropriate.

For all other transactions handled by the failed scheduler, the fail-over scheduler sends an abort to all database proxies. The database proxies rollback the specified transactions, and also fill the gap in lock sequence numbers in case the lock with the sequence number specified in the abort was never received. The purpose of this, is to let active transactions of live schedulers with higher sequence numbered locks proceed.

Each scheduler needs to keep logs for committed writes and records of assigned lock sequence numbers until all replicas commit the corresponding transactions. Afterwards, these records can be garbage-collected.

When a scheduler recovers, it contacts another scheduler and replicates its state. In the rare case where all schedulers fail, the state is reconstructed from the disk logs of all schedulers. All active transactions are aborted on all database replicas, the missing writes are applied on all databases, and all sequence numbers are reset everywhere.

**Network Failure**

To address temporary network connection failures, each database proxy can send a “selective retransmission” request for transactions it has missed. Specifically, the database proxy uses a timeout mechanism to detect gaps in lock sequence that have not been filled in a given period of time. It then contacts an available scheduler and provides its current state. The scheduler rolls forward the database proxy including updating its highest lock sequence number.

**Database Failure**

When a database recovers from failure, its database proxy contacts all available schedulers and selects one scheduler to coordinate its recovery. The coordinating scheduler instructs the recovering database to install a current database snapshot from another
replica, with its current state. Each scheduler re-establishes its connections to the
database proxy and adds the replica to its set of available machines. The scheduler
starts sending to the newly incorporated replica at the beginning of the next trans-
action. Afterwards, the database proxy becomes up-to-date by means of the selective
retransmission requests as described in the case of network failure.

In addition, each database proxy does periodic checkpoints [2] of its database
together with the current state (in terms of the last sequence numbers of its database
tables). To make a checkpoint, the database proxy stops all write operations going out
to the database engine, and when all pending write operations have finished, it takes
a snapshot of the database and writes the new state. If any tables have not changed
since the last checkpoint, they do not need to be included in the new checkpoint.
Checkpointing is only necessary to support recovery in the unlikely case where all
database replicas fail. In this case, each database proxy reinstalls the database from
its own checkpoint containing the database snapshot of the last known clean state.
Subsequently, recovery proceeds as in the single database failure case above.

4.6 Implementation Details

This section presents implementation features common to all schedulers, but orthog-
onal to conflict-awareness: the choice of load balancing algorithm and the connection
handling method at the schedulers and database proxies.

4.6.1 Load Balancing Algorithm

The load balancing algorithm used by the schedulers in all scheduling schemes intro-
duced in this dissertation is based on off-line measurements of the execution time of
each query on an unloaded (idle) machine.

At run-time, the scheduler estimates the load on a particular back-end as the sum
of the (a priori measured) execution times of all queries outstanding to that back-
end. This algorithm called Shortest Execution Length First (SELF) was shown
to outperform classic round-robin and shortest-queue-first algorithms for dynamic content applications [4]. SELF tries to take into account the widely varying execution times for different query types. The load estimate for each back-end is updated with feed-back from the database proxy upon each reply.

4.6.2 Connection Handling

The schedulers and database proxies are both implemented with event-driven loops that multiplex requests and responses between the web server and the database replicas. We use FreeBSD’s scalable kevent primitive [58] to efficiently handle thousands of connections at each scheduler.

4.7 Summary

In this chapter, we introduce two versions of the conflict-aware algorithm, one based on conservative 2PL and one based on explicit versions (distributed versioning), and our fault tolerance mechanisms. The scheduling aspects are similar in both versions of our conflict-aware algorithm. In both versions, the scheduler maintains serializability by enforcing a total order for the execution of conflicting operations. Furthermore, in both cases, the scheduler optimizes conflict waiting time at the expense of maintaining extra state. Lastly, in both versions, the scheduler makes fine-grained (i.e., per-query) decisions. Each read query is sent based on load balancing among all up-to-date replicas. Each write and commit reply is returned as soon as the first replica responds.

Distributed versioning improves concurrency compared to conservative 2PL by selectively waiting for table versions and early production of new versions. These advantages are obtained with minimal changes to the programming model and without sacrificing the desirable deadlock-free property of conservative 2PL.

The key aspect for providing fault tolerance and data availability is that the schedulers are responsible for coordinating the completion of committed updates on the database back-ends, in the case of a scheduler, sequencer, or a back-end database
failure. To meet this goal, the completion status and all the write queries of any update transaction together with the transaction’s sequence number are maintained in a fault-tolerant and highly-available manner at the schedulers.
Chapter 5

Experimental Methods

In this chapter, we introduce the algorithms that we use for comparison, the benchmarks, the experimental environment and the simulation method.

5.1 Other Methods Used for Comparison

To better understand the benefits of the various parts of the solution, in this section, we discuss other possible versions of scheduler, with and without conflict-awareness, fine-grained scheduling and consistency guarantees. These methods fall into two main categories depending on whether serializability or weaker consistency models are used.

5.1.1 Serializable Methods

In this section, we introduce a number of other scheduling algorithms for comparison with conflict-aware scheduling. We make this comparison only for our conflict-aware scheduler with conservative 2PL. Similar comparisons are possible, however, for distributed versioning as well. By gradually introducing some of the features of conflict-aware scheduling, we are able to demonstrate what aspects of conflict-aware scheduling contribute to its overall performance. Our first algorithm is an eager replication scheme that allows us to show the benefits of asynchrony. We then look at a number of scheduling algorithms for lazy replication. Our second algorithm is a conventional scheduler that chooses a fixed replica based on load at the beginning of the transaction and executes all operations on that replica. Our third algorithm is another fixed-replica approach, but it introduces one feature of conflict-aware scheduling: it chooses as the fixed replica the one that responds first to the transaction’s
lock request rather than the least loaded one. We then move away from fixed-replica algorithms, allowing different replicas to execute reads of the same transaction, as in conflict-aware scheduling. Our fourth and final scheduler chooses the replica with the lowest load at the time of the read, allowing us to assess the difference between this approach and conflict-aware scheduling, where a read is directed to a replica without conflicts.

We refer to these scheduler algorithms as Eager, FR-L (Fixed Replica based on Load), FR-C (Fixed Replica based on Conflict), and VR-L (Variable Replica based on Load). Using this terminology, the conflict-aware scheduler would be labeled VR-C (Variable Replica based on Conflict), but we continue to refer to it as the conflict-aware scheduler.

In all algorithms, we use the same concurrency control mechanism, i.e., conservative 2PL, the same sequence numbering method to maintain 1-copy serializability, and the same load balancing algorithm (see Section 4.6.1).

**Eager Replication (Eager)**

Eager follows the algorithm described by Weikum et al. [93], which uses synchronous execution of lock requests, writes and commits on all replicas. In other words, the scheduler waits for completion of every lock, write or commit operation on all replicas, before sending a response back to the application server. The scheduler directs a read to the replica with the lowest load.

**Fixed Replica Based on Load (FR-L)**

FR-L is a conventional scheduling algorithm, in which the scheduler is essentially a load balancer. At the beginning of the transaction, the scheduler selects the replica with the lowest load. It just passes through operations tagged with their appropriate sequence numbers. Lock requests, writes and commits are sent to all replicas, and the reply is sent back to the application server when the chosen replica replies to the
scheduler. Reads are sent to the chosen replica. The FR-L scheduler needs to record only the chosen replica for the duration of each transaction.

**Fixed Replica Based on Conflict (FR-C)**

FR-C is identical to FR-L, except that the scheduler chooses the replica that first responds to the lock request as the fixed replica for this transaction. As in FR-L, all reads are sent to this replica, and a response is returned to the application server when this replica responds to a lock request, a write or a commit. The FR-C scheduler’s state is also limited to the chosen replica for each transaction.

**Variable Replica Based on Load (VR-L)**

In the VR-L scheduler, the response to the application server on a lock request, write, or commit is sent as soon as the first response is received from any replica. A read is sent to the replica with the lowest load at the time the read arrives at the scheduler. This may result in the read being sent to a replica where it needs to wait for the completion of conflicting operations in other transactions or previous operations in its own transaction.

The VR-L scheduler needs to remember for the duration of each replicated query whether it has already forwarded the response and whether all machines have responded, but, unlike a conflict-aware scheduler, it need not remember which replicas have responded. In other words, the size of the state maintained is $O(1)$ not $O(N)$ in the number of replicas.

**5.1.2 Non-serializable Methods with Specialized Consistency**

In this section, we compare the overhead of consistency maintenance for serializability against specialized consistency schemes that do not enforce serializability.

Recent work [81] addressing replica consistency in dynamic content clusters, has argued that several distinct consistency models should be supplied, since applications
have different levels of consistency requirements.

Their work builds on traditional lazy schemes, commonly employed in wide-area applications, where writes can arrive out-of-order at different sites and reads can access inconsistent data. These inconsistencies are usually addressed in ad-hoc ways. For example, reads can access only out-of-date copies that satisfy a specific staleness bound, and reconciliation is used for out-of-order writes. In this basic scheme, the user needs to solve the inconsistencies manually to ensure eventual convergence to the same data image on all replicas. To address this problem, Neptune [81] proposes two additional levels of data consistency progressing towards stronger consistency features such as ordered writes, and automatic staleness control. Then, the programmer would presumably be able to choose the appropriate consistency model that fits his application.

An extension to this idea [98] proposes a “continuum” of consistency models with tunable parameters. In this model as well, the programmer would have to define the appropriate consistency abstraction (e.g., the consistency unit and appropriate parameters for the consistency model) for each individual application.

In the following, we describe the three consistency levels specified in Neptune [81], and the types of dynamic content web sites for which they are suitable. We further extend these consistency models with an additional model designed to incorporate features from the continuous consistency model spectrum [98].

**Level 0. Write-anywhere replication.**

This is the basic lazy consistency scheme that offers no ordering or consistency guarantees. Writes that arrive out-of-order are not reconciled later. This scheme is only applicable to simple services with append-only, commutative or total-updates such as an e-mail service.
Level 1. Ordered writes.

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

Level 2. Ordered writes and staleness control for reads.

Writes are totally ordered at all replicas, and reads satisfy the following two criteria:

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

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

Special. Per interaction or per object consistency

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

Implementation of Loose Consistency Methods

For Levels 0, 1 and 2, we remove any transaction delimiters and other annotations from the application code. The scheduler and database proxy are modified as follows.
For Level 0, we remove any checks pertaining to in-order delivery of writes at the database proxy. The database proxy still implements conflict resolution, but all writes are handled in the order of their arrival, which may be different at different replicas. No version numbers are used. The scheduler load balances reads among all database replicas.

To implement Level 1, the scheduler obtains version numbers for each write, and the database proxies deliver the writes in version number order, as in distributed versioning. No version numbers are assigned to reads. The scheduler load balances reads among all database replicas.

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

The implementation of Special consistency models is application-specific, and its implementation is deferred to Section 5.2 where we discuss application benchmarks.

5.2 Benchmarks

5.2.1 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council (TPC) [91] is a transactional web benchmark designed for evaluating e-commerce systems. The performance metric reported by TPC-W is the number of web interactions per second. Several interactions are used to simulate the activity of a retail store.

We implement all the functionality specified in TPC-W that has an impact on per-
formance, including transactional consistency. We do not implement some functionality specified in TPC-W that has an impact only on price and not on performance, such as the requirement to provide enough storage for 180 days of operation.

The database size is determined by the number of items in the inventory and the size of the customer population. We use 100,000 items and 2.8 million customers which results in a database of about 4 GB. This database is, in fact, comparable or larger than those used for the official results posted at the TPC site. The inventory images, totaling 1.8 GB, are resident on the web server. Table 5.1 lists the database tables and their sizes in our test-bed. The sizes include that of the necessary indexes on each of the tables to make the queries in the interactions efficient.

We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. The read-only interactions include access to the home page, listing of new products and best-sellers, requests for product detail, and two interactions involving searches.

Read-write interactions include user registration, updates of the shopping cart,
two order-placement interactions, two involving order inquiry and display, and two involving administrative tasks. The order-placement transactions (see Figure 5.1) update most frequently accessed tables (Orders, OrderLine, Item, CreditInfo). We use the same distribution of script execution as specified in TPC-W. An interaction may also involve requests for multiple embedded images, each image corresponding to an item in the inventory. With one exception, all interactions query the database server. Most interactions exhibit significant data locality in their access patterns (e.g., top-k best-seller, top-k published items and top-k recent orders). We implement each of the interactions as a separate PHP script.

TPC-W simulates three different interaction mixes by varying the ratio of read-only to read-write scripts: browsing, shopping, and ordering. The browsing mix contains 95% read-only scripts, the shopping mix 80%, and the ordering mix 50%.

Table 5.2 lists all the 14 different interactions and their proportions in the different profiles. The column Time, in the table refers to the average time (in milliseconds) measured for each interaction on an unloaded machine. This gives an idea of the relative complexity of the interactions. We can see that the complexity of the interactions varies widely, with interactions taking between 20 ms and 700 ms on an unloaded machine and read-only interactions up to 30 times more heavyweight than read-write interactions. The weight of a query (and interaction) is the same for a given query type largely independent of the arguments. Each interaction also involves requests for multiple embedded images, each image corresponding to an item in the inventory. Except for Order Inquiry, all interactions query the database server.

For TPC-W we implement a Special consistency model. This model follows the specification of TPC-W, which allows for some departures from 1-copy serializability. In more detail, the specification requires that all update interactions respect serializability. Read-only interactions on the retail inventory (i.e., best-sellers, new products, searches and product detail interactions) are allowed to return data that is at most 30 seconds old. Read-only interactions related to a particular customer (i.e., home and
<table>
<thead>
<tr>
<th>Web Interaction</th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Browse</strong></td>
<td>95%</td>
<td>80%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>Home</td>
<td>29.00%</td>
<td>16.00%</td>
<td>9.12%</td>
<td>26</td>
</tr>
<tr>
<td>New Products</td>
<td>11.00%</td>
<td>5.00%</td>
<td>0.46%</td>
<td>241</td>
</tr>
<tr>
<td>Best Sellers</td>
<td>11.00%</td>
<td>5.00%</td>
<td>0.46%</td>
<td>701</td>
</tr>
<tr>
<td>Product Detail</td>
<td>21.00%</td>
<td>17.00%</td>
<td>12.35%</td>
<td>25</td>
</tr>
<tr>
<td>Search Request</td>
<td>12.00%</td>
<td>20.00%</td>
<td>14.53%</td>
<td>22</td>
</tr>
<tr>
<td>Search Results</td>
<td>11.00%</td>
<td>17.00%</td>
<td>13.08%</td>
<td>288</td>
</tr>
<tr>
<td><strong>Order</strong></td>
<td>5%</td>
<td>20%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>Shopping Cart</td>
<td>2.00%</td>
<td>11.60%</td>
<td>13.53%</td>
<td>37</td>
</tr>
<tr>
<td>Customer Registration</td>
<td>0.82%</td>
<td>3.00%</td>
<td>12.86%</td>
<td>19</td>
</tr>
<tr>
<td>Buy Request</td>
<td>0.75%</td>
<td>2.60%</td>
<td>12.73%</td>
<td>47</td>
</tr>
<tr>
<td>Buy Confirm</td>
<td>0.69%</td>
<td>10.20%</td>
<td>10.18%</td>
<td>40</td>
</tr>
<tr>
<td>Order Inquiry</td>
<td>0.30%</td>
<td>0.75%</td>
<td>0.25%</td>
<td>3</td>
</tr>
<tr>
<td>Order Display</td>
<td>0.25%</td>
<td>0.66%</td>
<td>0.22%</td>
<td>77</td>
</tr>
<tr>
<td>Admin Request</td>
<td>0.10%</td>
<td>0.10%</td>
<td>0.12%</td>
<td>21</td>
</tr>
<tr>
<td>Admin Confirm</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.11%</td>
<td>4869</td>
</tr>
</tbody>
</table>

Table 5.2: Web interaction mix and characteristics
input: customer_id
tables: orders (write), order_line (write), item (write), address (read),
credit_card (write)

insert new order in orders table
for all items in shopping_cart
{
    place an order for the item in order_line
    adjust item_stock field in item
}
read the customer’s address
insert credit card info in credit card table

Figure 5.1: Example TPC-W transaction: Placing an order
order display interactions) are required to return up-to-date data. Even if allowed to read stale data, all queries need to respect the atomicity of the update transactions that they conflict with. We add a number of ad-hoc rules to the scheduler to implement this Special consistency model.

### 5.2.2 Auction Site Benchmark

Our auction site benchmark implements the core functionality of an auction site: selling, browsing and bidding. We do not implement complementary services like instant messaging or newsgroups. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, during a buyer session, users can bid on items and consult a summary of their current bid, rating and comments left by other users.

The database contains seven tables: users, items, bids, buy_now, comments, categories and regions. The users table records contain the user’s name, nickname, password, region, rating and balance. Besides the category and the seller’s nickname, the items table contains the name that briefly describes the item and a more extensive description, usually an HTML file. Every bid is stored in the bids table, which includes the seller, the bid, and a max_bid value used by the proxy bidder (a tool that bids automatically on behalf of a user). Items that are directly bought without any auction are stored in the separate buy_now table. The comments table records a comment from one user about another. As an optimization, the number of bids and the amount of the maximum bids are stored with each item to prevent many expensive lookups of the bids table. This redundant information is necessary to keep an acceptable response time for browsing requests. As users only browse and bid on items that are currently for sale, we split the item table in a new and an old item table. The very vast majority of the requests access the new items table, thus reducing
considerably the working set used by the database.

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

We use a workload mix that includes 15% read-write interactions. This mix is the most representative of an auction site workload according to an earlier study of eBay workloads [81].

We sized our system according to some observations found on the eBay web site. We always have about 33,000 items for sale, distributed among 40 categories and 62 regions. There is an average of 10 bids per item, or 330,000 entries in the bids table. The buy-now table is small, because less than 10% of the items are sold without auction. The users table has 1 million entries. The number of active users is low compared to the total number of registered users. Finally, we assume that users give feedback for 95% of the transactions. Thus, the comments table contains about 31,500 users comments. The total size of the database, including indices is 1.4GB.

Although it has been argued that an auction site can be supported by a Level 2 consistency model, as described in Section 5.1.2, program modifications are necessary to ensure correct outcome of the auction site with Level 2 consistency. The problem is that the code in several places relies on atomic sequences, which are no longer available in the absence of transactions. For instance, suppose we do not use a transaction for placing a bid. In the transaction for placing a bid (see Figure 5.2), the maximum bid is first read from the item table and then updated if the input bid is acceptable (higher). If reading and updating the maximum bid for an item are not done in a critical section, then if two clients submit bids concurrently, they can both read the same maximum bid value for that item. Assuming that both bids are higher, both will be accepted, and the maximum bid stored in the items table for that item could be wrong (e.g., the lower one of the new bids). Thus, additional code is necessary to
input: new_bid value, item_id for item on auction

tables: bids (write), items (write)

read the maximum bid max_bid for item from items
if (new_bid > max_bid)
{
    update the max_bid field of item
    update the number of bids field of item
    insert the new bid into bids
}

Figure 5.2: Example auction transaction: Placing a bid

verify the correctness of the maximum bid.

5.2.3 Bulletin Board Benchmark

Our bulletin board benchmark is modeled after an online news forum like Slashdot [85]. We implemented the essential bulletin board features of the Slashdot site in PHP. In particular, as in Slashcode [84], we support discussion threads. A discussion thread is a logical tree, containing a story at its root, and a number of comments for that story, which may be nested. Users have two different levels of authorized access: regular user and moderator. Regular users browse and submit stories and comments. Moderators review stories and rate comments.

The main tables in the database are the users, stories, comments, and submissions tables. The users table contains each user’s real name and nickname, contact information (email), password, level of authorized access, and user rating. The stories table contains each story’s title and body, the nickname of the story’s author, the
date the story was posted, the number of comments at the outermost nesting level, and the category the story fits under. The categories table contains the same categories as the Slashdot site. The comments table contains the comment’s subject and body, the nickname of the comment’s author, the date the comment was posted, the identifier of the story or the parent comment it belongs to, and a comment rating. Each story submission is initially placed in the submissions table, unless submitted by a moderator. We maintain a moderator log table, which stores the moderator ratings for comments. Regular user ratings are computed based on the ratings of the comments they have posted.

Stories and comments are maintained in separate new and old tables. In the new stories table we keep the most recent stories with a cut-off of one month. We keep old stories for a period of three years. The new and old comments tables correspond to the new and old stories respectively. The majority of the browsing requests are expected to access the new stories and comments tables, which are much smaller and therefore much more efficiently accessible. A daemon is activated periodically to move stories from the new to the old tables as appropriate.

We have defined ten web interactions. The main ones are: generate the stories of the day, browse new stories, older stories, or stories by category, show a particular story with different options on filtering comments, search for keywords in story titles, comments and user names, submit a story, add a comment, and review submitted stories and rate comments at moderator level.

We use a workload mix which contains 85% read-only interactions, with the remaining 15% being story and comment submissions and moderation interactions. This mix corresponds to the maximum posting activity of an active newsgroup as observed by browsing the Internet for typical breakdowns of url requests in bulletin boards sites [3].

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

Among the loose consistency models discussed in Section 5.1.2, the normal semantics of bulletin boards can be supported by the Level 1 consistency model.

5.2.4 Client Emulation

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

5.3 Experimental Environment

Hardware

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

We use three popular open source software packages: the Apache web server [1], the PHP web-scripting/application development language [69], and the MySQL database server [62]. Since PHP is implemented as an Apache module, the web server and application interpreter coexist on the same machine(s).

We use Apache v.1.3.22 [1] for our web server, configured with the PHP v.4.0.1 module [69] providing server-side scripting for generating dynamic content. We use MySQL v.4.0.1 [62] with InnoDB transactional extensions as our database server.

5.4 Simulation Method

To study the scaling of our techniques on a larger number of nodes, we developed two configurable cluster simulators: one for the web/application server front-ends and the other for the database back-ends. We use these front-end and back-end simulators to drive actual execution of schedulers and database proxies. Our motivation for running the real scheduler and database proxy code is to measure their overheads experimentally and determine whether or not the scheduler stage becomes a bottleneck for large clusters.

Each simulator models a powerful server of the given type (web server or database) equivalent to running a much larger number of real-life servers. The web server simulator takes each HTTP request generated by the client emulator and sends the queries embedded in the corresponding scripts with dummy arguments filled in to one of the schedulers, one query at a time. As before, the scheduler parses each query, selects a back-end and passes the query type and tables accessed, and actual query to the database proxy, which performs any necessary conflict resolution and reordering and passes the query to the database simulator.

The database simulator maintains a separate queue for each simulated back-end. Whenever a new query is received, a record is placed on the corresponding queue for the back-end the query was for. The record contains a future termination time for
the query using a cost estimate for its query type. The costs for the query processing at the database were derived by performing off-line measurements of each type of query on an unloaded machine. The database simulator polls the queues and sends responses when the simulated time reaches the termination time for each query. Based on profiling of real runs we estimate that the disk access time that is not overlapped with CPU time is negligible, thus disk accesses are not modeled. This is partly due to the locality in the application with few resulting disk accesses for reads, and partly due to the lazy commits in our database system.

Calibration of the simulated system against measurement of the real 8-node database cluster shows that the simulated throughput numbers are within 12% of the experimental numbers for all workloads.

5.5 Summary

This chapter describes the experimental methodology used in our evaluation: the algorithms used for comparison, the benchmarks and the measurement techniques.

To better understand the benefits of the various parts of our solution, we discuss other possible scheduling algorithms with combinations of some of the features in the conflict-aware algorithm.

We use three benchmarks representative of an e-commerce site, an auction site and a bulletin board site, respectively. In the implementation of these benchmarks we have followed an industry standard specification (TPC-W) and two real sites (eBay.com and slashdot.org).

In order to be able to measure the overheads of our conflict-aware solution, we define several loose consistency models. These models have been recently proposed with the goal to capture the best possible scaling-consistency trade-off, at the cost of potentially significant programming effort. For each application we define a specialized consistency model which has been tailored to the consistency requirements of that application.
Chapter 6

Experimental Results

In this chapter we show how the scaling of conflict-aware replication compares to the scaling of conflict-oblivious and eager replication schemes that provide the same consistency guarantees (1-copy serializability). We also compare the scaling of our conflict-aware replication solution to the upper bound scaling of a traditional lazy scheme without any consistency guarantees.

In more detail, we analyze the benefit of the various factors contributing to performance in our conflict-aware protocol (i.e., lazy replication, conflict avoidance and fine-grained scheduling), by comparing conflict-aware scheduling with all alternative serializable schedulers presented in Chapter 5, which contain different subsets and combinations of these features. The functionality of each individual conflict-aware feature in combination with either conservative 2PL or explicit versioning is very similar. Hence, we present this comparison only for the conflict-aware scheduler with conservative 2PL. We also quantify the overheads of our conflict-aware solution by comparing the performance of our best conflict-aware scheduler (with explicit versioning) to the performance of the loose consistency methods introduced in Section 5.1.2.

The results belong to two categories: experimental and simulated. We present experimental results for small clusters. We then extend our evaluation by simulation to larger clusters, and different database speeds to investigate whether the scheduler tier becomes a bottleneck. We also re-evaluate the benefits of conflict-awareness compared to conflict-oblivious and eager replication with different conflict rates by simulation.

The chapter is structured as follows.
Section 6.1 presents preliminary experiments on a configuration with one front-end running the web and application servers and one database back-end in order to capture basic workload characteristics and architectural bottlenecks.

In Section 6.2 we present an experimental comparison of our conflict-aware algorithm with conservative 2PL and all alternative schedulers with serializability using small clusters of up to 8 database nodes.

We then extend our evaluation of conflict-aware scheduling by simulation to larger clusters of up to 60 database nodes, different conflict rates and database speeds and present these results in Section 6.3.

In Section 6.4, we compare the two versions of conflict-aware algorithm described in Chapter 4 to show the benefit that explicit versioning brings compared to conservative 2PL.

Next, in Section 6.5, we investigate the overheads of our conflict-aware solution by comparing distributed versioning with the loose consistency methods.

Finally, we investigate the cost of providing fault tolerance and data availability in conflict-aware replication in Section 6.6.

6.1 Baseline Experiments

We run each benchmark with one web/application server front-end and one database engine back-end. A scheduler is not necessary in this configuration. There is no measurable difference in terms of throughput, however, when we interpose a scheduler between the front-end and the database back-end.

Figure 6.1 presents the CPU utilization on the web/application server machine and the database server machine for each of the three benchmarks. We see that for TPC-W, for all the three interaction mixes, the database server is the bottleneck, while for the bulletin board the web/application server is the bottleneck, and for the auction site the loads of the web/application server and database are closely matched. For the TPC-W ordering mix and the auction site benchmark the CPU utilization on
Figure 6.1: CPU utilization for the web server and database server in the 1-1 configuration for the TPC-W browsing, shopping and ordering mixes, the bulletin board, and auction site benchmarks.

the database server machine does not reach 100% due to lock waiting times in these workloads.

Lastly, in Figure 6.2, we show a breakdown of the average weight of reads, writes and idle time at the database as percentage of total database time. This characterization helps predict the success of scaling through replication and conflict avoidance of each application. In particular, for some of the workloads, the fraction of write processing is very small (not visible) compared to the read processing time (the TPC-W browsing and shopping mixes). The high ratio of read to write processing time predicts a limit due to saturation with writes at a high number of databases.

Furthermore, for the workloads where conflicts cannot be overlapped in the one-node case (TPC-W ordering and auction site), it is unlikely that conflict-avoidance will be able to overlap these conflicts in the replicated case either. The idle time is due to the fact that conflicts occur on frequently used tables such as the item table (for both TPC-W and auction).
Figure 6.2: Fraction of writes, reads and idle time at the database server in the 1-1 configuration for the TPC-W browsing, shopping and ordering mixes, the bulletin board, and auction site benchmarks

6.2 Experimental Comparison of Serializable Methods for Conservative 2PL

We present an experimental comparison of our conflict-aware algorithm using conservative 2PL and all alternative serializable schedulers introduced in Chapter 5.

The experimental numbers are obtained running a prototype implementation of our dynamic content server on a cluster of 1 to 8 database server machines.

As part of this comparison, we first present a measured throughput comparison in terms of the number of web interactions per second (Wips), the standard TPC-W performance metric. For a given number of machines we report the peak throughput. In other words, we vary the number of clients until we find the peak throughput and we report the average throughput number over several runs.

We use a number of web server machines sufficient for the web server stage not to be the bottleneck. The largest number of web server machines used for any experiment with TPC-W was 8. For the bulletin board, we use three web servers to saturate each database engine (12 web servers and 4 databases in the largest configuration). For the auction site the largest configuration consists of 8 web servers and 6 database
Figure 6.3: Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the TPC-W browsing mix

engines (i.e., slightly more web servers than databases are necessary to achieve peak performance). We use two schedulers to ensure data availability in all configurations.

We further report average response time at the peak throughput. Next, we break down the average query time into query execution time, and waiting time (for locks and previous writes in the same transaction).

6.2.1 Experimental Throughput Comparison

Figures 6.3 through 6.7 show the throughput of the various scheduling algorithms for each of the three applications. In the x-axis we have the number of database machines, and in the y-axis the number of web interactions per second.

First, conflict-aware scheduling outperforms all other algorithms, and increasingly so for workload mixes with a large fraction of writes. Second, all asynchronous schemes outperform the eager scheme, again increasingly so as the fraction of writes increases. In particular, the conflict-aware protocol outperforms the eager protocol by factors of 1.7, 2.4 and 3.5 for the TPC-W browsing, shopping, and ordering mix, respectively at eight replicas, and a factor of 2.6 for the auction benchmark at 6
Figure 6.4: Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the TPC-W shopping mix

Figure 6.5: Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the TPC-W ordering mix
Figure 6.6: Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the auction site benchmark

Figure 6.7: Throughput comparison: The benefits of conflict avoidance and fine-grained scheduling for the bulletin board benchmark
replicas. Since the bulletin board application has no transactions, there is little benefit from conflict avoidance. In spite of extra processing for ordering of writes and conflict avoidance in the conflict aware protocol compared to the eager protocol, the conflict-aware protocol is still able to achieve a 10% gain over Eager. Third, for the fixed replica algorithms, choosing the replica by conflict (FR-C) rather than by load (FR-L) provides substantial benefits, a factor of 1.4, 1.4, and 1.25 for the largest configuration, for each of the three mixes of TPC-W, respectively, and a factor of 1.5 for the auction site benchmark. Fourth, variable replica algorithms provide better results than fixed replica algorithms, with the conflict-aware scheduler showing a gain of a factor of 1.5, 1.6 and 2, for TPC-W browsing, shopping, and ordering, respectively, and a factor of 1.8 for the auction site benchmark compared to FR-L, at 8 replicas. Finally, FR-C performs better than VR-L for the TPC-W browsing and the shopping mix and for the auction site benchmark, but becomes worse for the ordering mix. Because of the larger numbers of reads in the TPC-W browsing and shopping mixes and the auction site benchmark, VR-L incurs a bigger penalty for these mixes by not sending the reads to a conflict-free replica, possibly causing them to have to wait. FR-C, in contrast, tries to send the reads to conflict-free replicas. In the TPC-W ordering mix, reads are fewer. Therefore, this advantage for FR-C becomes smaller. VR-L’s ability to shorten the write and commit operations by using the first response becomes the dominant factor.

6.2.2 Experimental Response Time Comparison

Figure 6.8 presents a comparison of the average web interaction response time for the five scheduler algorithms at peak throughput for the largest corresponding configuration of each workload. For each workload, the results are normalized to the average response time of the eager scheduler for that workload.

These results show that the conflict-aware scheduler provides better average response times than all other schedulers. The performance benefits of reducing conflict
Figure 6.8: Web interaction response time for all schedulers and all workloads, normalized to response time of the Eager scheduler.

Waiting times are reflected in response time reductions as well, with the same relative ranking for the different protocols as in the throughput comparison.

6.2.3 Breakdown of Query Time

Figures 6.9 to 6.11 show a breakdown of the query response time into query execution time and waiting time for the three mixes of TPC-W. The waiting time is mostly due to conflicts; waiting for previous writes in the same transaction is negligible. For each workload, the results are normalized to the average query response time for the eager scheduler for that workload.

These results further stress the importance of reducing conflict waiting time. For all protocols, and all workloads, conflict waiting time forms the largest fraction of the query time. Therefore, the scheduler that reduces the conflict waiting time the most performs the best in terms of overall throughput and response time. The differences in query execution time between the different protocols are minor and do not significantly influence the overall throughput and response time. One might, for instance, expect the conflict-aware scheduler to produce worse query execution times than VR-L, because of the latter’s potential for better load balancing. VR-L has
Figure 6.9: Query time breakdown for all schedulers for the TPC-W browsing mix, normalized by the query time of the Eager scheduler.

Figure 6.10: Query time breakdown for all schedulers for the TPC-W shopping mix, normalized by the query time of the Eager scheduler.

Figure 6.11: Query time breakdown for all schedulers for the TPC-W ordering mix, normalized by the query time of the Eager scheduler.
the opportunity to direct reads to all replicas in order to balance the load, while the conflict-aware scheduler directs reads only to conflict-free replicas. In practice, however, the positive effect of this extra degree of freedom is minimal and completely overwhelmed by the conflict-aware scheduler's reduced conflict waiting times.

6.3 Simulation of Serializable Methods for Conservative 2PL

We use simulation to extrapolate from our experimental results in three different directions. First, we explore how throughput scales if a larger cluster of database replicas is available. Second, we investigate the effect of faster databases, either by using faster database software or faster machines. Third, we show how the performance differences between the various schedulers evolve as the conflict rate is gradually reduced.

6.3.1 Large Database Clusters

Results

We simulate all five schedulers for all three applications for database cluster sizes up to 60 replicas. As with the experimental results, for a given number of replicas, we increase the number of clients until the system achieves peak throughput, and we report those peak throughput numbers. The results can be found in Figures 6.12 to 6.16. In the x-axis we have the number of simulated database replicas, and in the y-axis the throughput in web interactions per second.

The simulation results show that the experimental results obtained on small clusters can be extrapolated to larger clusters. In particular, the conflict-aware scheduler outperforms all other schedulers, and the benefits of conflict awareness grow as the cluster size grows, especially for the TPC-W shopping, the TPC-W ordering mix and the auction site benchmark. Furthermore, the relative order of the different schedulers remains the same, and, in particular, all lazy schemes outperform the eager scheduler (by up to a factor of 4.4 for the conflict-aware scheduler). The results for
Figure 6.12: Simulated throughput results for the TPC-W browsing mix

Figure 6.13: Simulated throughput results for the TPC-W shopping mix
Figure 6.14: Simulated throughput results for the TPC-W ordering mix

Figure 6.15: Simulated throughput results for the auction benchmark
the TPC-W shopping mix deserve particular attention, because they allow us to observe a flattening of the throughput of FR-C and VR-L as the number of machines grows, a phenomenon that we could not observe in the actual implementation. In contrast, throughput of the conflict-aware protocol continues to increase, albeit at a slower pace. With increasing cluster size, the number of conflicts increases [41]. Hence, choosing the replica based on a single criterion, either conflict (as in FR-C) or fine-grained load balancing (as in VR-L), is inferior to conflict-aware scheduling that combines both.

**Bottleneck Analysis**

As the cluster scales to larger numbers of machines, the following phenomena could limit throughput increases: growth in the number of conflicts, each replica becoming saturated with writes, or the scheduler becoming a bottleneck. In this section we show that the flattening of the throughput curves for the conflict-aware scheduler in Figures 6.13 and 6.14 is due to conflicts among transactions, even though the scheduler seeks to reduce conflict waiting time. A fortiori, for the other schedulers,
Figure 6.17: Breakdown of the average database CPU time into idle time, time for reads, and time for writes, for the TPC-W browsing, shopping and ordering mixes.

Figure 6.18: Breakdown of the average database CPU time into idle time, time for reads, and time for writes, for the bulletin board and auction benchmarks.

which invest less effort in reducing conflicts, conflicts are even more of an impediment to good performance at large cluster sizes.

Using the conflict-aware scheduler, Figures 6.17 and 6.18 shows the breakdown of the average database CPU time into idle time, time processing reads, and time processing writes. The breakdown is provided for each workload, for one replica and for either the largest number of replicas simulated for that workload or for a number of replicas at which the throughput curve has flattened out.

For the TPC-W browsing mix, which still scales at 60 replicas, idle time remains low even at that cluster size. For the TPC-W shopping mix, which starts to see
some flattening out at 60 machines, idle time has grown to 15%. For the TPC-W ordering mix, which does not see any improvement in throughput beyond 16 replicas, idle time has grown considerably, to 73%. For the auction benchmark, the idle time at the saturation point (40 databases) is 55%. In contrast, in the bulletin board application, where there are no transactions, idle time at the largest configuration is negligible.

For the TPC-W benchmark, the fraction of write (non-idle) time grows from under 1% for browsing and shopping and 6% for ordering on one replica, to 8% for the browsing mix (at 60 replicas), 30% in the shopping mix (at 60 replicas), and 16% for the ordering mix (at 16 replicas).

For the other two benchmarks, the fraction of write time grows from negligible at one replica to 58% for the bulletin board application and 40% for the auction site benchmark for their respective largest configurations.

Idle time is entirely due to conflicts. Idle time due to load imbalance is negligible. Most idle time occurs on a particular replica when a transaction holds locks on the database that conflict with all lock requests in the proxy’s lock queues, and that transaction is in the process of executing a read on a different replica. Additional idle time occurs while waiting for the next operation from such a transaction. The results in Figure 6.17 clearly show that idle time due to conflicts is the primary impediment to scaling. Write saturation (a replica being fully occupied with writes) does not occur.

In Table 6.1, we show the memory, disk, and network usage at the scheduler for the TPC-W shopping mix, the bulletin board, and the auction site for the largest respective configurations. One scheduler is used for the TPC-W shopping mix and the auction site, and two schedulers are used for the bulletin board benchmark. We see that the scheduler CPU is the resource with the highest usage, while all other resource usage is very low.

The CPU usage of a single scheduler reaches 58% for the TPC-W shopping mix, for
### Table 6.1: Average resource usage at the scheduler for the TPC-W shopping mix, the bulletin board and the auction site at the largest respective configurations. One scheduler is used for the TPC-W and auction benchmarks, two schedulers are used for the bulletin board benchmark.

<table>
<thead>
<tr>
<th>App</th>
<th>CPU (%)</th>
<th>Memory (MB)</th>
<th>Network (MB/sec)</th>
<th>Disk (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-W</td>
<td>58%</td>
<td>6</td>
<td>3.8</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>BB</td>
<td>50%</td>
<td>5</td>
<td>6.5</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Auction</td>
<td>71%</td>
<td>5</td>
<td>1.0</td>
<td>&lt; 1</td>
</tr>
</tbody>
</table>

The largest configuration with 60 replicas. For the auction benchmark, the scheduler CPU usage is 71%. Due to the higher loads that the system sustains at the largest configuration for the bulletin-board application, two schedulers become necessary in this configuration.

#### 6.3.2 Faster Replicas

If the database is significantly faster, either by using more powerful hardware or by using a high-performance database engine, a conflict-aware scheduler continues to provide good throughput scaling. In Figure 6.19 we show throughput as a function of the number of replicas for databases twice and four times faster than the MySQL database we use in the experiments. * We simulate faster databases by reducing the estimated length of each query in the simulation. Figure 6.19 shows that the faster databases produce similar scaling curves with correspondingly higher throughputs. Three schedulers are necessary in the largest configuration with the fastest database.

*This is the highest speed of database for which we could simulate a cluster with 60 replicas.*
6.3.3 Varying Conflict Rates

Table-level conservative 2PL, as used in our implementation, causes a high conflict rate. We investigate the benefits of conflict-aware scheduling under conflict rates as low as 1\% of that observed in the experimental workload. Figures 6.20 and 6.21 compare the throughput as a function of cluster size between the conflict-aware scheduler on one hand, and the eager and the conventional lazy FR-L schedulers on the other hand. We vary the conflict rate from 1\% to 100\% of the conflict rate observed in the experimental workload. In particular, if a table-level conflict occurs, we ignore the conflict with the specified probability.

Obviously, the performance differences become smaller as the number of conflicts decreases, but at a 1\% conflict rate and at the largest cluster size, the conflict-aware protocol is still a factor of 1.8 better than Eager, and a factor of 1.3 better than a lazy protocol without any optimizations (FR-L). This result demonstrates that conflict-awareness continues to offer benefits for workloads with lower conflict rates or systems with finer-grain concurrency control.
Figure 6.20: Simulated throughput results for various conflict rates for the conflict-aware and the Eager scheduler for the TPC-W shopping mix

Figure 6.21: Simulated throughput results for various conflict rates for the conflict-aware and the FR-L scheduler for the TPC-W shopping mix
6.4 Comparison of Conflict Awareness Using Conservative 2PL versus Versioning

This section shows the comparison between a conflict-aware scheduler using conservative 2PL, and a conflict-aware scheduler using explicit versioning (distributed versioning). We use the simulation environment with up to 60 database servers for this comparison and throughput as the comparison metric.

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

Figures 6.22 through 6.24 compare distributed versioning to conservative 2PL for the TPC-W shopping mix, the TPC-W ordering mix, and the auction site, respectively. We do not show results for the TPC-W browsing mix and the bulletin board, because the conflict rates are too low for there to be any difference between distributed versioning and conservative 2PL. These and all further graphs have in the x axis the number of database engines, and in the y axis the throughput in (client) web interactions per second. The graphs also show two protocols that only use a subset of the features of distributed versioning. \textit{Dversion - EarlyRel} uses version numbers to selectively wait for the right version of tables, but it does not produce a new version of the tables until commit. Vice versa, \textit{DVersion - LateAcq} waits for the correct versions of all tables at the beginning of the transaction, but produces new versions immediately after the transaction’s last use of a particular table. The results clearly confirm the discussion in Section 4.4. With increased conflict rates, distributed versioning produces superior throughput compared to conservative 2PL. Both features of distributed versioning, selective waiting for table versions and early production of new versions, are essential. Without either one of them, improvement over conservative 2PL is minimal.
Figure 6.22: Comparison of distributed versioning and conservative 2PL for the TPC-W shopping mix

Figure 6.23: Comparison of distributed versioning and conservative 2PL for the TPC-W ordering mix
6.5 Comparison of Specialized Consistency and Serializability

In this section, we investigate the overhead of consistency maintenance for maintaining serializability in our conflict-aware protocol. For this purpose, we compare the best version of our protocol (distributed versioning) with all other protocols introduced in Chapter 5 for all levels of consistency including specialized, and looser than required for each of the three applications described in Chapter 5.2. This allows us to detect the overhead of various parts of our solution. For example, we can detect the overhead of in-order delivery for writes or the success of our conflict avoidance and reduction techniques by comparison to the upper bound obtained by assuming that these overheads (for ordering writes or resolving read-write conflicts) do not exist.

Figures 6.25 through 6.29 show a comparison of the throughput between the distributed versioning protocol (DVersion) and various lazy replication methods. As a baseline, we also include the Eager protocol. These figures allow us to draw the following conclusions.

First, for all applications, the differences between Levels 0, 1, and 2 are negligible.
Figure 6.25: Comparison of all consistency levels for the TPC-W browsing mix. Special is the specialized consistency level.

Figure 6.26: Comparison of all consistency levels for the TPC-W shopping mix. Special is the specialized consistency level.
Figure 6.27: Comparison of all consistency levels for the TPC-W ordering mix. Special is the specialized consistency level.

Figure 6.28: Comparison of all consistency levels for the bulletin board. Level 1 is the specialized consistency level.
Second, for the workloads with low conflict rates (i.e., TPC-W browsing and bulletin board), there is no difference between any of the protocols. Third, as the conflict rate increases, there is a growing difference between Levels 0, 1, and 2, on one hand, and DVersion on the other. For the largest simulated configuration, these differences are 5%, 25% and 50%, for the TPC-W shopping mix, the auction site, and the TPC-W ordering mix, respectively. Fourth, the differences between the Special consistency model and DVersion are small for all workloads of TPC-W. Finally, for the bulletin board, which has no transactional requirements, the overhead of enforcing 1-copy serializability is minimal.

In a cluster, messages usually arrive in the order that they are sent, and the delay in message delivery is low. Furthermore, interactions between the client and the database are such that at least one database replica must have completed the previous request before the next one is issued. A conflict-aware scheduler directs this next request precisely to that replica. These observations explain the small differences in performance between Levels 0, 1 and 2 for all applications, and between DVersion and Special for TPC-W. Loose consistency models show a benefit, instead,
when transactional atomicity is removed, and hence the cost of waiting for read-write conflicts is alleviated. As the number of such conflicts increases, the benefit of loose consistency models grows. Among our applications, this is the case for the TPC-W shopping mix, the auction site, and the TPC-W ordering mix. These results should be viewed with the caveat that, as explained in Section 5.2, these looser consistency models do not, by themselves, provide the right semantics for these applications. Additional programming effort is required to achieve the right semantics, possibly entailing additional execution time overhead.

6.6 Cost of Fault Tolerance and Data Availability

Figure 6.30 shows the throughput of conflict-aware scheduling without fault tolerance (Unreliable), with the overhead of logging to disk (Reliable), and with logging to disk plus replicating the state using two and three schedulers (Reliable-2 and Reliable-3, respectively). The results for each workload are normalized to the throughput of the Unreliable scheduler for that workload. All the measurements were done using the experimental platform with the largest configuration for each workload at the peak throughput. The measurements from Sections 6.2.1 through 6.2.3 correspond to the Reliable-2 bar.

The overheads for fault tolerance and data availability range from negligible for the TPC-W browsing and shopping mixes to around 16% for the TPC-W ordering mix. This overhead is mainly due to logging, with no extra overhead when replication of writes to remote schedulers is added to logging (as seen by comparing the Reliable and Reliable-2 bars).

A full checkpoint currently takes 5 minutes to perform for our 4 GB database. We did not include this overhead in our above measurements, because well known techniques for minimizing the time for taking file snapshots exist [46].
Figure 6.30: Overhead of providing fault tolerance and various degrees of availability for the conflict-aware scheduler for all workloads. All results represent throughput normalized to the throughput of the Unreliable scheduler for that workload.

6.7 Summary

We have evaluated conflict-aware scheduling, both by measurement of an implementation and by simulation. In an 8-node cluster, the conflict-aware scheduler brings factors of 1.5, 1.6, 2 and 1.9 in throughput improvement, compared to a conflict-oblivious lazy scheduler, and factors of 1.7, 2.4, 3.5 and 2.6 compared to an eager scheduler, for the browsing, shopping and ordering mixes of TPC-W, and the auction benchmark respectively. For the bulletin board benchmark the benefits are smaller (up to 10%).

Furthermore, our simulations show that conflict-aware schedulers scale well to larger clusters and faster machines, and that they maintain an edge over eager and conflict-oblivious schedulers even if the conflict rate is much lower.

Distributed versioning improves performance compared to conservative 2PL, especially for the workloads with high conflict rates (the TPC-W ordering mix and the auction site). We have compared distributed versioning to various replication methods which only provide loose consistency guarantees. We find that for all our applications, except those with very high conflict rates, the performance of distributed versioning equals or approaches that of looser consistency models. For all workloads with a high
rate of conflicts, the loose consistency model that represents the upper bound does not satisfy the application’s requirement. A specialized consistency model that still respects each application’s requirement improves performance over our distributed versioning protocol only marginally (1\%) for all workloads.

The cost of providing fault tolerance and data availability is no more than 16\% for all workloads.
Chapter 7

Related Work

Our work builds on previous research in many different fields such as: cluster research for general purpose cluster servers, cluster servers for static content, cluster servers for dynamic content, cluster database servers, replication in database systems, caching dynamic content, and multiversion database concurrency control. In the following we briefly describe the most relevant work in each area.

7.1 General Purpose Cluster Servers

Clusters have been the focus of much research and several commercial products are available. Older commercial systems, such as Tandem [13] and Stratus [82], use custom-designed hardware. More recent efforts, such as Microsoft Cluster Service [92] and Compaq’s TruCluster [24], use commodity parts. While these systems address scalability and availability issues in cluster servers, they do not address replication for persistent data, nor do they address applicability to dynamic content web applications which is the focus of our work.

7.2 Cluster Servers for Static Content Web Sites

Network servers based on clusters of workstations are now widely used for serving static content [25, 33, 39, 49, 66, 99]. Cardellini et al. [23] provides a comprehensive survey.

Most request distribution strategies employed are variations on weighted round-robin request distribution (e.g., CISCO Local Director [33] and IBM Network Dispatcher [49]). The incoming requests are distributed in round-robin fashion, weighted
by some measure of the load on the different back-ends. This strategy produces good
load balancing among the back-ends. However, since it does not consider the type of
service or requested document in choosing a back-end node, each back-end node is
equally likely to receive a given type of request. Thus, frequent cache misses occur if
the working set exceeds the size of main memory available for caching documents.

Locality-Aware Request Distribution (LARD) [8, 10, 66] was developed and shown
to be successful for load balancing static content requests in a cluster. The goal of
LARD is to combine good load balance and high locality for increased hit rates in
the data caches of each back-end.

Currently, commercial content-aware (Layer-7) web switches such as the Dispatch
product by Resonate, Inc. [73], and Nortel’s WebOS server load balancer [63] have
become popular. They work at the application level, thus allowing request distri-
bution based on some affinity-based scheduling algorithm. Loosely-coupled distributed
servers are also widely deployed in the Internet and use various techniques for load
balancing including DNS round-robin [21], smart clients [97], source-based forwarding [35] and hardware translation of network addresses [33].

While these techniques have been shown to perform well for static content, the
results cannot be extrapolated to dynamic content and different techniques are nec-
essary. Intuitively, the need for different techniques arises from the basic differences
between static and dynamic content. The latter is typically more CPU intensive, has
much more locality, and, above all, contains updates to the content. This shifts the
focus from locality to consistency maintenance. Our work recognizes data conflicts as
the primary scaling bottleneck for dynamic content applications and addresses this
bottleneck by a conflict-aware scheme.

More recent techniques that are applicable to both static and dynamic content
serving include load balancing strategies that seek to satisfy response time based per-
formance criteria [77] and resource management schemes that provide differentiated
as well as predictable quality of service in terms of application-level metrics such as
average request rate [9, 80]. These techniques are orthogonal to ours, and could be used in combination with conflict-aware replication.

Casalicchio et al. [26] propose a new scheduling policy, called client-aware policy (CAP), for web switches operating at Layer-7. CAP is intended for cluster-based web servers that provide a variety of services at the same time, such as, static, dynamic and secure content. CAP classifies the client requests on the basis of their expected impact on main server resources (i.e., network interface, CPU, disk, etc). The service class for each request is known in advance. The web switch uses this classification when scheduling client requests with the goal to avoid any resource bottlenecks. They show that for web servers with heterogeneous workloads, CAP outperforms LARD techniques. CAP is similar to our SELF load balancing policy in that the scheduling choice is based on a priori knowledge. On the other hand, our focus is on reducing the consistency maintenance overheads inherent in replicating the data for scaling the same service, and our scheduling technique combines load balancing and conflict-awareness.

7.3 Cluster Servers for Dynamic Content Web Sites

Current high-volume web servers such as the official web server used for the Olympic games [29] and real-life e-commerce sites based on IBM’s WebSphere, Commerce Edition [51], rely on expensive supercomputers to satisfy the volume of requests. Nevertheless, performance of such servers may become a problem during periods of peak load. Hence, this brute-force approach of ever increasing the capacity of a stand-alone data server seems an unlikely solution for the scalability problem in the foreseeable future.

Luo et al. [60] and Oracle’s 9i Database Cache product [65] use a middle-tier database cache. They rely on replication tools to periodically propagate updates from the back-end database to the cache tier. This approach is orthogonal to ours, although it achieves the same goal, that of scaling the database tier. For applications
that allow out-of-date data, we could also benefit from a similar update propagator. On the other hand, our work attempts to parallelize even e-commerce interactions that need strict consistency. Thus, we need control and accurate information on exactly when updates occur at each database.

Recent work [81, 98] avoids paying the price of serializability for applications that don’t need it by providing specialized loose consistency models. Neptune [81] adopts a primary-copy approach to providing consistency in a partitioned service cluster. However, their scalability study is limited to web applications with loose consistency such as bulletin boards and auction sites, where scaling is easier to achieve. They do not address e-commerce workloads or other web applications with relatively strong consistency requirements.

Zhang et al. [99] have extended locality-aware request-distribution from static servers to dynamic content in their HACC project. Their study, however, is limited to read-only content workloads. In a more general dynamic content server, replication implies the need for consistency maintenance.

Our work is also related to the work of Ji et al. [52]. They cache raw database data in a set of separate cache servers. They use affinity-based query distribution in order to improve cache hits and minimize synchronization cost. As we have shown [4], caching raw data has limited benefits for typical dynamic content workloads (e.g., e-commerce workloads) due to the high query computation costs, and high workload locality.

Frolund et al. [40] implement the abstraction of exactly-once transactions in a three-tier web server. They employ a solution where the application server tier runs on a cluster and performs state replication and coordination. The implementations of the fault tolerance and data availability aspects of our solution have similarities with their work, but the goal of our system is different. Furthermore, their algorithm requires that part of the software is run on the client machine, while our algorithm is only deployed at the server.
7.4 Partitioning in Cluster Database Servers

Our scheduler is related to load balancing schemes used in cluster database systems [34], although the approaches are orthogonal. We use replication instead of declustering for data placement.

The traditional approach to load balancing in cluster database systems has been that data placement drives the load balancing. The name space of the database is partitioned in some way, and queries for all objects in a particular partition are assigned to a machine or set of machines. The objects or parts of them may be cached in memory or solely reside on disk on the above set of machines.

Finding the optimal placement of data across database nodes for good load balancing is an NP-complete problem. Usually the heuristics used in data placement take into account several characteristics of the objects and of the queries performed on the objects. Ideally all components of work (e.g., CPU, disk I/O) would be balanced. For example: A join on a relation typically requires more compute time than a select. A large table may involve more disk accesses. Some commercial systems, such as DB-2 [48], require the database administrator to specify the number of machines to use for each table and the exact data partitioning. Other academic systems, such as Bubba [34], that attempt to do the placement dynamically, use as heuristics the frequency of access for each object and the size of the objects. Thus, at intervals, the data placement’s efficiency needs to be probed by looking at the processor utilization and the disk utilization of backend nodes. If bottleneck backends where utilization is too high exist, a data reorganization step is performed.

7.5 Replication in Database Clusters

In practice, replication has previously been used mainly for fault tolerance and data availability [38, 68]. Gray et al. [41] shows that classic solutions based on eager (synchronous) replication which provide serializability do not scale well. Lazy replication algorithms [7, 45, 54, 74, 89] have been proposed as an alternative. They
asynchronously propagate replica updates to other nodes, possibly after the updating transaction commits. Most lazy techniques [45, 89] are used in mobile or wide-area environments. They scale well, but also expose inconsistencies and stale data to the user. This is because, in lazy replication, replica updates have already been committed when a serialization problem is first detected. There is usually no automatic way to reverse the committed replica updates, rather a program or person must reconcile conflicting transactions. The only guarantee that lazy replication algorithms usually offer is eventual convergence of all replicas to a consistent image. Microsoft Access [11] offers convergence as follows: It has a single design master node that controls all updates to the replicated database. Each node keeps a version vector with each replicated record. These version vectors are exchanged on demand or periodically. The most recent update wins each pairwise exchange. Rejected updates are reported. Lotus Notes [53] also uses a lazy replication design, providing convergence rather than an ACID transaction execution model.

Oracle Advanced Replication [64] supports both lazy and eager replication. It uses database triggers where a data transfer is triggered upon every update. Conflict detection can be done automatically using timestamps, but the user decides which is the correct copy.

Recent work [7, 54] has explored the possibility of using lazy replication while still providing serializability. This approach is, however, primary-copy and has inherent restrictions regarding accessing data. As in other primary-copy lazy replication schemes [93], each operation on a particular object is sent to a primary copy which decides the serialization order. The down-side of all primary-copy approaches is that they depend on the availability of the primary copy. Furthermore, the only possible load balancing in this type of scheme is based on a partitioning of the objects on master copies. In contrast to primary-copy replication schemes, in our scheme the scheduler decides the serialization order, the scheduler’s state is replicated, and each operation can be sent to any replica.
Breitbart et al. [19] propose a solution where lazy propagation is applied along the acyclic paths of the graph, while eager propagation is used whenever there are cycles. Anderson et al. [7] also present a primary-copy approach combining eager and lazy techniques. The system is eager since the serialization order is determined before the commit of the transaction using a global serialization graph. The system is also lazy, however, because within the boundaries of the transaction, the execution of the writes only takes place at one site. Propagating the updates to the other sites is only done after commit.

More recently, eager replication based on group communication has been proposed [87, 94], and optimized [55, 56, 67], to tackle the same problem: providing serializability and scaling at the same time. These approaches are implemented inside the database layer. Each replica functions independently. During the transaction execution, the local database acquires the proper locks for the read and write accesses on its own machine, performs the operations, then sends the other replicas the write-sets through group communication. Conflicts are solved by each replica locally. This implies the need to abort transactions when a write-set coming in conflicts with the local transaction. Approaches that eliminate the need for transaction aborts exist [56, 87], but they either imply relaxing the serializability requirement [56], or introducing an acknowledgment coordination phase between all replicas, at the end of the transaction [87].

These approaches differ from ours in that we consider a cluster in which a scheduler can direct operations to certain replicas, while they consider a more conventional distributed database setting in which a transaction normally executes locally. Also, our approach is implemented outside of the database.

The goals of the Ninja project [44] are similar to our own: to provide incremental scaling, fault tolerance and high availability, high concurrency and consistency for Internet sites by using clusters. Their approach is, however, different. They argue that conventional databases have not been designed with the needs of Internet workloads
and cluster platforms in mind. Thus, they provide a self-managing cluster-based
data repository [43], called a scalable distributed data structure (DDS), for use in
building new Internet services, instead of relying on a conventional database tier. The
application accesses the DDS in the same way as any regular in-memory data structure
(e.g., hash table), while the system handles the persistence and distribution of data
across the cluster transparently. The underlying replication protocol is optimized for
cluster platforms and most common application needs. For instance, they use an
optimistic two-phase commit protocol [16] among the replicas which provides atomic,
but not transactional updates.

A lot of current work [14, 18, 83, 90, 95] is focused on scaling the application
server tier in dynamic content servers using the Java 2 Platform Enterprise Edition [88] (J2EE). Major J2EE vendors such as BEA [14] and IBM [95] use application
server clusters to achieve scalability and high availability. The Middleware Technology Evaluation project [47] rigorously evaluates the scaling of the application server,
on single and dual node platforms, for different middleware technologies, such as
WebSphere [95], WebLogic [14], Borland [18], SilverStream [83] and JBoss [90].

TP-monitors such as BEA’s Tuxedo [15, 42], are also superficially similar in func-
tionality to our scheduler.

These systems provide support for replicated application servers and databases
replicated for availability, not for databases replicated for performance as in our sys-
tem. Their multi-database transactions provide programming support for applica-
tions with transactions accessing different databases (using conventional two-phase
commit) and not transparent support for a large number of replicas of the same
database.

7.6 Caching Dynamic Content

Another approach to scaling dynamic content sites by alleviating the database bot-
tleneck is dynamic content caching. This approach is orthogonal to ours, since it
allows scaling while using the same hardware configuration, while we explore scaling by replication on clusters. In this section, we provide a brief overview of dynamic content caching, and describe a few relevant approaches.

Caching dynamic content has been studied mainly in stand-alone dynamic content servers [27, 29, 71]. Dynamic web data can be cached at different stages in its production: the final HTML page (e.g., [22, 57], intermediate HTML or XML fragments (e.g., [36]), database queries (e.g., [61]), or database tables (e.g., [60, 65]). Combination of various caches are also possible (e.g., [28, 96]). Intuitively, caching at the database stage typically offers higher hit ratios, while caching at the HTML or XML stage offers greater benefits in the case of a hit. There is no conclusive evidence at this point that caching at any single stage dominates the others. For instance, Labrinidis and Roussoulo use a synthetic workload and conclude that HTML page caching is superior [57], but Yagoub et al. use TPC-D and conclude that database query caching is more effective [96]. It appears that the different caches are complimentary [72, 96]. For instance, Rajamany [72] uses caching at multiple levels: a dynamic content cache for HTML pages is integrated with the web server through a dynamic cache extension module, while the database engine is enhanced with a query result cache. Their quantitative evaluation uses the same e-commerce benchmark we are using, TPC-W, and shows aggregate performance benefits from using these caching schemes together.

IBM’s Web Accelerator [59] combines web page caching with content-based scheduling in a web server cluster. The Web Accelerator runs on the same node as the IBM Network Dispatcher [49]. When a client attempts to connect to the cluster web server, the Accelerator accepts the connection and the client request. If possible, this request is served out of an in-memory cache on the Dispatcher. In the event that there is a cache miss, the Dispatcher contacts a server node, issues the client request, then caches the response and forwards the response back to the client.
7.7 Multiversion Concurrency Control

Concurrency control protocols based on multiple versions have been discussed [16] and implemented in real systems [70] to increase concurrency while maintaining serializability in stand-alone database systems. More recently, multiversion ordering [12, 50, 75, 79] has been used and optimized for distributed database systems as well. Most of these systems use transaction aborts to resolve serialization inconsistencies. Some systems targeted at advanced database applications such as computer-aided design and collaborative software development environments [12] use pre-declared write-sets to determine if a schedule conflicting at the object level can be serialized, thus avoiding transaction aborts.

Such systems maintain a history of old versions at each distributed location and need a special scheme for reducing version space consumption and version access time [32], or limiting the number of versions stored [50]. Furthermore, if replication is used at all in these distributed systems, the goal is to increase the availability of a particular version [75]. In contrast, in our versioning concurrency control algorithm used in distributed versioning [6] we do not maintain old copies of items, all modifications are made in place. The goal of our extra-database algorithm is to allow us to choose the correct version among the different versions of a table which occur naturally due to asynchronous replication. On the other hand, multiversion systems have the advantage that the execution of read-only transactions can be made more efficient by completely decoupling their execution from update transactions [50, 75].
Chapter 8

Conclusions and Future Work

In this dissertation, we present novel conflict-aware lazy replication techniques in dynamic content sites using a cluster of web servers and database back-ends. Serializability is maintained by assigning a total order to pre-declared table accesses. The scheduler maintains state about conflict-free replicas, and load balances read queries only among these replicas. This approach avoids modifications to both the web server, and the database engine. We assume software platforms in common use: the Apache web server, the MySQL database engine, and the PHP scripting language. As a result, these scaling methods are applicable without burdensome development or reconfiguration.

We have evaluated conflict-aware scheduling, both by measurement of an implementation and by simulation. We use the various workload mixes of the TPC-W benchmark and two additional web server benchmarks modeling an auction site and a bulletin board site, respectively, to evaluate overall scaling behavior and the contribution of the conflict-aware scheduling.

We find that query-level conflict-awareness brings factors of 1.5, 1.6, 2, and 1.9 improvement in terms of throughput, compared to a conflict-oblivialous lazy scheme, (and 1.7, 2.4, 3.5, and 2.6 compared to an eager scheme) at the largest experimental configuration for the TPC-W browsing shopping, ordering mixes and the auction site benchmarks, respectively. For the bulletin board benchmark, improvements are limited to 10%, because the application does not require strong consistency, thus does not contain transactions. Our simulations show that a prototype cluster architecture using conflict-aware scheduling scales well up to the maximum simulated configuration
for the most representative of the TPC-W workload mixes, the shopping mix, and the bulletin board benchmark and up to 40 databases for the auction site benchmark. Our simulations also show that conflict-aware schedulers scale well to faster machines, and that they maintain an edge over eager and conflict-oblivious schedulers even if the conflict rate is much lower.

Conflict-aware scheduling introduces low overheads for enforcing replication with serializability for applications with looser consistency semantics. Hence, specialized consistency models are of little benefit for these applications. Furthermore, conflict-aware scheduling reduces the unavoidable overheads due to conflicts for applications with strong consistency semantics, where specialized consistency models have less applicability.

Whether specialized consistency models are still useful in a wide-area network or when dynamic content caching is used remains an open research question. In particular, staleness control has been traditionally used in web services whenever caching for static (or dynamic) content is employed [71]. Hence, a legitimate question is: Even if the benefits of specialized consistency models are minimal for supporting replication, should a specialized consistency model be used anyway to support dynamic content caching? Thus, further research is needed to investigate a combined scheme where dynamic content caching is used in addition to content replication on clusters, and to investigate the applicability and overheads of the conflict-aware protocol in wide-area environments.

While the focus of this thesis has been to show that commonly used software can be used to scale the system without any modifications, this also implies some limitations to the current scheme. A promising avenue for future work is a possible integration of conflict-awareness with the internal database consistency maintenance policies, such as fine-grain locking. In particular, once the scheduler establishes an order for the operations, an optimized form of batching at the database could be used to reduce the actual number of conflicts between operations to the number of
fine-grained conflicts detected at the database.
Bibliography


